UNIVERSITY OF WARWICK

DEPARTMENT OF COMPUTER SCIENCE

CS344 DISCRETE MATHEMATICS PROJECT

An investigation in using reinforcement learning for betting through the game of blackjack

Author

Patrik GERGELY

Supervisor

Prof. Artur CZUMAJ

May 4, 2021

Abstract

Although many tutorials use blackjack as an introductory problem to teach reinforcement learning, these tutorials are only concerned with a simplified version of the game, such as the Blackjack-v0 environment by OpenAI. This project designs a framework that allows the implementation and evaluation of different blackjack playing bots. Within the framework, bots are composed of a bettor, deciding on bet sizes before games, and the strategist which aims to win the games. Bots are evaluated based on their long-term performance by interacting with a custom blackjack environment, that better resembles the game than other environments. The included three bettors and two strategists provide a baseline when evaluating newly researched bettors or strategists. Using the framework, the project aims to implement and train a bettor that uses reinforcement learning to choose bet sizes, in contrast to the aforementioned tutorials, which solve the strategist component of the problem. An explanation of the parallels between the bet sizing process in blackjack, and the portfolio management problem, creates the potential for the conclusions here drawn to be extrapolated to financial investment strategy.

Keywords: betting, blackjack, logarithmic utility, long term investing, portfolio allocation, reinforcement learning

Contents

1 Introduction

1.1 Problem statement

The following investigation aims to evaluate the performance of multiple blackjack playing bots by implementing an environment that the bots can interact with. Blackjack comprises two stages, first the player places a bet with the dealer, then the player repeatedly chooses from the actions splitting, doubling down, hitting and standing, until either the player or the dealer wins the placed bet. As the betting stage is distinct from performing well in the game, the project will aim to solve these independently. The aforementioned bot will comprise a bettor, capable of determining appropriate bet sizes and a strategist, capable of selecting actions within the game in order to maximize its chance of winning the bet. The project will involve implementing two different strategists and five different bettors, two of which will be reinforcement learning based. Ten different bots (given by the combination of the two strategists and five bettors) will be evaluated by an approximation of how well each optimizes the utility $\mathbb{E}[\log(x)]$

1.2 Problem motivation

In gambling a person is proposed with a game where after placing a bet they receive a reward based on some random factor. In investing, the investor has some perceived understanding about price changes of a security and they need to decide on the proportion

of their assets they want to invest in that security and afterwards their allocated asset will either grow or decrease similarly to gambling. For this reason betting strategies, such as the Kelly criteria, have been used in investing for a long time [Thorp, 2008]. This paper aims to investigate betting strategies that are based on reinforcement learning techniques, which could act as a proxy for a reinforcement learning agent used for investing. Section 3.3 describes multiple reinforcement learning models which aim to be used for investing, but these all depend on historical data, while the proposed agent depends on the environment instead, which is a better predictor for future trends. The game of blackjack is chosen as the proxy, as it is relatively well studied because of its popularity. The game is hosted by casinos, which implies that most strategies played by humans will lose money in the long term. However, so-called card counters aim to win money on blackjack by keeping track of the discarded cards and vary their bet sizes based on this information. [Thorp, 2008] presents how to use the Kelly criterion to produce bet sizes for the game of blackjack, promising optimal return for the long run. Having three different betting strategies, one employed by most humans, one developed by mathematicians and employed by people trying to earn money playing blackjack and one that uses computers to compute optimal bet sizes, present baselines that can be used to evaluate new betting strategist. Although the purpose of the investigation is finding a betting strategy, which already has an optimal strategy (which has been implemented by this project), investigating how reinforcement learning performs in this setting still poses interesting results, when blackjack is only thought of as a proxy.

1.3 Challenges

One of the main introductory exercises in reinforcement learning is to implement a blackjack playing agent, posing the question why is this project more challenging than the agents available in brief articles? The distinction between the blackjack playing agents available and the bot proposed by this project is that, while existing ones solve the strategist part of the game, while ignoring the betting aspect of the game, the proposed bot solves the bettor using reinforcement learning. These two stages of the game are distinct, therefore implementing a reinforcement learning based bettor will differ from implementing only a strategist. Another difference is that most of these projects, such as [Dickson, 2019] and [Solai, 2020] use an environment called Blackack-v0 [OpenAI, 2016], which is a simplified version of the game (only providing 2 out of the 4 moves in the game, without the capability of placing bets), consequently the project will require the implementation of a blackjack environment that better captures the game. Other projects, such as [Henighan, 2019] use their own blackjack environment, but these are even more simple than the one provided by OpenAI. Setting up an environment that perfectly replicates the rules of blackjack played in most Las Vegas casinos is a challenging software engineering task, as different casinos have different rule variations to gain advantage over the players. The project is research-wise challenging as it requires knowledge in a wide domain. First, it is required to find a utility that optimizes for long-term results and a metric to measure and compare the performance of different bots. Second, research is needed in the current state-of-the-art bettors and strategists in blackjack to set up baselines. Third, research is required in reinforce-

ment learning algorithms to find and assess potential agents for the reinforcement learning bettors. Fourth, an efficient way to tune and train these agents is required. Finally, to run training and tuning on the Computer Science Departments machines, an understanding of the SLURM Workload Manager is required.

2 Blackjack rules

Before a blackjack game can start, the player is required to place a bet against the dealer. After the bet has been placed, the game is started by the dealer dealing two card face up to the player and placing one card face up and another card face down in front of themselves. The aim is to get a hand total closer to 21 than the dealer without going over 21 (*busting*). The hand total is calculated by adding the values of each card together. Face cards are valued at ten, aces can either be valued at 1 or 11, whichever is more favorable, while other cards are valued at their pip values.

After the initial setup, the player has the option ask for as many cards as they want, by hitting. If after hitting the player gets a hand total over 21, they bust and instantly lose their wager. Otherwise, they have the option to hit again or stand, in which case they freeze their hand total and it is the dealer's turn to draw cards. The dealer first reveals their face-down card, then they draw new cards as long as their hand total is 16 or under. If the dealer busts the player wins the wager, otherwise the wager is won by whoever is closer to 21. In case

the hand total of the player and the dealer are equal, the outcome of the game is a push and the original bet gets returned to the player with no additional reward.

If the player did not hit yet, they have the option to double their initial bet by doubling down. Here, the player receives exactly one more card before freezing their hand total.

If the player is dealt matching cards, they have the option to split them and use them as separate hands. After splitting a hand, the player needs to place a bet equal to the original wager on the newly created hand. These hands compete against the same dealer independently, and the dealer only reveals their cards after both hands of the player stood.

The rules above describe the general rules of blackjack, but casinos use different rule variations in order to fine tune their advantage over the player. The following explains the specific rules of Vegas Strip, one of the most popular blackjack rule variation [Reynolds, 2021].

- **Four deck shoe** In blackjack, the dealer draws the card from a holder called the shoe, which in Vegas Strip contains four decks of cards. After shuffling and placing the cards into the shoe, a white plastic card is inserted somewhere into the deck, which when drawn signals for the cards to be reshuffled. In the implemented blackjack environment, this always happens when there are only 52 cards remaining in the deck.

- **Naturals pay** 3 : 2 If the player's initial hand total with two cards is 21 it is called a

natural or “blackjack” in which case they automatically win the wager, which in this case pays 3 : 2.

- **Dealer peeks** If the dealer is dealt a 10 valued card or an ace, they check their hidden card and if the two add up to 21 or not. If they do, the game is automatically over, in which case the player loses the wager (unless they had a blackjack, which would cause a push).

- **Stand on soft 17** A hand total that with an ace that can be valued at either 1 or 11 is called a soft hand. When the dealer has a soft hand, the hand total is calculated by using valuing ace as an 11. When the dealer has a hand total of 16 or under, they are required to take an additional card. When the dealer has a hand total of 18 or over, they are required to stand. When the dealer has no ace that can be valued as 11, they have a hard hand. Hard 17 is always required to stand, but whether soft 17 hits is up to the rule variation, in Vegas Strip the dealer stands soft 17.

- **Splitting is allowed for all cards** Some variations only allow to split specific cards such as eights, nines and tens, but Vegas Strip allows the player to split all cards.

- **Splitting uneven** The player may split different ten valued cards, such as a ten and a queen.

- **Splitting aces** After splitting aces and both hands getting dealt an additional card, both hands are required to stand.

- **Blackjack with split aces** Drawing a ten valued card after splitting aces does not count as a natural and does not pay 3 : 2. However, drawing an ace after splitting tens pays 3 : 2
- **Re-splitting** The player may re-split their hands up to three times for every card other than aces. Aces may not perform any action after splitting, including splitting.
- **Doubling down after splitting** The player may double down after splitting non-ace cards.

Although insurance and surrender are sometimes offered to player, as these moves are considered being suboptimal, the environment does not implement them.

3 Background

3.1 Geometric Mean Maximization

In gambling, the bettor faces a crucial question before they even can start to set up a betting strategy for a game. This is about determining the goal utility they optimize for. One possible goal could be to optimize for the highest immediate reward, which results in betting the entire wealth on any game that has a positive expected value and nothing for games with a negative expected value. It seems justified to not take bets with negative expected values, but betting one's entire wealth on each game with a positive expected value

promises near certain bankruptcy on the long run. [Breiman, 1961] suggests maximizing for the expected logarithm and proves that doing so will both minimize the expected number of games needed to obtain a predetermined goal and maximize the bankroll after a predetermined number of games. Although any utility chosen will be arbitrary and in investing the general consensus is to maximize risk-adjusted returns measured by the Sharpe ratio, [Estrada, 2010] argues, that the maximum expected log, also known as geometric mean maximization is a good alternative to the Sharpe ratio for investing. However, optimising for $\mathbb{E}[\log(x)]$ poses an issue, that the function is unbounded form below. For this reason the project will aim to maximise a shifted version of the logarithm function: $\mathbb{E}[\log(x+1)]$. To evaluate different bots based on how well they maximize the expected logarithm, an approximation is needed for $\mathbb{E}[\log(x+1)]$. The following formula given by [Markowitz, 1959] is going to be used to approximate a bot's performance. Suppose that the bot has a fortune of S_i at time i (and 1 at time 0) and define $r_i = \frac{S_i}{S_{i-1}} - 1$. Using r_i the following formula can be given for the wealth at time i, $S_i = \prod_{j=1}^{i}(1+r_j)$. The logarithm of the wealth at time i is given by $\log(S_i) = \log\left(\prod_{j=1}^{i}(1+r_j)\right) = \sum_{j=1}^{i}\log(1+r_j)$. Now assuming that the rewards r_i do not change throughout time $\mathbb{E}[\log(S_n)] = n\mathbb{E}[\log(1+r)]$. [Markowitz, 1959] compares multiple approximations for $\mathbb{E}[\log(1+r)]$ and suggests the use of $\mathbb{E}[\log(1+r)] \approx \log(1+\mu) - \frac{\sigma^2}{2(1+\mu)^2}$, where μ is the expected value of r and σ^2 is the variance of r. As $\mathbb{E}[\log(S_i)]$ is exactly n, the number of games, more than $\mathbb{E}[\log(1+r)]$ the bot will be evaluated using the approximation $\mathbb{E}[\log(1+r)] \approx \log(1+\mu) - \frac{\sigma^2}{2(1+\mu)^2}$ based on a realization of a given bot's interaction with the environment.

3.2 Blackjack

Blackjack contains two separate phases: betting before games and playing the actual game. Both have literature about strategies that make it a suitable candidate to test out new betting strategies. Card counting is when a player keeps track of discarded cards and uses that information to approximate their chance of winning and based on this information increase or decrease their bet sizes[Tamburin, 2017]. The bettor pairs numbers (usually ± 1) to each card value and every time a card gets discarded they add the corresponding value to the so called running count. The idea behind this is that high cards are beneficial to the player while low cards are beneficial to the dealer, so keeping track of the ratio of the discarded cards (and thus the remaining cards), the card counter can approximate whether they or the dealer are advantageous. [Vidámi et al., 2020] evaluates multiple of these strategies, each assigning different values to the cards, and concludes that one of the most well known and easiest to learn, Hi-Lo strategy compares well to the other strategies. For this reason, the Hi-Lo strategy will provide one of the baselines for the bettor component. These card counting strategies are essentially all the same, where the bet size is calculated by taking the dot product of the discarded cards and a second vector describing how valuable each card is to the player. Taking the linear combination of the environment, which are the discarded cards in this case, with some pre-defined weights to calculate an output resembles a single-layer neural network, suggesting that a well trained neural network based bettor should be at least as capable as the Hi-Lo method. [Thorp, 2008] applied the Kelly criterion to the game of Blackjack by calculating the probability of each payout then by finding the

roots of the derivative of the utility function to find the optimal bet size. Although Thorp explains the design of such a bettor, his paper provides no evaluation of such a bettor. This investigation evaluates and compares multiple bettors, including the bettor proposed by [Thorp, 2008].

When playing the game, most card counters do not vary their strategies, instead they follow a so-called basic strategy [Shackleford, 2019], which is a lookup table mapping the Cartesian product of the player's and dealer's hands to the optimal action while disregarding any information about discarded cards. However, when a bot has complete information on the discarded cards, and thus the distribution of remaining cards, they can calculate more appropriate actions by taking every possible deck ordering into account and weighting them by prevalence. This investigation will also aim to compare this basic and optimal strategy to each other.

3.3 Reinforcement Learning

As described by [Sutton and Barto, 2018], the purpose of reinforcement learning is for an agent to learn how to act in order to achieve some long-term goal through interaction with an environment. This interaction happens over a sequence of time steps, where the agent takes an observation of the state of the environment, chooses an action to perform which changes the state of the environment for which the agent receives some immediate reward. The goal of the agent is to obtain a policy, a stochastic rule that maps states to actions, which

maximizes the cumulative reward achieved over time. In the case of betting in blackjack, the environment consists of the deck of cards and the chips available to the agent, the action space is any real between 1 and the number of chips available and the reward is the payout for a given game. The observations taken by the agent comprise the discarded cards and the amount of chips available. The next state after performing an action, placing bets, depends on the actions taken by the strategist, for example, a strategist that always stands, will result in less cards being drawn than a strategist that always hits. Subsequently, the transition function between states is not well defined for this environment, thus whenever the paper talks about a bettor, it is in conjunction with a fixed strategist. As strategists follow a deterministic algorithm, for a fixed strategist the transition function is well defined and therefore it constitutes as a Markov decision process.

As shown previously, the bettor phase of blackjack can be thought of as a Markov decision process and therefore a reinforcement learning agent can learn to produce appropriate bet sizes by interacting with the environment.

Although there are multiple papers investigating the use of machine learning for investing, such as [Jiang et al., 2017] and [Sato, 2019], these aren't directly applicable as the project aims to take the observations of the discarded cards as input, while the aforementioned papers have an architecture that looks at historic data, with no additional information about the current state of the environment. Historic data might be useful to find trends in price changes of securities, but whether the player is advantageous in blackjack changes

rapidly and only slightly in the game of blackjack, so looking at historic data can't be used to predict the future performance of the strategist. As casinos never want players to be overly advantageous, blackjack is played with multiple decks of cards and shuffled regularly in order to keep the distribution of cards close to uniform. For this reason, even when the player is advantageous, it is only a slight advantage and by the time this advantage could be calculated based on historical data, the advantage will change.

As mentioned in section 1.3, there are multiple projects implementing blackjack playing reinforcement learning agents. One of the most detailed investigation into blackjack playing agents is provided by [Mao, 2019], implementing 6 different agents, but these agents focus on providing actions inside the game. This paper proposes to allow agents to use different 3 different bet sizes, but it contains a single action space used for both choosing actions inside a game and bet sizes. The results presented by Mao suggests, that such an architecture is incapable of producing appropriate bet sizes. This investigation uses reinforcement learning only to produce best sizes, with the ambition that this simplified problem can be solved using machine learning.

4 Tools

4.1 Cython

The project will be implemented using Python [Rossum and Drake, 2009] as this programming language has the highest number of machine learning libraries and frameworks. However, Python falls short of other languages, such as C when it comes to performance, as Python is an interpreted language. This problem can be addressed using either ahead-of-time (AOT) or just-in-time (JIT) compilation. This project will use Cython [Behnel et al., 2011], an AOT optimising static compiler that supports calling C functions and declaring C types. Statically typing the variables allows the compiler to generate efficient C code, which will increase the speed when compared to Python. This generated C code then can be compiled in CPython to integrate the Cython code with the rest of the Python code. Besides statically typing variables, the project leverages Cython syntax to use C variables, for implementing efficient lookup tables using unordered_map and storing arrays on the heap using pointers and thus saving memory usage.

4.2 SLURM

The produced code will not be feasible to run on a personal computer, as an evaluation of different bots will require numerous simulations, additionally training machine learning models require CUDA [NVIDIA et al., 2020] supported GPUs in order to run efficiently.

Both concerns can be managed, by running the code on the University of Warwick, Department of Computer Science provided batch compute system. The cluster provides 3 partitions, cpu-batch containing nodes with 20-core CPUs, gpu-batch containing nodes with 64GB of RAM and a RTX2080Ti GPU and desktop-batch containing nodes with i5 CPUs, at least 16GB of RAM and a GTX 750Ti or GTX 1050Ti CUDA supporting GPU. The project will utilize the desktop-batch, as the project does not require more cores or better GPUs than the ones provided in the desktop-batch partition. The batch compute system uses the SLURM [Yoo et al., 2003] job scheduling system, allowing users to requests GPUs to tune and train machine learning models or to run simulations in parallel, by running one simulation on each core and on multiple nodes. Being able to run simulations in parallel is crucial, as some simulations require over a day to halt. As each node contains 4-6 cores on the desktop-batch and the partition contains over 150 nodes, using SLURM can decrease simulation time by nearly a thousandfold.

4.3 Bayesian Optimization

Deep learning systems depend on several hyperparameters, such as the number of hidden layers used by the underlying neural network, the number of nodes in each layer or the learning rate used to update the weights with. Although these hyperparameters can be set by developers, it is more common to be determined using hyperparameter optimization or tuning. [Hutter et al., 2019] defines a validation protocol $V(\mathbb{L}, \mathbb{A}_\lambda, D_{train}, D_{valid})$ that

measures the loss $\mathbb{L}$ of a model generated by an algorithm $\mathbb{A}$ with hyperparameters λ on training data D_{train} and evaluated on validation data D_{valid}. The project uses reinforcement learning which lacks both training and validation data and uses reward instead of loss, the definition of V can be modified to only depend on the hyperparameters, the algorithm used to generate the model and instead of a loss function use optimize for maximal cumulative reward. Using this definition the goal of hyperparameter optimization is to find $\lambda^* = \text{argmax}_{\lambda \in \Lambda} \mathbb{E}(V(\mathbb{A}_\lambda))$, where Λ is the hyperparameter configuration space.

[Hutter et al., 2019] describes Bayesian optimization, a state-of-the-art optimization framework, consisting of a probabilistic surrogate model and an acquisition function. The optimization algorithm is performed by a sequence of iterations, where the surrogate model is fitted to all previous observations, followed by the acquisition function determining the utility of different candidates using the surrogate model. After trading off exploration and exploitation of possible candidates, the Bayesian optimization trains and evaluates a model with the new hyperparameters and repeats the process. [Hutter et al., 2019] explains that when using Gaussian processes as the surrogate model, the optimizer fails to scale well, but under 250 function evaluations it performs well when compared to other optimization methods. As the project will not train over 250 models, it will use Bayesian optimization for hyperparameter optimization.

The project is going to use Tune [Liaw et al., 2018], a Python library for hyperparameter tuning at scale. Although Tune is capable of launching hyperparameter optimization on

a cluster and train models in parallel using Ray [Moritz et al., 2017], it mainly supports the use of cloud providers such as AWS, GCP or Azure. When launching Ray on departmental computers, they fail to communicate with each other. For this reason, the project will use BayesOpt, a Bayesian optimizer provided by tune, on a single computer.

4.4 ACME

The project is going to use ACME [Hoffman et al., 2020], a Python library for building research-oriented reinforcement learning algorithms. By exposing an environment and actor interface, ACME provides a framework to implement environments and actors capable of interacting with each other. The environment is based on the dm_env interface [Muldal et al., 2019], uses a method called step to take an action a_t from the actor, returning a tuple (r_t, o_{t+1}, e_{t+1}) object after modifying its internal state based on the action. The tuple consists of r_t, the immediate reward, o_{t+1}, the observation of the state after taking action a_t and e_{t+1} indicating whether the episode has terminated. Actors are implemented using comprises interface that uses a method called select_action that selects an action to take, based on some received observation. The framework handles the interaction between the environment and the actor using an environmental loop. ACME also support Reverb [Cassirer et al., 2021], an experience replay system for reinforcement learning algorithms, allowing agents to store past transitions and sample from these experiences while updating the internal weights, therefore requiring fewer samples per update. The project will utilize

the implementation of Deep Q-Network and Deep Deterministic Policy Gradient agents included in ACME.

5 Proposed solution

5.1 Outline

The blackjack bot will be written in python and implement the Actor interface provided by ACME and will interact with an environment implementing the dm_env package. This allows us to run simulations by providing the agent and environment to Acme's environment loop. Two strategists are implemented which are capable producing actions based on the discarded cards and 5 bettors all capable of producing bet sizes based on the discarded cards. The bot will memorize the discarded cards and contain a strategist and a bettor, and at each time step the bot will delegate its action to one of them depending on the state of the game. This modular design will allow to evaluate the combination of every strategist with every bettor. In order to prevent the agent from cheating and provide a more realistic blackjack environment, the environment does not include the shoe (nor the distribution of the cards remaining in the shoe), but rather similarly to how the game is played show the discarded cards to the agent at each time step. Different bots will be evaluated by simulating 900 games, each starting with 600 chips, and used the formula mentioned in section 3.1 to approximate $E(log(1+r))$. As this requires intense computation, the departmental batch

compute system was used to simulate these games using SLURM. As the desktop-batch computers all contain GPUs, these nodes will be used to tune and train the reinforcement learning bettors. For tuning purposes Ray will be used to perform Bayesian Optimization in order to search for the best hyperparameters. Afterwards reinforcement learning bettors will be trained using the found hyperparameters.

5.2 Project Management

The project is going to majorly employ the agile methodology with some principles taken from the waterfall methodology. One of the key benefits of the waterfall methodology is robust code, which in a project like this one, where the software is only used to simulate something, is unnecessary. If the code fails, it can be easily run again. On the other side agile enables faster code development and quick adaptation to requirement changes which matches the short and fixed lifespan of the project. Working in an area that is unknown initially will force shift to the requirements depending on the research. The agile method is a perfect fit for the structure of the program as the individual bettors and strategists are able to run independent of each other, allowing their development in one- to two-week long sprints. As mentioned before, the project started with some initial research similar to the waterfall methodology to determine an outline of the software, but additional research was performed before each sprint to better understand the components and slightly modify the requirements based on the additional information. As the code was running on the

departmental computers and developed on a personal computer, data redundancy helped against possible data corruption due to external factors. Additionally, as the departmental computers have GPUs, no external GPU enabled servers were required. To maintain good code health throughout the project, stylistic errors have been managed using pylint, unit tests were used to validate that units perform as expected using pytest [Krekel et al., 2004] and type safety was provided by annotating every variable, and checked for by pytype.

5.3 Timeline

The following two gannt charts show the proposed and the actual timeline of the project, which underwent some minor changes. Mainly the order of the sprints changed, which proved that the agile methodology was an excellent choice due to its adaptability. The projected started early and made constant progress even during the holiday. The original timetable included buffer periods, which were a tremendous help in avoiding falling behind schedule.

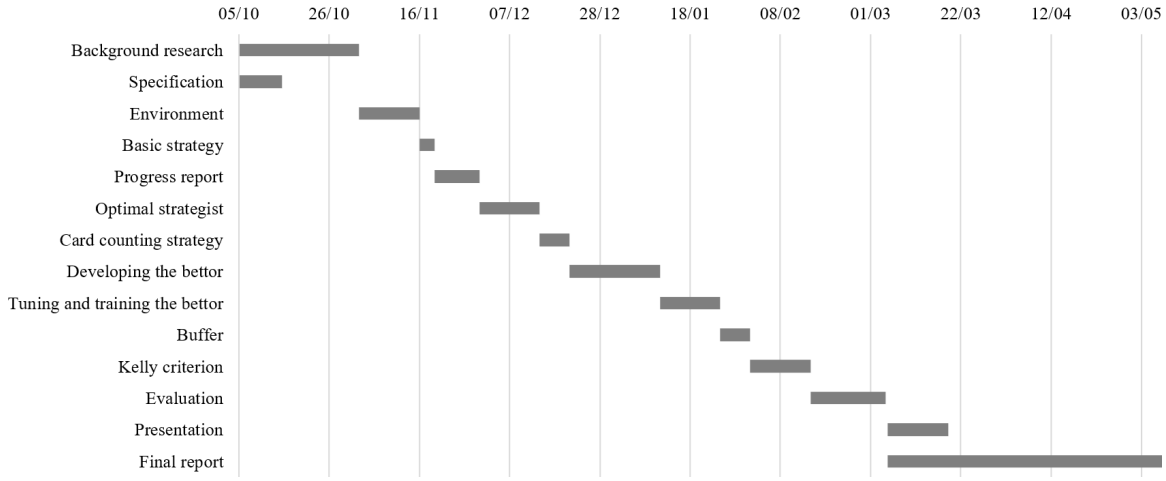


Figure 1: Timetable provided in progress report

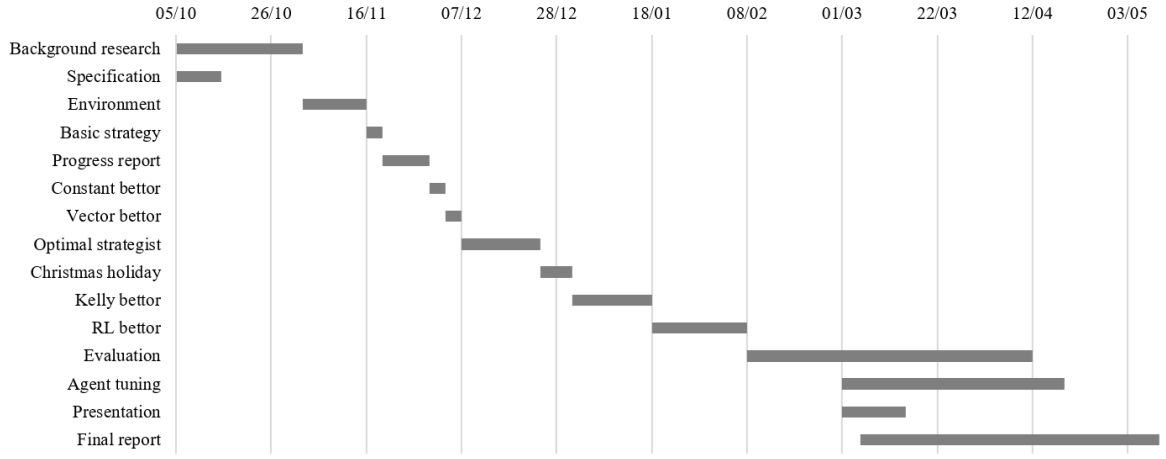


Figure 2: Final timetable

6 Overview

The implementation of both the environment and the bot follows the principles of object-oriented programming. Following the paradigm when implementing the bot is necessary to enable the evaluation between different strategies and bettors, by initializing the bot with different strategists and bettors. The object-oriented programming paradigm helped the implementation of the environment as the blackjack environment in the real world consists of objects such as players, dealers and games, each of which can be implemented using object-oriented programming.

6.1 Environment

Everything related to card drawing is handled within a shoe object, which internally shuffles the cards using numpy's random function, and then individual cards can be accessed one at a time using the draw method, which will be called by the environment. It is crucial that the cards are never exposed to the bot to avoid it cheating, thus the shoe array is hidden even from the environment. The environment keeps track of the cards that it just drew, and every time it sends an observation to the bot; it includes an array of the discarded cards to allow the bot to memorise the discarded cards. The deck automatically gets shuffled as soon as there are less than 25% of the cards left, in which case the environment sends a token to the actor to show that the deck is being reshuffled. The environment class is called Table, which holds a shoe object, the bankroll of the bot, and a game object. The game

object holds a single dealer object and one or more player objects, while keeping track of which player is in “focus”. The environment sends an observation of the game to the bot, which includes the recently discarded cards, the dealer’s hand and the player’s hand that is in focus and the next stage.

There are 4 stages:

- Placing a bet.
- Determining whether to split.
- Deciding whether to double down.
- Choosing between hitting and standing.

The received action is applied to the player in focus and potentially moves the focus to the next player. This terminates when the focus moves from the last player in which case it’s the dealers turn and the game is finished. The game object calculates the payout for that given game and returns it to the table, which multiplied by the last bet-size sends it to the bot as reward. Afterwards, the next action coming from the bot is treated as a bet-size and a new game will be started. This is done until either the bot goes bankrupt or a set amount of games have been played. This is here so an bot which never goes bankrupt can still be evaluated by assessing how it performed in a given number of games. The environment only allows the bot to take actions which can’t result in negative chips after the game by keeping track of the maximum possible loss during each game. This is important as the

utility is logarithm based, which is undefined for negative values (it is defined for 0 as the uility is $\log(1+x)$).

6.2 Bot

The bot receives an action from the environment containing the player's hand, dealer's hand, recently revealed cards and the current option the bot needs to make. First it updates its internal memory of the cards left, based on the revealed cards. If the option is to choose a bet size, it passes all its information to the contained bettor object responsible for determining an appropriate bet size, which will be sent back as an action. If the option is anything else, all the information is sent to the contained strategist object that tries to find the best move based on long-term gains. After performing an action, the bot logs the action-observation pair to later allow evaluation based on these logs.

7 Reward distribution

The aim of this section is to recursively compute the distribution of rewards for a given game when following an optimal strategy. This distribution will be used for implementing both a strategist capable of taking optimal steps and a bettor producing optimal bet sizes.

Winning or losing a game yields ± 1 times the original bet size, while obtaining a blackjack pays $3:2$. Doubling down doubles the reward after a game, and splitting allows

the player to play two hands at the same time. Under the assumption that the player never splits their hand more than once[1], thus playing at most 2 hands at the same time, their possible rewards include values from -4 to $+4$ in 0.5 increments, which are 17 different values. For this reason reward distributions are discrete and can be stored in an array, by storing the probability of each reward occurring. A utility function is assigned to reward distributions, which calculated the utility of each possible payout, and calculates a utility for a reward distribution by taking the dot product of the reward utility and the reward distribution vectors.

Let $R_u(s,a)$ denote the reward distribution with utility function u, where s is the current state, including the discarded cards and the hand of the dealer and the player, and a is the last action taken by the agent which can be either betting, splitting, doubling down, hitting or standing. As the last action determines the actions available in the next time step, $R_u(s,a)$ can be calculated recursively using the following formula:

$$R_u(s,a) = max_{a' \in A_a} \sum_{c \in C} p_{(s,c)} R_u(s_{(c,a')}, a') \tag{1}$$

Where A_a are the possible actions after taking a as the last action, C are the set of cards, $p_{(s,c)}$ is the probability of drawing card c given state s, $s_{(c,a')}$ is the state after drawing card c after performing action a' and the summation is maximized regarding the reward distribution's utility value.

[1]Splitting more than once is rare, while it makes the computation significantly harder.

As the hand total of either the player or the dealer increases by at least one every time the player hits or stands, and the player can perform the other actions no more than once, the recursion is finitely deep and $R_u(s,a)$ can be computed using recursion in a finite time. At the terminal nodes of the computation, the winner is determined by the rules of the game, assigning a constant $+1$ payout for winning leaves, -1 for losing leaves or 0 in the case of a draw.

Equation 1 provides a general formula for hitting and standing, but splitting or doubling down modifies the payout and therefore the reward distribution needs to be modified accordingly.

$$R_u(s,split) = max_{a' \in A_{split}} \sum_{c \in C} p_{(s,c)} R_u(s_{(c,a')}, a') + \sum_{c \in C} p_{(s,c)} R_u(s_{(c,a')}, a') \tag{2}$$

$$R_u(s,double) = max_{a' \in A_{double}} 2 \sum_{c \in C} p_{(s,c)} R_u(s_{(c,a')}, a') \tag{3}$$

Equation 2 and 3 are relatively similar, but there $2R$ and $R+R$ differs, as $2R$ denotes, sampling from R and doubling the reward, similarly to how doubling down works, while $R+R$ denotes sampling R twice and adding the results together, similar to how splitting hands works.

Every $R_u(s,a)$ will be stored on the heap and their pointers will be stored in a lookup table to avoid recomputing the same reward distributions. As the recursion is deep, the

performance of an interpreted language is insufficient to compute some $R_u(s,a)$ values and therefore the code is written using Cython and the lookup table is implemented using the unordered_map data structure.

8 Strategists

The strategist part of the bettor is responsible for deciding between the actions splitting, doubling down, hitting and standing. The strategist interface exposes three methods to allow the bot to determine the appropriate actions to perform, these methods are:

- should_split
- should_double
- should_hit

As described in Section 6, the environment has four different stages, three of which directly correspond to these methods, to allow the bot to delegate tasks to the strategist. Although the strategist is arguably the more crucial part of the bot, as it determines whether the player wins or loses a bet, most of the literature focuses on the bettor. This is due to the fact that the player is presented with 3 information, the hand of the player and the dealer and the recently discarded cards. When the information about the discarded cards is disregarded, the number of possible states is limited. The player's hand has a total of at

most 21, which can be either soft or hard, while the dealer can have 10 differently valued cards. Each of these combinations will have an action which is optimal when any other information is disregarded and therefore a lookup table can be used by the player to choose appropriate actions. The idea using such a lookup table is called the basic strategy, which will be further investigated in Section 8.1. The other option is to include the information about the discarded cards, which is hard for humans to keep track of. Slight changes to the basic strategy based on discarded cards, would only result in slight changes in the payout, as most of the games the taken actions would match the ones proposed by the basic strategy. However, significant changes to the basic strategy would be hard for humans to keep track of, thus the literature on how humans should play blackjack is limited to the basic strategy.

Computers on the other hand are great at memorizing information such as discarded cards and therefore using reward distributions introduced in Section 7 allows computers to use every available information to choose the action that promises the highest utility.

8.1 Basic Strategist

The basic strategist is based on the idea of a basic strategy. The implementation of the strategist is straightforward, it just requires a lookup table and each method to use the appropriate lookup table with the appropriate values. Most casinos use a different rule variation of blackjack, in order to finely tune their advantage over the gamblers, for example some casinos allow players to double down after splitting their hands, while some do not.

These rule variations not only modify the advantage of the dealer but also change the basic strategy. Although many basic strategies are available online, these often times differ, and for this reason the project implements its basic strategists by generating its own lookup table. This is done by using the optimal strategist, used with a linear utility function, to calculate the optimal action for each of the available states, assuming that no cards have been discarded. The basic strategists make this assumption as when disregarding the information about discarded cards, the cards have a uniform chance of being drawn, just as if there were no cards discarded yet.

8.2 Optimal Strategist

The optimal strategist uses the reward distribution introduced in Section 7, with a logarithmic utility function. As the bot has a utility function of $\log(1+x)$, where x is the bankroll, the utility function used by the optimal strategist is $\log(1+x+pb)$, where p is the payout for the game and b is the bet size. As the strategist takes action after placing bets, the bet size is available for the definition of the utility function. The reward distribution can be used to determine which action to take, by evaluating the reward distribution when taking either of the available actions and taking the action with the reward distribution that has the highest utility. As reward distributions are stored on the heap using Cython, native Python code can not access these distributions, thus the optimal strategist is written in Cython.

9 Bettors

Bettor classes are responsible for determining the bet sizes to make before each game. Most bettors only implement a single method called get_bet_size, which takes two arguments, the bankroll of the bot and the distribution of the remaining cards, and returns a preferred bet size. Machine learning based bettors additionally implement two methods to update their internal neural network and to save their weight. When players disregard information about discarded cards, the only information available to them is the size of their bankroll. However, the outcome of the next game does not depend on this information, so there are no betting strategies using only this information. Casinos set their rules to give advantage to the dealer, so unless the player has any additional information, they should bet as little as possible. This is captured by the constant bettor. Contrary to the strategist not having any simple algorithms that are based on the discarded cards, the bettor can use card counting to change their bet sizes based on the discarded cards. The idea behind card counting is that some cards are more advantageous for the dealer, while others give advantage to the player. By counting how many of the advantageous and disadvantageous cards are discarded, the player can approximate the edge the dealer has over them. When the player's advantage increases, the bettor increases the size of the bets.

9.1 Constant Bettor

This is the most straightforward way to bet in blackjack. The bettor disregards any information and places the minimum available amount as a bet. Although, this is a simple agent to implement it provides a significant baseline for betting strategies. Every blackjack game can be thought of as a random variable X, that is conditional on the discarded cards. When disregarding this information, every game is the same random variable X. For this reason betting b_i at time step i has an expected value of $\mathbb{E}(\sum_i b_i X) = \sum_i b_i \mathbb{E}(X)$. As casinos set their rules so they are advantageous $\mathbb{E}(X) < 0$ and therefore $\mathbb{E}(\sum_i b_i X) = \sum_i b_i \mathbb{E}(X)$ is maximized when $\sum_i b_i$ is minimal, that is when $b_i = 1$ for every i. It follows that when disregarding information about the distribution of the deck, the best available bettor is the constant bettor. For this reason, any bettor that performs better than the constant bettor needs to use follows information about the distribution of cards that have been discarded. This baseline can be used to show that a machine learning bettor is learning.

9.2 Vector Bettor

The most famous card counting strategy is called the Hi-Lo card counting system [Tamburin, 2017], where the bettor keeps track of the proportion of advantageous to disadvantageous cards in the remaining deck. When more low cards have been played than high cards, the probability of the player getting blackjack increases, which pays $3:2$. Even if the player had the same chances to win a given game, because more of these wins will be due to blackjack,

the player gains some advantage. Additionally, the dealer has no choice whether to hit a card when they have a hand total less than 17, so high cards are disadvantageous to them as these are likely to bust them.

Humans performs the Hi-Lo card counting system by starting with a running count of zero, subtracting 1 from it every time it sees an Ace, 10 or face card and adding 1 every time it sees a card between 2 and 6. Dividing the running card by the number of remaining decks, which can be approximated by looking at the cards in the shoe, results in the true count. The bettor uses the true count to calculate an appropriate bet size given by $TC - 1$, where TC is the true count.

The implemented vector bettor is initialized with a vector, that maps cards to values based on how advantageous they are. The running count is given by the dot product of the vector it was initialized with and the distribution of the remaining cards. As the number of cards left in the is known, the true count is computed by dividing with the exact number of decks left in the shoe.

The implementation of the vector bettor allows for any vector to be used, but during evaluation the bettor was always initialized using the number from the Hi-Lo card counting strategy.

9.3 Kelly Bettor

[Thorp, 2008] proposes to use the Kelly criterion to produce bet sizes in blackjack, by calculating the probability of each possible payout, then use a computer to calculate the bet size that optimizes for the Kelly criterion, which is equivalent to the maximum expected log. The probability of each possible payout can be calculated by taking the weighted sum of the reward distribution of every possible game initialization weighted by the probability of the game being initialized in a given way. Reward distribution requires a utility function for which using the logarithm function would require to know the size of the bet, which is currently being determined. For this reason, the reward distribution uses the linear function as a utility, since the actions taken and thus the reward will be relatively similar.

After obtaining the probability of each possible payout, the utility function is given by

$$\mathbb{E}(log(1+x+r*b)) = \sum_r (p_r \log(1+x+r*b))$$

where x is the bankroll of the agent, b is the bet size, r is the reward for the game and p_r is the probability of being rewarded r after the game. The objective is to maximize $\mathbb{E}(log(1+x+r*b))$, by finding the optimal b, which can be found by differentiating the function with respect to b.

$$\frac{\partial}{\partial b} \sum_r (p_r \log(1+x+r*b)) = \sum_r \frac{p_r * r}{1+x+r*b} \tag{4}$$

There is no explicit formula to finding the roots of equation 4, but computers can find these approximate these roots. The bettor uses SymPy [Meurer et al., 2017], in order to find these roots, which are evaluated one at a time to find the bet size b that optimizes the maximum expected log. Although SymPy successfully finds the roots of the function, calculating bet sizes using the Kelly Bettor is significantly slower than other methods. Obtaining 10000 bet sizes takes over 24 hours when using the Kelly bettor, while obtaining 10000 bet sizes using other bettors are nearly instantaneous.

9.4 ML Bettor

The purpose of the investigation is to implement a bettor capable of producing bet sizes based after learning from past experiences. Section 10 introduces how the reinforcement learning agent is implemented, trained and tuned, while this section explains the general architecture of a machine learning based bettor. A bettor called TrainerBettor is implemented that initializes using a reinforcement learning agent and interacts with this agent. As the environment anticipate actions other than bet sizes, the reinforcement learning agent can not interact directly with the environment, instead it interacts with the TrainerBettor. Whenever the bot calls the get_bet_size method of the TrainerBettor, the bettor creates an observation based on the parameters and passes it to its agent, then transforms the action provided by the agent to a bet size which gets returned to the bot. After the bot finishes the game using the strategist, it calls the set_payout method of the TrainerBettor to return the payout for the

game. TrainerBettor updates the agent using the reward $\log(1+x+p)-\log(1+x)$, where x is the bankroll before the game and p is the payout for the game, in order to optimize for maximum expected log. When the training process finishes, the bot calls the save method of TrainerBettor in order to save the policy network used by the trainer.

The saved policy networks are used using the PolicyBettor, which is initialized using a policy network that it uses in order to produce bet sizes, without updating the internal weights.

10 Machine Learning

For some policy π [TensorFlow, 2021] defines the Q-function, $Q^{\pi}(s,a)$, as the expected value of the discounted sum of rewards obtained by performing action a at state s and following the polict π afterwards. The aim of Q-learning is to approximate the optimal Q-function $Q^{*}(s,a) = argmax_{\pi}Q^{\pi}(s,a)$.

10.1 Deep Q-learning

Deep Q-learning [Mnih et al., 2013] uses a neural network to approximate a Q-function. The network takes a state s as an input and each of the $|A|$ many output nodes correspond to $Q(s,a)$ for a different action $a \in A$. Due to the architecture of Deep Q-learning, the action space of the agent needs to be discrete.

Using a tuple, DQNBettor maps the integer-valued action provided by the agent to possible bet sizes. The values that the DQNBettor can bet consists of $(1.0, 1.5, 2.0, 3.0, 5.0, 10.0, 15.0, 20.0, 30.0)$, which numbers were determined by examining the bet sizes produced by other bettors.

10.2 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient [Lillicrap et al., 2019] is an actor-critic algorithm that supports a continuous action space. In this algorithm, critic is a neural network that takes both a state s and an action a in order to estimate the Q-function $Q^{\pi}(s,a)$ when following the current policy π. The policy network, also known as the actor, is a neural network that takes a state s to produce an action a. The critic and policy networks are updated at the same time, where the critic network trains to improve the estimate for the Q-function $Q^{\pi}(s,a)$ with regard to the policy π given by the policy network, while the policy networks trains in order to maximize the critic's Q-function estimates.

The DDPGBettor implements a DDPG agent, with both policy and critic networks being a multilayer perceptron. The policy network uses the $tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ activation function to provide values in the range (-1, 1). DDPGBettor using the function $max(1.0, (x + 1.0) * 29.5 + 0.9)$ scales the outputs to the range (1, 59.9) to produce bet sizes. Another option would be to use the output between (-1, 1) to determine what proportion of the bankroll to bet. However, the static range is most of the times smaller and therefore should be easier

to learn than a dynamic range.

10.3 Tuning process

As described in section 4.3, the project uses BayesOpt provided by Tune, in order to tune the hyperparameters of different agents. As Bayesian optimization does not scale well nor do the departmental computers support tuning that uses multiple machines, the tuning will take process on a single machine. To slightly scale this system, the number of layers in the networks will not be treated as hyperparameters during tuning, but rather passed to the tuner as constants. As the number of hidden layers are constants, multiple tuners can be run in parallel each optimizing using a different number of hidden layers. During tuning, the bot will use the BasicStrategist, as it is significantly faster, while achieving similar payouts.

The DQN based bettor has been tuned with the following search space, with the number of layers ranging from 1 to 3:

- learning_rate: (1e-15, 1e-5)
- layer_i: [1, 2, 4, 8, 16, 32, 64, 128, 256]
- epsilon: (0, 0.15)
- batch_size: [8, 16, 32, 64, 128, 256, 512]

The DDPG based bettor has been tuned with the following search space, with the

number of layers ranging from 1 to 3:

- policy_learning_rate: (1e-20, 1e-3)
- critic_learning_rate: (1e-20, 1e-3)
- policy_layer_i: [1, 2, 4, 8, 16, 32, 64, 128, 256]
- critic_layer_i: [1, 2, 4, 8, 16, 32, 64, 128, 256]
- sigma: (0, 0.5)
- batch_size: [8, 16, 32, 64, 128, 256, 512]

Additionally, each configuration space was run in three different copies, one training the agent for 30 minutes, one training the agent for an hour and one training the agent for two hours.

After tuning for two days, the best hyperparameters for the DQN agent were had following values:

- learning_rate: 1.4100798723572584e-05,
- network_shape: [32, 16, 2],
- epsilon: 0.022540060104017743
- batch_size: 256

After tuning for two days, the best hyperparameters for the DDPG agent were had following values:

- policy_learning_rate: 2.4800149377144643e-17
- critic_learning_rate: 5.664867179427271e-13
- policy_network_shape: [64]
- critic_network_shape: [4, 8]
- sigma: 0.2571172192068058
- batch_size: 16

10.4 Training process

Using SLURM, multiple machine learning based bettors were trained at the same time, using the desktop-batch partition provided by the department. After training the bettors for two hours with the hyperparameters defined in section 10.3 each of them were evaluated to obtain the policy which performs best in the long run. The best performing policies have been moved to bbwrl/bot/bettors/ml/default_networks.

11 Analysis

11.1 Methods

Different bots will be evaluated by simulating interactions between the given bot and the environment, while logging every step taken by the bot. These logs will be used to determine how the bankroll available to the bot changed over time and how much it chose to bet at any given time. As the objective of the bot is to optimize $\mathbb{E}[\log(1+x)]$, where x denotes the bankroll, the bot is evaluated by the approximation $\mathbb{E}[\log(1+r)] \approx \log(1+\mu) - \frac{\sigma^2}{2(1+\mu)^2}$ described in section 3.1. For a given realization of data, the sample mean and sample variance can be calculated for r, which values will be used to calculate the approximation as a metric.

Running simulations require two parameters, the number of games after which the game is terminated automatically and the starting bankroll available to the bot. The game is automatically terminated after 10.000 games. This number was chosen as the Kelly bettor needs more than a day to simulate 10.000 games, while the clusters allow at most 48 hours of uninterrupted running time. Determining the starting bankroll of the bot was based on the performance of different bots. The higher the starting bankroll, the easier it is for a bot to avoid bankruptcy and therefore the better it performs. When the bankroll is set as 300 times the minimal betting size, even the KellyBettor combined with the OptimalStrategist, scores a negative metric -1.193542315501154e-07. When the bankroll is set at 800 times

the minimal betting size, even the VectorBettor combined with the BasicStrategist scores well. The following table shows the metric scored by the VectorBettor when combined with the BasicStrategist:

bankroll	metric
100	-0.0002173890505687638
200	-9.341839849138445e-05
300	-3.545722616651242e-05
400	-1.453925862725105e-05
500	-2.9839053514622953e-06
600	-3.9616658721630907e-07
700	1.2417035476220497e-06
800	2.1410844862880517e-06
900	2.6703882718163795e-06
1000	2.5562206751873035e-06

Table 1: Performance of bot based on strating bankroll.

The bot scores similarly when starting with a bankroll of 800, 900 or 1000. This is because when the bot starts with a bankroll of 800 it only infrequently bankrupts, in which case the performance is similar as it provides the same bet sizes in all 3 cases.

As one of the simplest bettor is profitable when the starting bankroll is 800, but even the optimal bot loses in the long run when the starting bankroll is 300, the bots will be

evaluated with a starting bankroll of 600.

As simulating these bots take a significant amount of time, it is important to utilize the number of computers available on the desktop-batch partition. The Kelly based bots will run one episode per core and will be run on 50 nodes to obtain 300 simulations. As the other bots are significantly faster, these simulate ten episodes per core and will be run on 15 nodes to obtain 900 simulations. As the simulations are large files, they are not provided with the project.

11.2 Results

Evaluating the simulations described in the previous section resulted in the following scores

	BasicStrategist	OptimalStrategist
ConstantBettor	-9.14176058263226e-06	-2.4681096249020366e-06
VectorBettor	-3.9616658721630907e-07	1.6155563911756714e-05
KellyBettor	7.48523145534362e-06	3.1125585328481645e-05
DQNBettor	-9.015988479329113e-06	-3.4521271904230075e-06
DDPGBettor	-9.971453194454193e-06	-3.996055182445184e-06

Table 2: The approximated maximum log utility performed by each bot.

The machine learning based bettors perform similarly to the ConstantBettor, which implies that the agent was capable of determining that the game has a negative expected

value and therefore it should bet as little as possible. The machine learning agents failed to recognise that some card distributions are advantageous to the player. This could be resulted, by the problem being too complex for such an agent or alternatively, due to unsuitable tuning or training. As explained in section 3.2 VectorBettor is based on a single layer neural network and therefore DQN and DDPG should both be capable of learning a strategy at least as rewarding as the Hi-Lo strategy.

An interesting observation that can be made by comparing the bots is how similarly the simple Hi-Lo system (VectorBettor) used by card counters compares to the optimal bettor (KellyBettor). The combination of the Hi-Lo system with the optimal strategist performs slightly better than the combination of the optimal bettor with the basic strategy. This observation is unexpected due to the lack of literature about strategies that are better than the basic strategy, but simple enough for humans to apply.

For every available bot, two plots has been generated to show how their bankroll change over time and how much they bet at given a time t, located in Appendix A.

12 Reflection

12.1 Contributions

The project provides a framework to implement, tune, train and evaluate blackjack playing bots. Although the scripts provided with the project that enable batch compute are specific to SLURM, the object-oriented design of the rest of the code is highly adaptable, enabling other to research new bettors and strategists and compare against various baselines.

12.2 Shortcomings

The project required a shallow research in a wide area of topics, while deeper research in a smaller field would have been more appropriate for a third-year project. The other crucial observation is that the majority of the time was spent on setting up an environment that is a perfect replica of blackjack, which made implementing optimal bettors and strategists more time-consuming than necessary. As the machine learning agent did not learn to implement the optimal bettor as a baseline was unnecessary. The order of implementing the optimal bettor first was motivated by the ease of evaluating a bettor during training, but this was not the right choice. Implementing a simplified version of blackjack could have allowed to further develop the machine learning aspect of the project and potentially present an agent that out preforms some baseline.

12.3 Future extensions

As the project was successful at implementing a blackjack framework, but failed to train well performing agents, the project can be extended by machine learning based blackjack bettors. This could involve the use of the same agents with better tuning and training techniques or with the use of more advanced reinforcement learning agents.

As explained in chapter 11.2, the investigation found out that the strategist is just as important when playing blackjack as the bettor component, which suggests further investigation into the development of strategists that are more advanced than the basic strategist, yet it can be employed by humans.

12.4 Evaluation

Although the project has failed to produce a reinforcement learning based bettor that out performs the Hi-Lo card counting system, it was well managed and implemented every component that was set out in the specification.

References

[Behnel et al., 2011] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineer-*

ing, 13(2):31–39.

[Breiman, 1961] Breiman, L. (1961). *Fourth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, chapter Optimal gambling systems for favorable games, pages 65–78. University of California Press.

[Cassirer et al., 2021] Cassirer, A., Barth-Maron, G., Brevdo, E., Ramos, S., Boyd, T., Sottiaux, T., and Kroiss, M. (2021). Reverb: A framework for experience replay.

[Dickson, 2019] Dickson, H. (2019). Blackjack with reinforcement learning. https://www.kaggle.com/hamishdickson/blackjack-with-reinforcement-learning.

[Estrada, 2010] Estrada, J. (2010). Geometric mean maximization: An overlooked portfolio approach? *Journal of Investing*, 19(4):134–147,6.

[Henighan, 2019] Henighan, T. (2019). Blackjack reinforcement learning. https://github.com/henighan/blackjack-rl.

[Hoffman et al., 2020] Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., Abdolmaleki, A., Cassirer, A., Yang, F., Baumli, K., Henderson, S., Novikov, A., Colmenarejo, S. G., Cabi, S., Gulcehre, C., Paine, T. L., Cowie, A., Wang, Z., Piot, B., and de Freitas, N. (2020). Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*.

[Hutter et al., 2019] Hutter, F., Kotthoff, L., and Vanschoren, J. (2019). *Automated Machine Learning: Methods, Systems, Challenges*. Springer International Publishing.

[Jiang et al., 2017] Jiang, Z., Xu, D., and Liang, J. (2017). A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem. *arXiv e-prints*, page arXiv:1706.10059.

[Krekel et al., 2004] Krekel, H., Oliveira, B., Pfannschmidt, R., Bruynooghe, F., Laugher, B., and Bruhin, F. (2004). pytest x.y.

[Liaw et al., 2018] Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. (2018). Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

[Lillicrap et al., 2019] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2019). Continuous control with deep reinforcement learning.

[Mao, 2019] Mao, C. (2019). Reinforcement Learning with Blackjack. Master's thesis, California State University, Northridge.

[Markowitz, 1959] Markowitz, H. M. (1959). *Portfolio Selection: Efficient Diversification of Investments*. Yale University Press.

[Meurer et al., 2017] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman,

R., and Scopatz, A. (2017). Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

[Moritz et al., 2017] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, W., Jordan, M. I., and Stoica, I. (2017). Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889.

[Muldal et al., 2019] Muldal, A., Doron, Y., Aslanides, J., Harley, T., Ward, T., and Liu, S. (2019). dm_env: A python interface for reinforcement learning environments.

[NVIDIA et al., 2020] NVIDIA, Vingelmann, P., and Fitzek, F. H. (2020). Cuda, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit.

[OpenAI, 2016] OpenAI (2016). Blackjack-v0. https://gym.openai.com/envs/Blackjack-v0/.

[Reynolds, 2021] Reynolds, T. (2021). Strip blackjack: how to play and where to play it in the uk. https://www.telegraph.co.uk/betting/casino-guides/blackjack/vegas-strip-blackjack/.

[Rossum and Drake, 2009] Rossum, G. V. and Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

[Sato, 2019] Sato, Y. (2019). Model-Free Reinforcement Learning for Financial Portfolios: A Brief Survey. *arXiv e-prints*, page arXiv:1904.04973.

[Shackleford, 2019] Shackleford, M. (2019). 4-deck to 8-deck blackjack strategy. https://wizardofodds.com/games/blackjack/strategy/4-decks/.

[Solai, 2020] Solai, A. (2020). Cracking blackjack — part 2. https://towardsdatascience.com/cracking-blackjack-part-2-75e32363e38.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

[Tamburin, 2017] Tamburin, H. (2017). How to count cards in blackjack. https://www.888casino.com/blog/card-counting-trainer.

[TensorFlow, 2021] TensorFlow (2021). Introduction to rl and deep q networks. https://www.tensorflow.org/agents/tutorials/$0_{i}ntro_{r}l$.

[Thorp, 2008] Thorp, E. (2008). The kelly criterion in blackjack, sports betting, and the stock market. *Handbook of Asset and Liability Management*, 1.

[Vidámi et al., 2020] Vidámi, M., Szilágyi, L., and Iclanzan, D. (2020). Real valued card counting strategies for the game of blackjack. In Yang, H., Pasupa, K., Leung, A. C.-S., Kwok, J. T., Chan, J. H., and King, I., editors, *Neural Information Processing*, pages 63–73, Cham. Springer International Publishing.

[Yoo et al., 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In Feitelson, D., Rudolph, L., and Schwiegelshohn, U., editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg. Springer Berlin Heidelberg.

A Plots

The following figures plots 20 simulations for each of the bots. The number above the plots is the metric achieved for that 20 simulations, the upper plot describes the how the bankroll changed over time, while the lower plot shows the produced bet sizes at given time t.

-5.844756819401681e-06

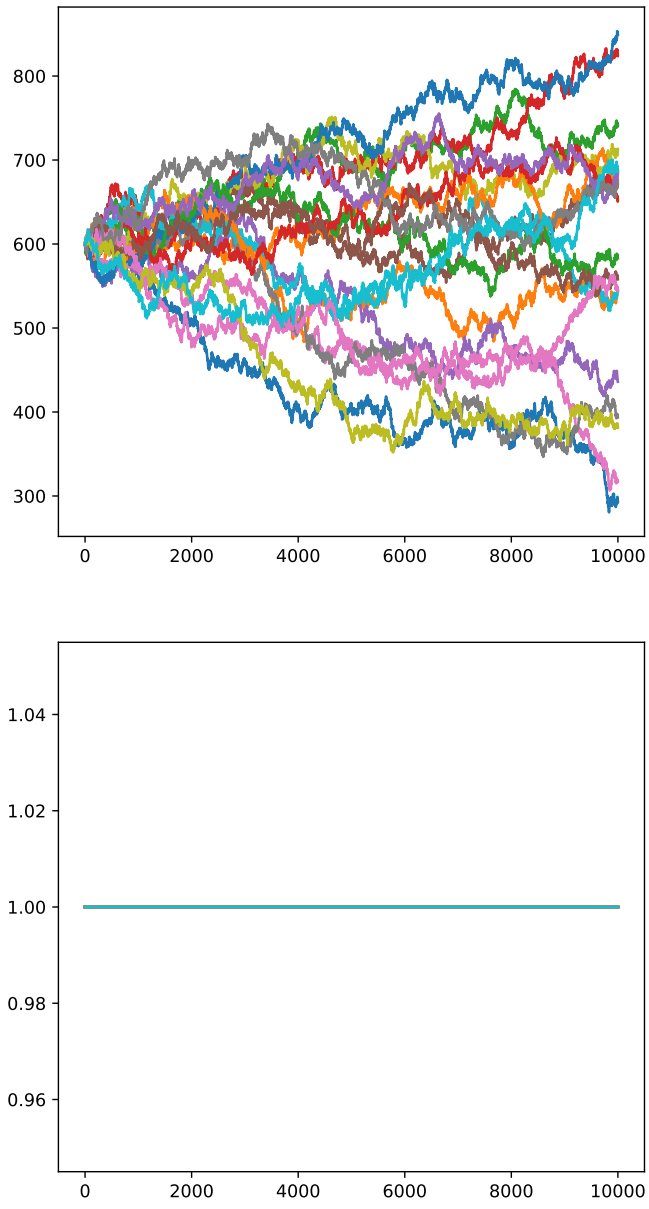


Figure 3: Constant bettor + Basic strategist

-2.4386254513963453e-06

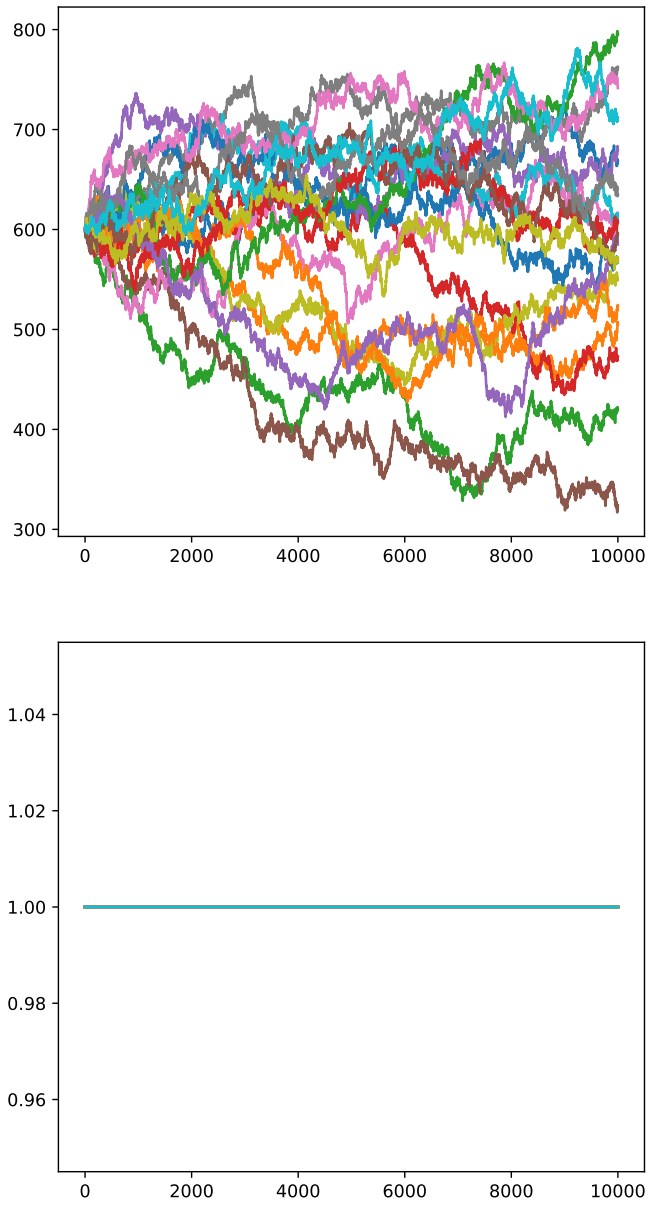


Figure 4: Constant bettor + Optimal strategist

-1.7192619906118063e-05

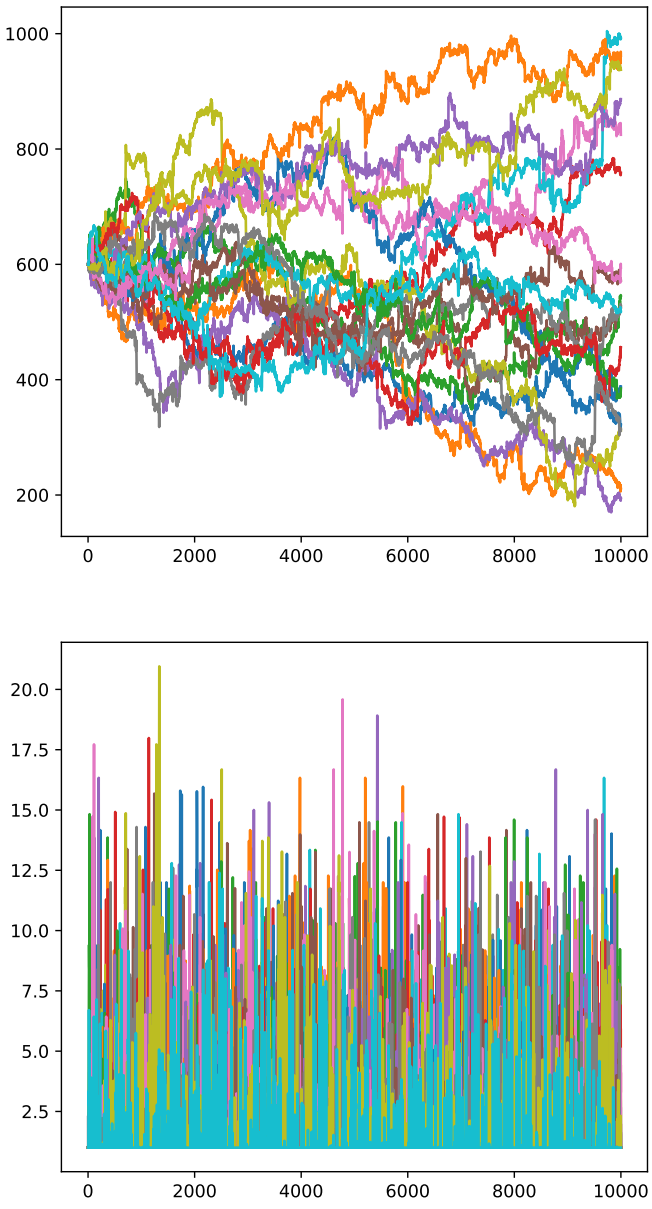


Figure 5: Vector bettor + Basic strategist

1.7202546447120594e-05

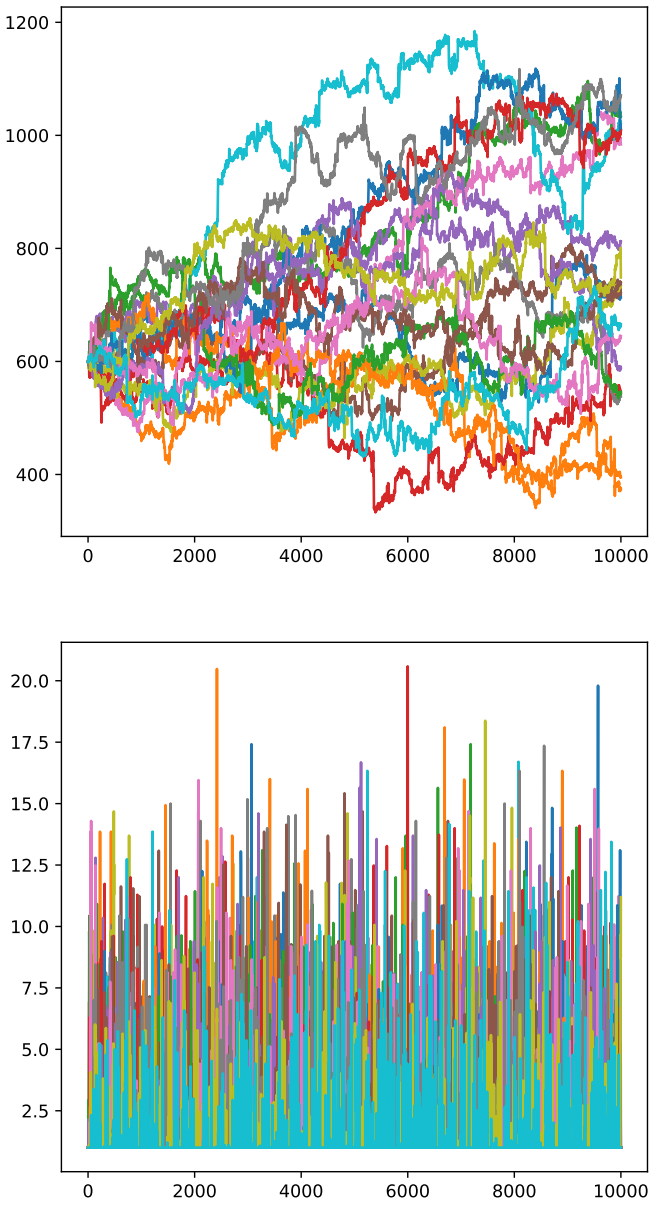


Figure 6: Vector bettor + Optimal strategist

4.6601989226527166e-05

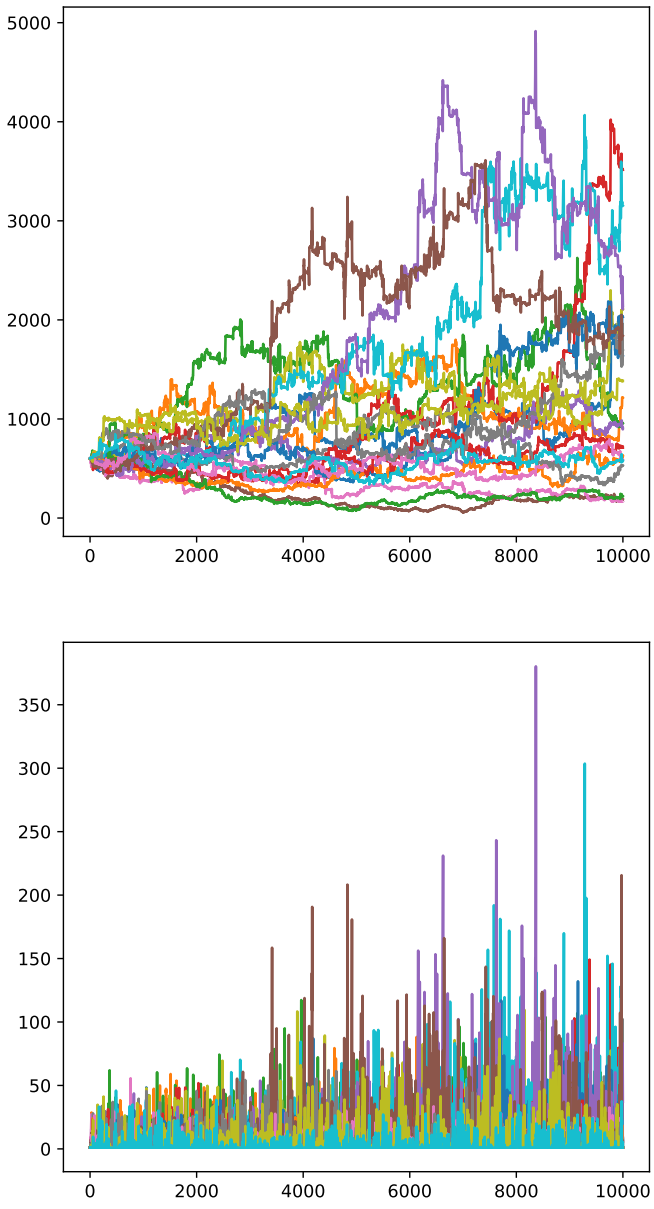


Figure 7: Kelly bettor + Basic strategist

-7.650226074266603e-06

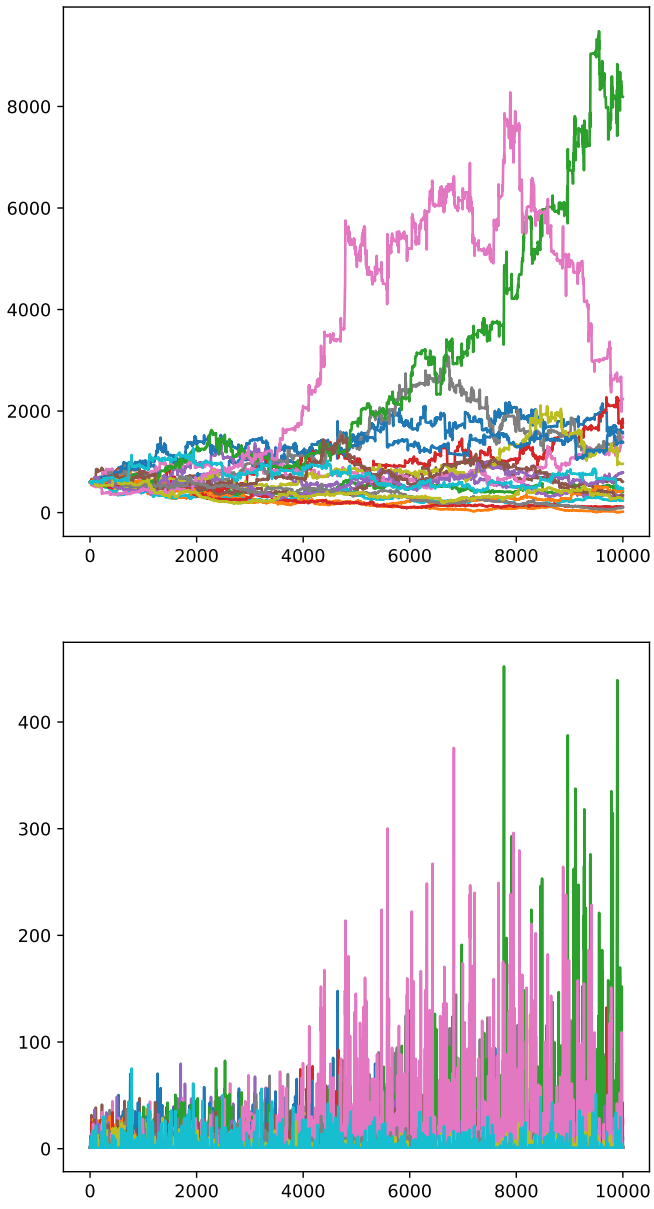


Figure 8: Kelly bettor + Optimal strategist

-8.544373904356471e-06

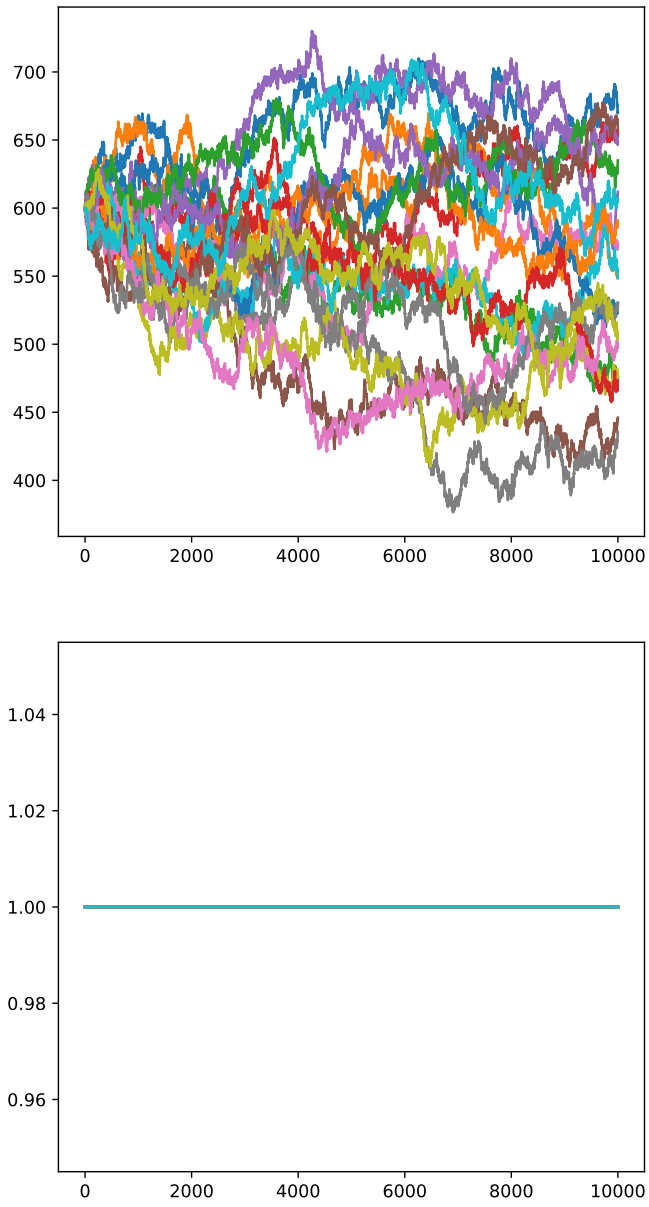


Figure 9: DQN bettor + Basic strategist

1.6183378727207954e-06

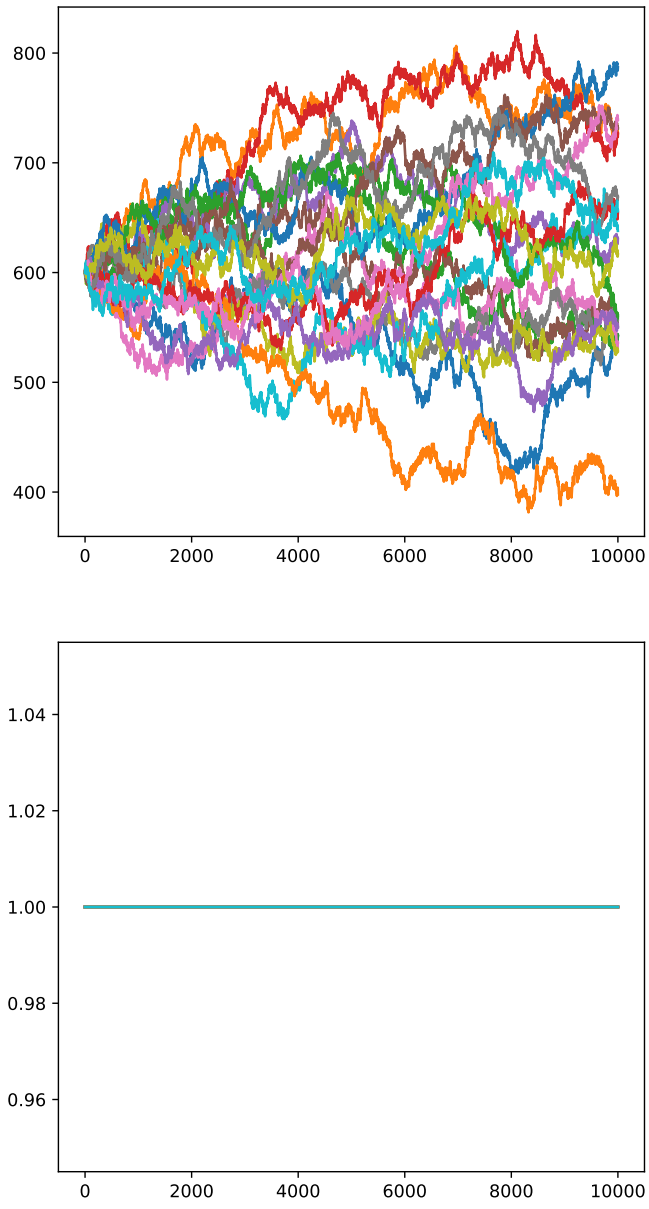


Figure 10: DQN bettor + Optimal strategist

-8.489606413819756e-06

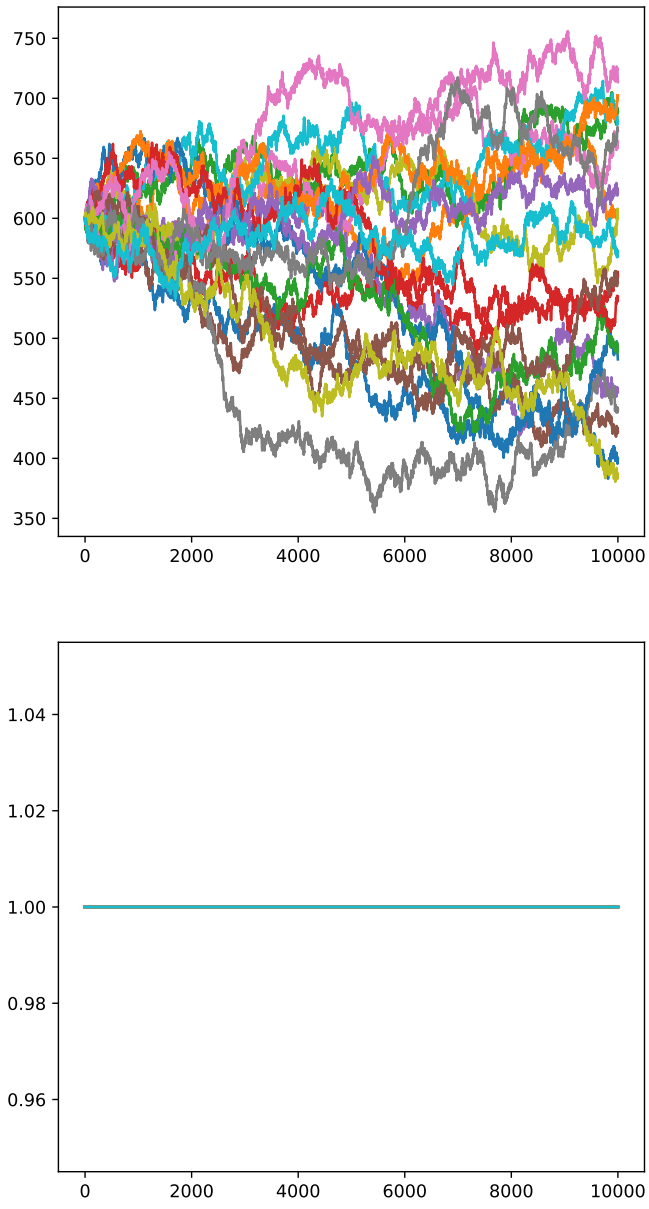


Figure 11: DDPG bettor + Basic strategist

-1.8240308658852639e-06

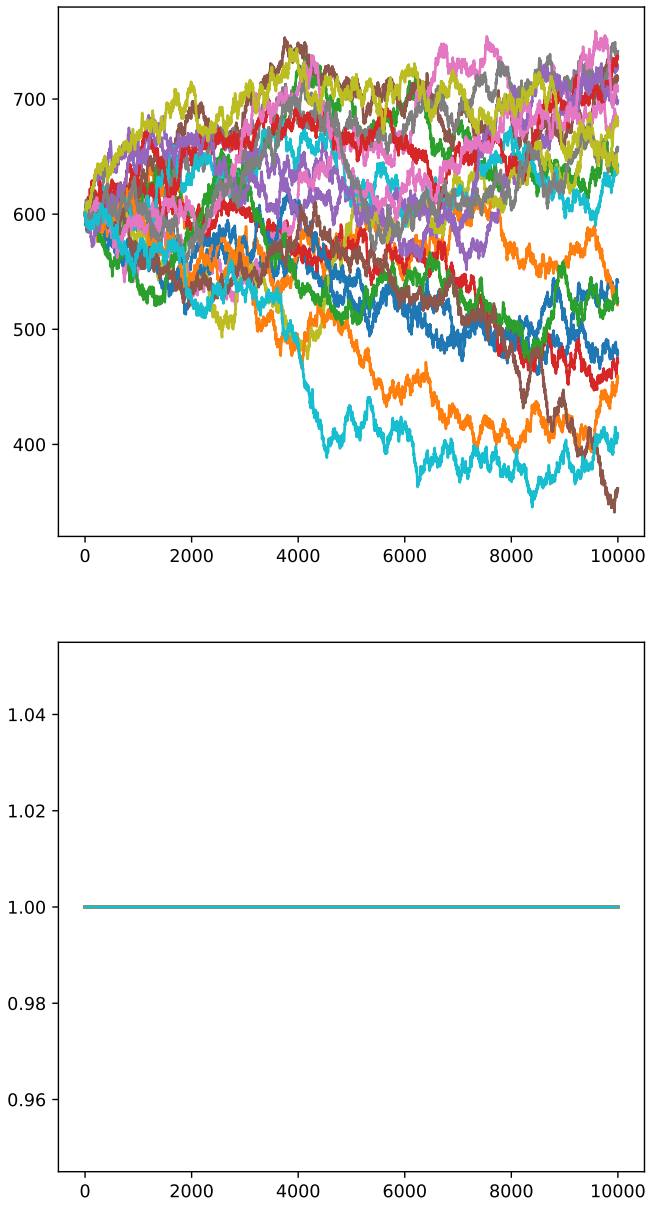


Figure 12: DDPG bettor + Optimal strategist

BEATING BLACKJACK WITH REINFORCEMENT LEARNING

PROGRESS REPORT

Patrik Gergely

November 30, 2020

ABSTRACT

This paper aims to contextualise the progress of an investigation into the improvement of existing blackjack strategies using reinforcement learning. In the casino game blackjack, a player competes against the dealer, with the objective of achieving a hand total closer to, but strictly less than, 21. Before the start of each game, the player places a bet which will either be doubled or lost, depending on the outcome of the game. This project proposes to play blackjack using two independent components: the bettor, deciding on appropriate bet sizes before games, and the strategist, which aims to win the games. An actor-critic algorithm will be used to produce the bettor, while the strategist will be implemented using dynamic programming. An explanation of the parallels between this bet sizing process in blackjack, and the portfolio management problem, creates the potential for the conclusions here drawn to be extrapolated to financial investment strategy.

1 Introduction

The overarching aim of the project is to produce and evaluate a blackjack-playing bot[1], consisting of a bettor that decides on bet sizes and a strategist that finds optimal actions according to the rules of blackjack. This aim will be achieved once the following conditions have been satisfied:

- Developed a strategist that can choose between splitting, doubling down, hitting, and standing.
- Implemented a bettor capable of determining appropriate bet sizes.
- Set up an environment with which the bot can interact.
- Evaluated the bot against existing strategies.
- Presented the findings in a final report.

Currently, mainstream algorithms applied in investment strategy revolve around data about historic market performance. This assumption that future market conditions will emulate (or be accurately predicted using) historic trends often falls short, and fails to take into account the current affairs that heavily influence market changes and future stock prices. In response to this failure, the algorithm proposed in this project aims to produce appropriate bet-sizes, based on the state of the environment (within the blackjack game), rather than data about historic win-rates. This bet-sizing mechanism resembles financial asset allocation strategy, and will potentially give rise to an alternative approach to financial risk and reward projections that account for holistic 'environmental' events. Accordingly, the emphasis throughout the project will be on the bettor, the component capable of betting appropriately sized portions of the bankroll after observing the environment.

2 Background

Three different approaches to bet-sizing in blackjack have been identified; outlined are the questions that arise from this existing body of literature, and the methods through which this project aims to address these.

[1]The term agent was purposefully avoided throughout the report to prevent any confusion when talking about agents in reinforcement learning.

Mao (2019) explores the performances of various reinforcement learning (RL) agents when playing blackjack, all of which fail to reliably beat the game. This is due to blackjack being too complex for any of these agents to master the game on their own. To combat this, RL will only be used to train the bettor, thereby restricting the complexity of the problem, making RL a potentially feasible solution.

Jiang et al. (2017) presents how RL can be used for the continuous reallocation of capital into several financial assets, with the aim of maximising return, while restraining risk. The paper produces a trading algorithm with the limitation of only accessing the history of each asset. This constraint, however, is not justified as the market highly depends on the state of the world, e.g. a pandemic having a significant influence on the stock market. The proposed bettor differs from the aforementioned algorithm as it observes the environment rather than the history.

As discussed by Thorp (2008) the Kelly criterion can be applied to blackjack to obtain optimal bet sizes when trying to maximize the bankroll in the long run. The Kelly criterion optimizes for the logarithmic utility, which can be argued to be a good choice, yet it is not the only valid option. On the contrary, RL generalizes to any utility function, allowing a more comprehensive way to balance risk and profit.

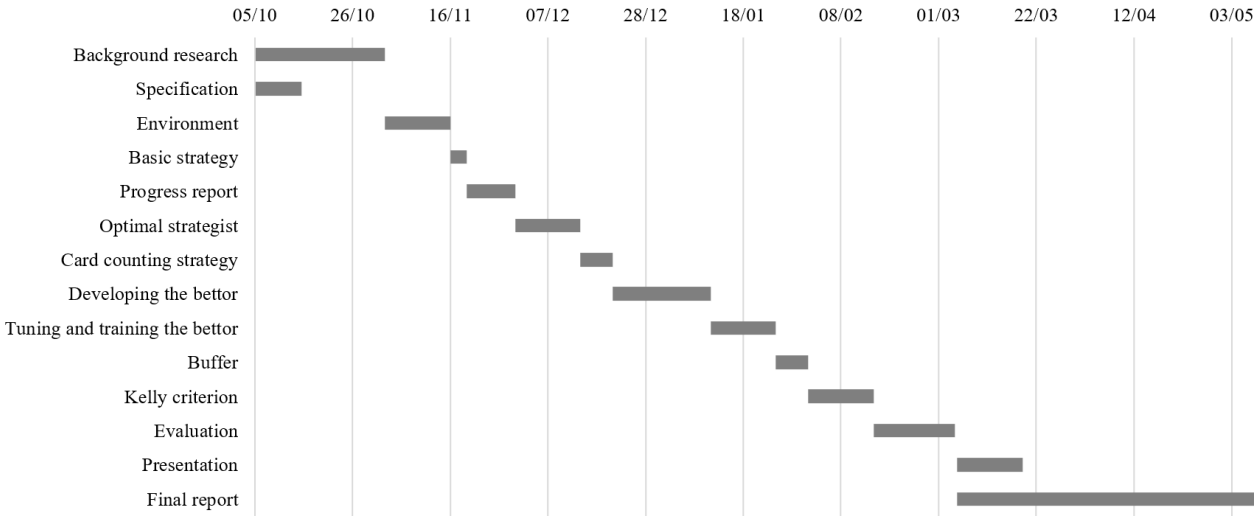
3 Overview

The final bot will consist of two modules: the bettor, trained using a RL algorithm and the strategist, implemented using dynamic programming. Considering the modular design, three interchangeable bettors will be developed: a popular card counting strategy, the Kelly criterion and one produced using RL. The expectation is for the RL based bettor to perform between the baseline card counting strategy and the optimal Kelly criterion. Two interchangeable strategists will be implemented: the basic strategy, which is a simple lookup table followable by anyone and the optimal strategist which will be discussed later.

Performance can be evaluated using different metrics based on the requirements. One possible metric definition for long-term return is fixing a number n and inspecting S_n, the bankroll after n trials. One other possibility is fixing a goal x and inspecting the number of trials needed for the bankroll to reach x. As maximising $\mathbb{E}\left[\log\left(S_n\right)\right]$, optimizes both previous metrics (Breiman, 1961), the bot will be rewarded based on the logarithmic utility: given a bankroll X and a Y payout, the reward is defined as $\log(X+Y)-\log(X)$. Bettors can only be evaluated when paired with strategists, as the reward is based on whether the bot wins in a given state, which is determined by the strategist. Consequently, all six proposed combinations will be evaluated to avoid developing bias for a strategist, thus better assessing the performance of bettors.

4 Project plan

The timetable includes the nine components of the project, the creation of four reports, and an additional week-long buffer period during the second term to address possible delays. At the time of writing background research, environment, and basic strategy have been implemented.



4.1 Background research

Although blackjack is a simple game, rules vary between casinos based on the desired edge over the player. After inspecting several rule variations, the project will be based on Vegas Strip[1] due to its popularity.

As discussed in section 2, according to Mao (2019) using only RL performs poorly, thus an accurate strategist is needed to handle the complexity of the game. The transition probability function can be precisely calculated based on the distribution of discarded cards and as hand totals grow after each action transitions between states are acyclic. These two properties ensure that dynamic programming can be applied to calculate the value of each state recursively.

To allow betting any real number between the minimum bet size and the bankroll, a RL algorithm supporting continuous action space is needed. As the strategist needs to calculate the value of numerous states, determining the optimal action is expected to be slow. To address these requirements, the bettor will be based on the sample efficient actor-critic with experience replay presented by Wang et al. (2017).

Part of the background research involved finding the appropriate resources matching the requirements of the project, which are described in detail in section 5.

4.2 Environment

In blackjack, the environment consists of a deck, where the dealer is drawing cards from, the bankroll of the agent, and the game which is currently being played. A game consists of a dealer and a number of players each represented by their hand total and whether they have a soft hand[2]. After a game is finished the only valid action is placing a bet, which starts a new game and every other time the action space is a subset of splitting, doubling down, hitting and standing. This is implemented using the dm_env interface to allow the use of Acme's EnvironmentLoop method for training agents.

4.3 Basic strategy

The basic strategy is a lookup table, mapping every possible state (consisting of the player's hand and the dealer's hand) to the optimal action in that situation. As the state-space is small it is feasible to compute every optimal action. Blackjack has numerous rule variations and each basic strategy depends on the rules, therefore it is not obvious to find best one for the current variation. However, the probabilities in the optimal strategist can be modified slightly to calculate the basic strategy. Currently, a basic strategy is implemented from the web[1], but it will later be calculated using code for the optimal strategist.

4.4 Optimal strategist

Let $q_*(s, a)$ be the optimal state-action value function for a state s and action a, and $v_*(s) = \max_{a \in \mathbb{A}} q_*(s, a)$ be the optimal value function. Let the state-space be a blackjack table and the action space consist of *hit* and *stand*, then $v_*(s) = \max(q_*(s, hit), q_*(s, stand))$. Given a card distribution, $q_*(s, stand)$ can directly be calculated, as the dealer follows a strict set of rules, while $q_*(s, hit) = \sum_c p_c v_*(s_c)$[3], where p_c is the chance of drawing a card c and s_c is the state after drawing card c.

The state-space forms a tree, as after each action the hand totals grow making it impossible to have a cycle. Given this, dynamic programming can be applied to calculate $v_*(s) = \max(\sum_c p_c v_*(s_c), q_*(s, stand))$. As at each step, the player's hand total grows by at least one and $v_*(s)$ is a constant when the player's hand total is over 21, therefore computing $v_*(s)$ recursively necessarily terminates.

As the bot uses the logarithmic utility, the strategy will calculate it's value function using the logarithmic utility as well, meaning if the bot is betting Y with X bankroll, the reward is either $\log(X - Y) - \log(X)$, $\log(X + Y) - \log(X)$ or 0.

The action space additionally includes doubling down and splitting. Doubling down consists of hitting exactly once and then standing, with the expected reward being the $\sum_c p_c q'_*(s_c, stand)$, where q'_* is the state-action value function when betting $2Y$ with X bankroll. Splitting is the option of splitting two identical cards into two individual hands. As the

[1] https://www.blackjackexpert.com/variations/vegas-strip-blackjack/

[2] In blackjack aces are worth either one or eleven. A hand is soft when the hand total has two possible values.

[3] This holds only under the assumption that the hitting gives no reward, which is not necessarily true. If the player busts there is a negative reward and the game is over. However, the problem is equivalent to making $v_*(s)$ take the penalty for losing whenever the player has a hand total over 21 in state s.

number of times, this can be done is limited and it can be performed only as the first action of the player, the expected reward from splitting can be calculated directly from the previous results.

4.5 Card counting strategy

Card counting is performed through the assignment of point scores to card values. After every dealt card, the corresponding point score is added to the running count, which is a value starting at 0, representing whether the player or the dealer is in favour. Low cards are assigned positive scores, while high cards are assigned negative scores, as this way a higher running count implies a higher concentration of high cards, which is favourable for the player. As the running count determines the player's advantage over the dealer it is used to determine the appropriate bet size. Vidámi et al. (2020) evaluates numerous card counting systems based on different point scores, multiple of which will be implemented.

4.6 Developing the bettor

This stage will start with additional research about potential RL algorithms. As discussed above, ACER (Wang et al., 2017) is a possible candidate for producing a bettor, due to its continuous action space and sample efficiency. However, there are various alternative RL algorithms, which could be considered if ACER does not fit the requirements of the problem. The neural network for the agent will be constructed through the python library Sonnet and the environment will consist of the card distribution corresponding to the deck and the bankroll of the agent, with a single continuous action, which is the proposed bet size. The agent will be rewarded based on the logarithmic utility: given a bankroll X and a Y payout, the reward is defined as $\log(X + Y) - \log(X)$.

4.7 Tuning and training the bettor

Bettors can only be trained with a strategist, as the reward is based on whether the bot wins in a given state, which is determined by the strategist. The model will first be trained and tuned using the basic strategy, as it is significantly faster when compared to its optimal counterpart. After finding a model shape which works well for the basic strategy, the hyperparameters will be tuned with respect to the optimal strategist.

4.8 Kelly criterion

According to Thorp (2008) the probability of each payoff type is sufficient to calculate the optimal bet sizes. The optimal strategist can be modified to calculate these probabilities using the same recursion. However, the optimal strategist requires the bet size, which the bettor is trying to find. Nevertheless, the behaviour of the optimal strategist should be similar for most bet sizes, giving an appropriate approximation for the needed probabilities when used with constant bet size.

4.9 Evaluation

Further research is needed in the evaluation of different bots. As investment is a well-studied field, a wide variety of metrics should be available capable of assessing the amount of risk a bot is taking and its expected reward.

5 Resources

The project will be written in Python as this is the best-supported language for machine learning. To manage the project MyPy will be used for static type checking pytest for testing purposes and Sphinx for creating documentation. Acme will be used for training the bettor through its EnvironmentLoop function while storing data using Reverb for experience replay. EnvironmentLoop requires an environment, which will implement the dm_env interface and an agent, which will be based on a neural network constructed with Sonnet. As training is computation heavy, it will take place on the Colaboratory service, providing free access to computing resources including GPUs. Using Git the code will regularly be backed up to GitHub to mitigate the risk of data loss. The final report will be written using Overleaf, a free online LaTeX editor.

MyPy	The MIT License
pytest	The MIT License
Sphinx	GNU General Public License version 2.0
Acme	Apache License 2.0
Reverb	Apache License 2.0
dm_env	Apache License 2.0
Sonnet	Apache License 2.0

As none of the software is going to be modified, usage under all of the aforementioned licences is allowed.

6 Methodology

The principles of Agile software development are used, as this suits best the project. Both the tasks and the time-frame are fixed, therefore constant progress needs to be made, which is easiest by sticking to goals that take one-two weeks to complete. As tasks are independent of one another, developing tasks during a sprint will not raise any issues, because they can be evolved later on.

When compared to the plan outlined in the specification, the project underwent some changes about the structure of the final bot. However, the Agile methodology allowed development to continue with minimal change to the timetable, as the content of the components has changed, but not the link between them.

7 Reflection

The original project specification did not include background research in the initial timetable with the intention of doing research at the beginning of each stage. Due to the nature of the project, this would have been possible, but still, it made more sense to spend four weeks in the beginning to develop a deeper understanding about the field and a more precise plan for the project. Due to the initial background research, the project fell behind the original timetable, but as this detailed plan removes the research aspect in many of the milestones, the delay should be combated by shorter development stages.

A week-long buffer was included in the original specification, which will be used to catch up with the schedule. As the buffer proved to be useful, the new timetable includes an other week-long buffer period during the second term to deal with any possible issues. The last stage is longer than the amount of time expected to complete it, as every delay will pile up to this point and during evaluation, it is expected to discover issues that all need to be addressed.

The developed code so far has been uploaded to `https://github.com/PatrikGergely/beating-blackjack-with-reinforcement-learning`, which follows the PEP 8 style guide for Python. To maintain code health unit tests have been implemented for every module and documentation according to the Google Python style guide.

References

Breiman, L. (1961). *Fourth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, chapter Optimal gambling systems for favorable games, pages 65–78. University of California Press.

Jiang, Z., Xu, D., and Liang, J. (2017). A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem. *arXiv e-prints*, page arXiv:1706.10059.

Mao, C. (2019). Reinforcement Learning with Blackjack. Master's thesis, California State University, Northridge.

Thorp, E. (2008). The kelly criterion in blackjack, sports betting, and the stock market. *Handbook of Asset and Liability Management*, 1.

Vidámi, M., Szilágyi, L., and Iclanzan, D. (2020). Real valued card counting strategies for the game of blackjack. In Yang, H., Pasupa, K., Leung, A. C.-S., Kwok, J. T., Chan, J. H., and King, I., editors, *Neural Information Processing*, pages 63–73, Cham. Springer International Publishing.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2017). Sample efficient actor-critic with experience replay.

BEATING BLACKJACK WITH REINFORCEMENT LEARNING

Patrik Gergely

October 15, 2020

ABSTRACT

Blackjack is a casino game, wherein a player competes against the house. The objective is to get a hand total closer to 21 than the dealer without going over 21 (*busting*). Portrayals in popular culture depict individuals with perfect memory, counting every discarded card in order to infer the distribution of incoming cards, thereby beating the casino. In reality, however, people are only able to perform simple card counting strategies like the Hi-Lo strategy [1]. These are too simple to gain significant edge over the house, giving the typical card counter an advantage of approximately 1-2% [2]. Computers are known for being better than humans at following complex algorithms and remembering data, which raises the question, could a computer program beat the Blackjack setup used in most Las Vegas casinos by *counting cards*?

1 Problem

It is a fairly common introductory exercise for reinforcement learning (RL) to train a Q-learning agent to play Blackjack using two inputs - the hand of the player, and the hand of the dealer - to produce a binary decision about when to *hit* and when to *stand*. However, most people stop at this point without actually solving the game, which would involve devising a strategy, assessing risks and placing bets. Could RL be used to train an agent which goes further, and plays all parts of the game - from making bets to standing at the right time?

Such an agent would consist of two main parts. The first part takes place during a game, when bet-sizing is no longer a concern, and resembles the mainstream Q-learning agent. At this point, the agent must simply decide whether to *hit* or *stand*, depending on expected payouts. This improves upon the mainstream agent by extending observations from the environment to include the state of the deck, and can be solved using Deep Q-learning [3]. The second part occurs in between games, when the agent has to evaluate the risk of playing based on the state of the deck in order to decide on a bet size. This second part poses two problems: firstly, it involves creating a distribution of the potential payouts for the given deck, and can be addressed using Monte Carlo experiments; secondly, it involves making appropriate bet sizes, using the payout distribution. This second problem has no prescribed algorithm, so additional research will be required to identify and test possible RL (or other deterministic) ideas. Subsequent evaluation of agents will require a strategy API, which can be consumed by a standalone evaluator program to generate statistics about their potential losses or wins.

2 Objectives

This project will attempt to design and implement a program capable of playing all parts of Blackjack, deciding

- whether to *hit* or *stand*,
- whether to *double down*,
- whether to *split*,
- whether to *surrender*,
- the size of the bet.

An additional program is needed to evaluate different strategies, which will simulate a casino and call upon the agent hundreds of thousands of times to adequately evaluate its performance over time.

2.1 Milestones

- **Evaluator:** Set up the evaluator first, to test progress during the entire project.
- **Basic strategy:** Evaluate Blackjack *basic strategies* [4] to both test the evaluator and set up a baseline.
- **Environment:** Gain understanding about the Python libraries used for the project.
- **Hi-Lo:** Train an agent using Deep Q-learning, using the same information provided to the Hi-Lo strategy, then compare the performance of this agent to that of the original Hi-Lo strategy.
- **Hyperthymesia:** Give all the available information to the agent, hoping for an improvement.
- **Table distribution:** Create a distribution of winnings corresponding to various different states of the Blackjack table.
- **Distribution scoring:** Train an agent which assigns high scores to (or in other words to prefer) distributions with high return and low risk, so as to optimize long-term return.
- **Bets:** Use the score corresponding to the empty table to decide on a bet size. After revealing the initial cards, use the score corresponding to the subsequent table to evaluate whether it is worth it to split, double down or surrender.
- **Evaluate strategy:** Combine all previous steps to formulate a strategy, and compare it to others using the evaluator.

2.2 Additional milestones

The project might develop faster than expected, in which case its scope would need to change. The following additional milestones are independent from the project, and so while finishing them is not a hard requirement, they would still function as extensions rather than separate tasks. These additional milestones are also set apart by the fact that they are unbound, which is advantageous because they could therefore be improved continuously over time.

- **UI to input cards:** Create a graphical user interface to allow users to input the card that is next drawn from the deck, to manually evaluate agents.
- **Bet-sizing with different strategies:** Originally, bet-sizing is only combined with the RL agent strategy, but it can be evaluated with the basic and the Hi-Lo strategy.
- **Model tuning:** As machine learning is being used, the hyperparameters and the model shape can always be further improved.

3 Methods

The principles of Agile software development will be used, as this suits best the project. As both the tasks and the time-frame are fixed, constant progress needs to be made, which is easiest by sticking to goals that take one-two weeks to complete. As each task is independent from the other, developing tasks during a sprint will not raise any issues, because they can be evolved later on. To assure that progress is being made at the right pace, a meeting with the supervisor will be set up after each milestone in order to assess the scope, based on the pace of the project.

4 Resources to be used

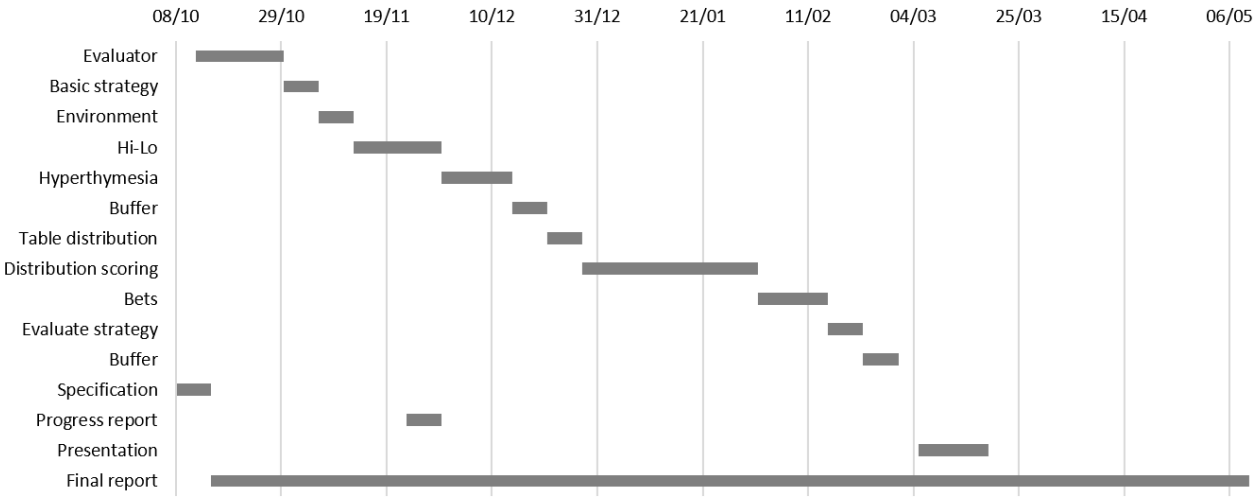
The program is going to be written in Python, with a research framework for reinforcement learning called Acme [5]. As machine learning requires intense computation power, Colab-provided TPUs will be used to train the model.

To avoid data loss, the codebase will be stored on GitHub and on a personal computer simultaneously.

5 Legal, social, ethical and professional issues

Not applicable for the project.

6 Timetable



References

[1] N. Richard Werthamer. *Risk and reward : the science of casino blackjack*. Springer, 2 edition, 2018. Page 17.

[2] N. Richard Werthamer. *Risk and reward : the science of casino blackjack*. Springer, 2 edition, 2018. Page 47.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[4] N. Richard Werthamer. *Risk and reward : the science of casino blackjack*. Springer, 2 edition, 2018. Page 9.

[5] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.