Actor-Critic Catan: Reinforcement Learning in High-Strategic Environments

Gabriël van der Kooij

November 2020

Abstract

Settlers of Catan is a complex board game that makes a great Reinforcement Learning research environment because of its high strategic nature and a good mixture of competition and cooperation. In this research, Settlers of Catan was digitized using the Unity game engine and Python. Utilizing the versatility of the Actor-Critic methods, Advantage-Actor-Critic a.k.a A2C was used to try tackle a simplified version of Catan by trying to play Catan on a human level, beating Random-Agents on the way. It was found that the A2C agent learns to spend its resources and to avoid no-ops / passing. This was revealed by decreasing loss functions and inspection of individual episodes. However, because of several biases, it does not perform well enough against even the Random-Agents that just perform random legal moves. This is mostly due to the agent not learning how to prioritize specific locations for buildings and the difficulties surrounding the sparse rewards in this game. Promoting curiosity and decentralizing Actors could be utilized in the future to improve results of an RL-powered agent in Settlers of Catan.

Contents

1 Introduction

The last few years, there has been a great increase of research involving well-known games being "solved" by AI's, such as chess, Go, certain types of Poker, Jeopardy and Starcraft, to name just the most popular. Surprisingly, not much such research has been done involving one of the most popular board games of all time: Settlers of Catan (called Catan from now on). Developed by Klaus Teuber in 1995, this game has been the number one board game in much of Europe over the last decade. It mixes competition and cooperation, involves a lot of strategy and is hard to master. Not unlike Starcraft, Catan has a large action- and state-space and can be viewed as a multi-agent problem. Furthermore, players of Catan have to deal with relatively sparse rewards and adjust their strategies accordingly. This all makes solving Catan quite a complex problem and an interesting environment to tackle using automated methods.

One of the more prominent AI techniques in these researches is Reinforcement Learning (RL). RL has proven to be powerful and versatile technique, to the point where it is used to fight the global pandemic caused by the COVID-19 virus by modeling the spread and helping to find an anti-viral cure. In this research, one of most promising Reinforcement Learning techniques, Advantage Actor-Critic a.k.a A2C, will be used to try to create an agent which can play Settlers of Catan at (least at) a human-like level. As an intermediate step, the A2C-agent will need to beat agents that take random (legal) moves, which are used to model low level human players that are new to the game.

1.1 Related work

1.1.1 Solving Catan

Games are and have been a inexhaustible source of challenges for Artificial Intelligence, being used often to benchmark older and newer machine learning techniques. A 2009 paper by István Szita, Guillaume Chaslot and Pieter Spronck [1] proposed a Monte-Carlo tree search algorithm (MCTS) for playing a custom simplified digitized Catan environment called 'SmartSettlers'. Even though SmartSettlers did not encompass all aspects of Catan, the were able to draw interesting conclusions surrounding game-play, such as the effects on players having the first, second, third and fourth turn. Section **7** will discuss our own findings on this matter. Moreover, they were able to create a strong Catan player that could consistently beat Random-Agents (agents that take random legal moves) and the built-in AI of the Java package where SmartSettlers was based on.

A few years prior, in 2004, Michael Pfeiffer [2] proposed a combination of RL and high-level heuristics to create an agent for Catan. These heuristics captured important aspects of the game (according to experienced players) in essentially a summary of the current environment state information, after which

the information was passed to the RL agent as state observations. The agent learned using self-play and was tested against human players a handful of times. Evidence was presented that using a learning approach performed consistently better than fixed strategy- or guided methods. Both these papers prove the potential of Catan as a research environment.

1.1.2 Reinforcement Learning in Games

A team from Google owned Deepmind [3] used RL more recently (2017) to create an agent that could play Starcraft II on a level comparable to professional human players. Even more impressive is the fact that this level of play was reached without using any heuristics. The technique used in this paper is comparable with the RL algorithm used to train the Deepmind Starcraft agent. There are key similarities between Starcraft and Catan, such as the high-strategic imperfect information nature of both games, the delayed rewards (and related long-term planning) and multi-agent interaction.

In another paper from 2017, OpenAI and the University of Berkeley [4] used Actor-Critic algorithms for a wide variety of multi-agent environments. These multi-agent environment where quite small in action and state-space, but explored cooperation, competition and mixtures of both in a key ways, proving the versatility of the Actor-Critic algorithm. In the paper a specific Actor-Critic algorithm was used where information about multiple policies[1] where combined to provide extra feedback to the agents, next to extrinsic (read: external) in-game rewards.

[1] The term 'policy/ies' is used during this paper to refer to the mapping of actions to states or, stated simply, the strategy of the agent.

2 Summary of Paper

Firstly, the 'Settlers of Catan' section (3) contains information about the board game itself and explains the fundamental rules. The 'Reinforcement Learning Algorithm' section (4) is about the setup. It provides insight on the used reinforcement learning framework in general and explains the used algorithm in specific. The 'Reinforcement Learning Environment' section (5) contains information about the conversion from board-game to a digital equivalent, including simplifications and their (shortened) justifications. It also discusses results found with this environment. Furthermore, this section contains general information about how input and output of the agent were handled. That carries over to the next section; 'Environment Overhaul' (6). This section contains information about how and why the aforementioned environment got a complete overhaul/remake to be more efficient. It also discusses simplifications made during this process and preliminary results. The 'Further Exploration' (7) section discusses the final results (and the needed and presents additional findings. The 'Discussion and Future Work' section (8) discusses the accuracy of the results and the method used. It also provides some proposals for future research. The 'Conclusion' section (9) summarizes it all in an overview of this research; where it has been valuable and where it has been lacking.

3 Settlers of Catan

Settlers of Catan is an easy board-game to learn, but a hard game to master. There are various aspects and rules[5] to the game, of which the ones relevant to the research will be explained below. If the reader is familiar with Catan, they are encouraged to skip this section.

3.1 General Game-play

Every game/match of Catan consists of rounds, which consist of turns; 1 for every player. The amount of rounds in a match can, in theory, be infinite because players can always pass during their turn to end it. Every turn of a player consists of 3 distinct phases. In the first phase dice are thrown (see **3.3**), in the second phase trading between players is possible (see **3.4**) and in the third phase the current player can buy and place buildings (see **3.5**), given he/she/it has the resources to do so. The trading and building-phases can be passed upon at will of the current player, even though in most situations this is only done when forced to do so. The rule-book of Catan states that matches end when a player reaches 12 victory points[2], however this is sometimes adjusted in accordance with the preferences of the players. How VP can be acquired will be explained in the subsections **3.5** and **3.6**.

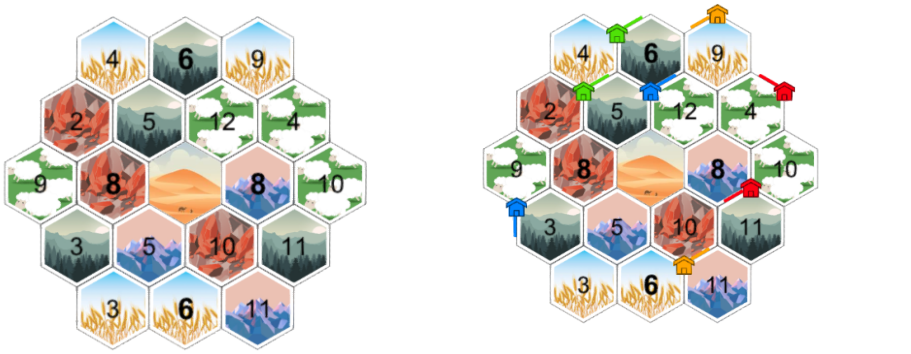
Catan can be played with 3 or 4 players. The order of players is determined by choosing one of the players at random that gets to go first, after which the turns go clockwise depending on the seating positions of the players. During their turns, players will collect or lose resources and strengthen their position using their collected resources (called 'in-hand' resources), of which there are 5 types. The resource-types are the following (in no particular order): Grain, Brick, Wood/Lumber, Wool and Ore. These resources can be spend on buildings or Development Cards during the building-phase of players' turns. There are three kinds of buildings; the Street, the Village and the City. These have different costs and advantages and will be discussed further in **3.5** .
There is also the so-called "bank", which holds the stockpile of resources that can still be collected and the unbought Development Cards. The bank starts with 19 of every of the 5 resource-types. The Development Cards are discussed in the Miscellaneous Rules **3.6**.

3.2 The Board

The board consists of 19 hexagonal tiles, laid out in 5 horizontal rows. From top-to-bottom, the amount of tiles in the rows are 3-4-5-4-3. See Figure **1a** and **1b** below for visualisations. The vertexes of these hexagonal tiles are called gridpoints. Some of these gridpoints are special in that they give certain advantages when occupied. How gridpoints become occupied will be explained in the

[2]Victory Points will be abbreviated to VP, to avoid confusion with rewards/points of the RL agents.

building subsection **3.5**. These special gridpoints are called harbors and will be discussed further in the Trading subsection **3.4**. The harbors and gridpoints are also visualized in figure **2**.
Every tile is associated with 1 of the 5 resource-types, except the middle tile[3], which has no value and is called the "Desert Tile". Every tile is also associated with a number (again, except the Desert Tile), ranging 2 to 12 (both inclusive) excluding 7. Each of the numbers correspond to a dice roll (see **3.3**). Of the included numbers, all but 2 and 12 appear twice on the board. Every tile can only be associated with a single resource-type and a single number. The associated resource and number of/on a tile don't change over the course of the game. As can be seen in figure **1** (and easily be deducted from the total number of tiles), the resources are not evenly spread over the number of tiles. Wool, Grain and Lumber (dark green) are associated with 4 tiles, while Brick (red) and Ore (indigo) are associated with only 3 tiles. Some basic math learns us that 6 and 8 have the highest probabilities (around 0.14) of all *included* numbers that are the sum of 2 dice. Tiles associated with these numbers are considered "high-chance" and are not allowed to be next to each other, to prevent large unfair advantages, which will become clear in the following subsections.



(a) Initial board.

(b) The initial board after players have placed their initial villages and streets.

Figure 1: The default board layout in 2 stages of the game. The board layout depicted is the layout suggested by the handbook that comes with the basic set of Catan.

3.3 Dice - The First Phase

The dice-phases of turns are the main method for players to gather resources. Catan is played using 2 regular dice with 6 sides. In the dice-phase of a turn

[3]Readers with experience in Catan will note that the Desert Tile can be placed anywhere and doesn't need to be in the center of the board. This is true and centering the Desert Tile here is solely because the default board in the rule-book has the Desert Tile in the center.

(which is always the first phase and cannot be skipped), the 2 dice are thrown at the same time. The eyes of the dice are summed and used to determine which players get which resources and how many. This process is fairly straightforward: Every player checks the tiles which are associated with the same number as the result of the dice roll. If a player X has at least 1 Village or City connected to this tile / these tiles, than player X gets the resource associated with that tile. The amount of said resource that player X gets is the sum of the amount of connected villages and cities to that tile. How and when villages and cities are connected to tiles is discussed in section **3.5**. Each connected village provides 1 resource and each connected city provides 2. For example, if 6 is the result of the dice roll and the current state of the board is the one depicted in **1b** , than the green and blue player would both get 1 Lumber and the yellow player would get 1 Grain. In the context of Catan, a player getting/collecting a resource means that the resources are deducted from the stockpile of the bank (if possible) and become in-hand cards for the receiving player. A player may hide the types of the resources that are in-hand, but not the number of in-hand cards in total.

3.4 Trading - The Second Phase

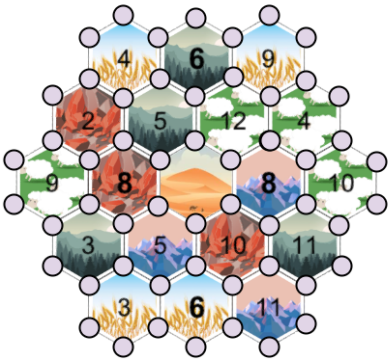
Trading in-hand resources is an essential part of Catan. It is the main interaction with other players in the game. Making good trades can speed up progress a lot, especially in the first rounds of the game. However, handling trading by reinforcement learning is beyond the scope of this research, mainly because of limitations to computing power. Therefore, an heuristic algorithm is used to handle trading. This simple algorithm will be discussed in section **5.1**. The rules of Catan state that trading is the second phase of every turn, after rolling the dice[4].
In this 'trading-phase' of the turn, trading is initiated by the current player[5]. The current player must be involved in all trades made during this phase; i.e. players whose turn it currently isn't cannot trade with each other. They can, however, accept a proposal made by the current player, make a counteroffer or deny said proposal altogether. The current player can also always trade with the bank (called 'domestic-trading'), which only accepts 4 to 1 trades (read: give-4-take-1 trades) in which the proposal must always contain 4 resources of the same type to give and 1 resource of yet another type to take.
The 4:1 ratio trading is of course very costly and can be improved upon by placing villages and cities on specific spots on the board, called harbors or ports. See figure **2b** for a visualisation of an example of where the harbors can be positioned. These harbors are only accessible on certain positions on the rim/edge of the board, or more formally, only vertexes/gridpoints that are shared by less than 3 hexagons/tiles can be considered for a harbor. A single harbor always consists of two gridpoints that share an edge. Two different

[4]For readers that have played Catan before: It must be noted that this strict take on the rules is not always shared among players. Some allow free trading throughout the whole turn and even among players whose turn it is not.

[5]'Current player' refers to the player whose turn it currently is, i.e. the player in-turn.

harbors must be separated by at least 2 subsequent edges. Because there are 9 harbors, harbors are always distanced by 2 or 3 edges.
Harbors come in two types of which the first type is the 'random-harbor', which improves the ratio of domestic trading from 4:1 to a 3:1 ratio. The second type concerns resource-specific harbors that improve domestic trading of the corresponding resource to a ratio of 2:1. These trading ratios make a huge difference over the course of a game of Catan, but have a drawback: To get to the harbors, you need to build on the 'coast', where the villages or cities can only be connected to 1 or 2 tiles, instead of 3. There are always 4 'random-harbors' and 1 resource-specific harbor per resource. See the next subsection for more information on the gridpoints and the placement of buildings.

Consider the following example: Player Alice did not balance her resource income well and has 7 resources in hand: 5 lumber and 3 grain. She has access to a random harbor and a grain-harbor. With her current resources the following actions are available: She could trade 2 grain for a brick and build a road or she could trade 3 lumber for a brick and 2 grain for a wool and build a village. If she had no access to harbors, she would only be able to trade 4 lumber for a brick and build a road and been left with 3 grain.



(a) The initial board with the 54 gridpoints visualized in purple. On these gridpoints, villages and cities may be placed.

(b) A visualisation of the harbors placed at random on the default board. All harbors are gridpoints, but not all gridpoints are harbors.

Figure 2: Visualisations of the gridpoints and harbors.

3.5 Building - The Third Phase

In the third phase of a turn, a player can build as many buildings as he/she wants, using the resources obtained during the dice- and trading-phases. This also includes building nothing and thus skipping this phase completely. After buying and placing a building, the resource-cost of the building goes from the

hand of the player into the stockpile of the bank. There are 3 kinds of buildings, each with different rules of when and where they are allowed to be placed.
The first is the Street, which are depicted by a bar in the color of the player that owns them in the visualisations of the environment. Streets are only allowed to be placed on the edges of the hexagons and there cannot be 2 streets on 1 edge at any point in time. That edge is then occupied by that street. Streets always have to connected to other streets[6], villages or cities[7] owned by the same player. Streets have a resource-cost of 1 Brick and 1 Lumber and are worth no Victory Points. Each player can build a maximum of 15 streets in a single game.
The second type of building is the Village. They are depicted as colored houses in the visualisations throughout this paper. Villages can only be placed on empty vertexes/gridpoints to which an edge is connected that is occupied with a street owned by the same player. The resource-cost of a village is 1 Brick, 1 Lumber, 1 Grain and 1 Wool. This makes the village the only purchasable thing in the game that costs 4 different resources. It is not allowed to build a village on a gridpoint/vertex when that vertex shares an edge with another occupied vertex. This is called "The Distance Rule". See figure **3** for a visualisation. Villages are worth 1 Victory Point and provide 1 resource per connected tile. A player can build a maximum of 5 villages in a game.
The third type of building is the City. It is essentially a direct upgrade for the village, costing 3 Ore and 2 Grain. A city can only be build on a GP where there is already a village of the current player, replacing said village entirely. The replaced village is returned into the stockpile of the player, providing the player with 1 more village to build. A city is depicted by a church-like icon in the visualisations of the game. A city is worth 2 Victory Points, but it is worth noting that, because it replaces a village, it practically only adds 1 VP. Aside from the VP, a city also provides 2 resources per connected tile, doubling it from the original village. A player can build a maximum of 4 cities in a game.

Players do not start with any in-hand resources in the base-game of Catan. They do, however, start with 2 free villages and streets. These free villages can be placed wherever the player wants, but the placement has to abide to the Distance Rule. The 2 free streets must be connected to different villages. To keep the game fair[8] the order in which the players may place their free villages and streets is fixed. The starting player (which is chosen at random) may first place 1 free village and 1 free street, which must be connected. Then, in clockwise order based on the seating, the other players place 1 free village and street as well. Then the process repeats, but in reverse. The player that placed their village and street last may now go first. In counter-clockwise seating order the players may place their second free village and street, until every player has 2 villages and streets on the board. After this initial phase, the starting player

[6]Connected in the context of streets means: There needs to exist a vertex to which both edges (where the streets are positioned) are connected.

[7]In the context of cities and villages; streets are connected if the edge (they are positioned on) is connected to the vertex the village or city is on.

[8]Not entirely fair, see [1] for effect of the seating order.

starts his/her turn, which of course starts with rolling the dice.

Figure 3: A visualisation of the Distance Rule. If we start with the village with the black square around it, the villages surrounded by the red square cannot be placed because they share an edge with the starting village. The villages with the green square around them *can* be placed because they don't share any edge with the starting village.

3.6 Miscellaneous Rules

There are also a variety of miscellaneous rules in Catan. Arguably the most important (gameplay-wise) and disruptive is the Robber. The Robber comes in the Catan box as a black figurine which resides on the desert tile. Whenever the current player rolls a 7 summed with the dice, the Robber comes into play. The current player chooses a certain tile on which he/she places the Robber figurine. This tile must be connected to villages or cities of other players[9], or more formal, at least 1 vertex of the hexagon must be occupied by a village of city. After the current player places the figurine, he/she chooses one of his/her opponents and takes one of their hand-cards at random. The current player must choose a player whose villages and/or cities are connected to the robber-occupied tile. While the Robber resides on a tile, that tile will not generate any resources, no matter what number is rolled with the dice. The only way to get rid of the Robber is to throw 7 yourself and move it to another tile where it doesn't bother you anymore. Furthermore, whenever 7 is rolled, every player that has more than 7 hand-cards/in-hand resources must immediately return half of their resources to the stockpile. Players can choose which of their cards are returned and do not have to show to the other players which resource-type(s) he/she returns, only that it's the right amount.

Next to the resource cards there exist the Development Cards. These development cards are each bought with 1 Ore, 1 Grain and 1 Wool and are used for

[9] Of course, the tile chosen may also be connected to villages and cities owned by the current player.

a variety or different things. These include straight up giving Victory Points to the owner, the ability to steal cards from other players at will and Knight cards. Knight cards enable the owner to move the Robber and steal a single card from a chosen victim. Used Knight cards also count towards the Largest Army. When a player has used 3 Knight cards, they gain the Largest Army card which counts for 2 Victory Points. Whenever another player has more used Knight cards than the player that currently holds the Largest Army, they gain the Largest Army card and the associated 2 Victory Points (which means 2 VP's are deducted from the previous Largest Army holder). The Longest Road works the same way as the Largest Army regarding points, but instead of used Knight cards, the number of consecutive connected streets are counted. Every road can only be counted once.

4 Reinforcement Learning Algorithm

The focus of this research is the performance of an agent based on an Actor-Critic neural network in Catan. Actor-Critic is an umbrella term for a method that encapsulates similar algorithms. The most prominent feature all Actor-Critic algorithms share is that they consist of two parts, called the Actor and the Critic. On a high level, the Actor determines the executed action, while the Critic gives feedback on this action. The specific Actor-Critic algorithm used in this paper is called Advantage Actor-Critic, or A2C for short. A2C is a very versatile reinforcement learning technique because of its great generalization capabilities. The current state of the art algorithm used in research surrounding games and RL is called A3C (Asynchronous Advantage Actor-Critic). This algorithm is essentially boils down to running a lot of instances of an environment and A2C simultaneously, where every A2C instance controls an agent in one of the environments. Because of time- en resource-limitations, A3C could not be used in this research, simply because it requires a lot of instances of the environment to be ran simultaneously to be effective.

4.1 Setup of the Actor-Critic Agent

A2C consists (just like all Actor-Critics) of an Actor and a Critic. In this research, a neural network was used to implement A2C in. It is often the case in practice that the Actor and Critic share a part of a neural network for stabilization. The neural network used here consisted of 6 layers. The first was the input layer with a size dependent on the environment type, see **5.2.1** and **6.2** for more information. The following 3 layers were fully-connected, so-called Dense layers, and shared by both the Actor and Critic network. They consist of 300, 200 and 150 nodes respectively. Connected to the last Dense layer are the Actor layer and the Critic layer. The output of the Actor is a vector with probabilities for every action. The Critic (layer) on the other hand only consists of a single node activated linearly by the previous (Dense) layer. A visualisation of the network can be found in figure **4**. When the network is training, the Critic is comparable to a Q-learning[10] algorithm where a neural network (NN) approximates the Q-function using gradient ascent, which is done using back-propagation in the case of an NN.

4.2 A2C Algorithm

In this paper A2C was used which, in contrast to more bare-bones Actor-Critic algorithms, forces the critic to approximate so-called 'Advantages' or 'advantage-function'. This advantage function stabilizes the network by reducing the variance because the estimated values are used indirectly (instead of

[10]In Q-learning, values (so-called Q-values) for each action in every state (the state-action pairs) are calculated and updated using rewards (read: feedback) from the environment. A strategy/policy can be formed using these Q-values by choosing the action that leads to the state with the highest Q-value.

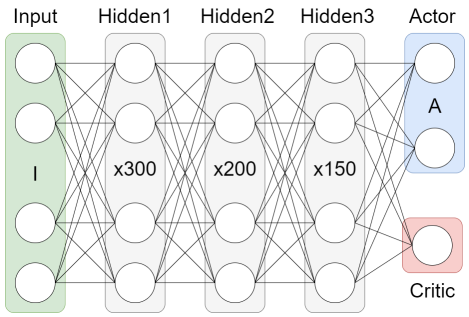


Figure 4: A visual graph representation of the neural network used. The number of nodes in each layer is noted in the middle of each layer. The 'I' in the input layer indicates that the number of nodes is dependent on the specific environment used. The same holds for the actor layer, where 'A' indicates that the number of nodes depends on the action space of the environment.

directly) to provide feedback to the Actor. This is done by calculating the value of the executed action in a given state. If advantages would not be used (i.e. in a 'standard' Actor-Critic network), the Critic values would serve as a value estimate for the state only. Mathematically the advantage function, given a state and action, $A(s, a)$ is defined as:

$$A(s, a) = Q(s, a) - V(s) = r + \gamma \cdot V(s') - V(s) \tag{1}$$

In the advantage equation, s and s' are the current and next state respectively. The value of a state s is defined by $V(s)$. A discount factor γ is applied to discount future rewards and $Q(s, a)$ is the standard Q-value of state-action pair (s, a). The reward gained from taking action a in state s is represented by r. In practice, these advantages are used to weight the different samples that are used to train the Actor network.

4.3 Implementation

The actual code for the network was provided on Medium.com by Python-Lessons [6], which was actually originally used for the video-game classic Pong. However, due to the flexibility and generalizability of this Actor-Critic implementation, it could be modified to work with our current environment. This code was build on the free Python machine-learning platform Keras, powered by Tensorflow.

Some changes were made to the code to fit our environment(s). Because our environment has a discrete action space, the Actor used a softmax activation for its output. The softmax activation function σ is defined as follows :

$$\sigma(\boldsymbol{s})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{where } i = 1, .., K \text{ and } \boldsymbol{z} = (s_1, ..., s_K) \in \mathbb{R}^K. \tag{2}$$

The input vector (the current state) is represented by $\boldsymbol{s}$, and its $i-$th element is thus s_i. The contents of the input vector $\boldsymbol{s}$ and what they represent are explained in **5.2.1** and **6.2**. K represents the number of elements in vector $\boldsymbol{s}$. The softmax function essentially normalizes a vector into a probability distribution. In context of Catan, this probability distribution contains the probabilities of each action being taken. See **5.2.2** for more information on the action space of the agents.

Both the Actor and Critic used Mean-Squared-Error as their loss-functions. Loss functions are used in the context of neural networks to provide feedback during training on the performance off the network. The network will update its weights such that the loss function is minimized. The loss function takes a prediction and the actual value and returns some number that indicates the difference between the two (according to that loss-function). This loss value is indicative of the performance. The Mean-Squared-Error loss function is defined as follows:

$$\text{MSE}(P, R) = \frac{1}{n} \sum_{i=1}^{n} (P_i - R_i)^2$$

In this equation, P and R are vectors of length n containing the predictions and actual values respectively. In the case of the Critic, the predictions are the estimated values of given states, while the real values are the collected reward in those states.

Furthermore, the Actor and Critic both used Adam as their optimizers (unless stated otherwise). Optimizers are used in the context of neural networks as the algorithm to update the weights of the connections between the nodes/neurons. Choosing a optimizer can be critical to the performance of a network. Because most optimizers (including Adam) are enhanced versions of standard stochastic gradient descent, the learning-rate/step-size parameter of the optimizer plays a significant role in the convergence speed and performance of the network. Adam[7] has proven itself being versatile and memory- and computationally inexpensive, while maintaining great convergence speed and performance.

Of course, not every imaginable action is legal at every stage/state of the game. For example, you cannot place a village too close to another village according to the distance rule, or place a city on top of another players' village. Because of this, a so-called 'action mask' (or mask for short) was used. The

action mask $\boldsymbol{m}$ is a vector and defined as follows:

$$\boldsymbol{m}(\boldsymbol{s})_i = \begin{cases} 1 & \text{if action } i \text{ is legal.} \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The mask takes a state s as its input and is different for every player in each state[11]. The output of the Actor-layer was again normalized to take this mask (and as such, illegal actions) into account. Practically this meant that the probabilities in the output of the Actor network were set to 0 where they corresponded to an illegal action according to the action-mask. An action was chosen using this normalized and 'masked' probability-vector. Mathematically, this entire process from state to action can be viewed as:

$$A(\boldsymbol{s})_i = \frac{\sigma(N(\boldsymbol{s}))_i \cdot \boldsymbol{m}(\boldsymbol{s})_i}{\sum_{j=1}^{K} \sigma(N(\boldsymbol{s}))_j \cdot \boldsymbol{m}(\boldsymbol{s})_j} \tag{4}$$

Here, $A(\boldsymbol{s})$ represents the vector containing probability for each action i given input vector/state $\boldsymbol{s}$. The length of A is represented by K. The softmax activation function is represented by σ as before. The parameter of σ is filled by $N(s)$ which represents the output of the third hidden layer as received by the Actor layer.

The pseudo-code for the entire algorithm can be found below. The 'Train' procedure is used to train the agent using the taken actions, collected observations and rewards. The 'TakeAction' procedure returns an action from the Actors' policy given a state. It also accounts for the action mask.
Please note that fitting / training in the case of a neural network means adjusting the weights of the network. Other methods for function approximation can use different terminology for this behaviour than 'Fit' or 'Train'. As stated before, the Fit procedure-call on line 11 includes *advantages* as weights for the samples (which are the inputted states S) during fitting.

[11] This is because the positions where buildings can be build or if they can be build at all is depended on the current resources and placed buildings of a certain player. This results in different states for every player, even if nothing about/on the board changes.

Algorithm 1 A2C

$A \leftarrow$ Actor
$C \leftarrow$ Critic
$S \leftarrow$ States / Observations collected during an episode
$X \leftarrow$ Actions taken by this agent during an episode
$R \leftarrow$ Rewards collected during an episode

procedure TRAIN(A, C, S, X, R)
    $D \leftarrow$ Discount(R, γ) ▷ Discount R using discount factor γ
    $V \leftarrow$ Predict(C, S) ▷ Value prediction of Critic
    $advantages \leftarrow D - V$
    Fit(A, S, X, $advantages$) ▷ Fit the Actor on S given X
    Fit(C, S, D, None) ▷ Fit the Critic on S given D
end procedure

procedure TAKEACTION(A, state, mask)
    $\overrightarrow{P} \leftarrow$ Policy(A, state) ▷ Policy / Probability vector
    for iteration $i = 1, 2, \ldots, |\overrightarrow{P}|$ **do** ▷ Account for mask
        if mask_i is True **then** ▷ mask_i is illegality of action i
            $P_i \leftarrow 0$
        end if
    end for
    $\overrightarrow{P} \leftarrow$ Normalize($\overrightarrow{P}$)
    $action \leftarrow$ Choose($\overrightarrow{P}$) ▷ Choose action using probabilities $\overrightarrow{P}$
    return $action$
end procedure

5 Reinforcement Learning Environment

To convert the world of Catan to a digital one, the Unity game engine was chosen. Unity provides a package called ML-agents that can interact with Python seamlessly and is very suitable for making small games and simulations. The game was made quite bare-bones, with little to no overhead to speak off considering graphics (as can be seen in figure **2**, which depicts screenshots of the actual game). Because of time-restrictions, changes had to be made to the game to remove some of the complexity.

Before discussing these changes there must be a definition of the 'core' of the game. This core consists of the game-play elements which, when removed, make the game unplayable in the sense that the game cannot progress and/or finish. When changes to the environment were made, the core of the game was kept in mind, in order to maintain a game representative to the original game. In the case of Catan, the core of the game is the building of buildings using resources gathered from dice-rolls. Without this, a game of Catan would never finish because no player is able to reach the goal of the game, i.e. get an x amount of Victory Points. Below are the changes made to the game in the environment, which we will call Environment U (short for Unity) from now on.

5.1 Changes

- 1. The Development Cards, Longest Road and Largest Army cards were removed. These cards complicate the game quite a bit and only serve to speed up the progress of players.
- 2. The goal the game was reduced to 8 Victory Points.
- 3. Trading between players was removed but trading with the bank was still possible. While trading is an essential part for cooperation in Catan, it theoretically only speeds up the progress of the players.
- 4. The resources on the tiles, locations of harbors and numbers on the tiles were fixed (i.e. the map was fixed). This was primarily done to reduce training time. This of course reduces the generalizability of the agent, see Discussion section for more information on this.

This changed game still tries to contain the core gameplay of Catan, while reducing the complexity of the game (and thus the training time) by quite a lot. See the Justification subsection (**8.3**) for more information.

It important to note that the trading procedure was the same for all players and was based on the following simple heuristic method; If a player has excess resources, it trades the excess for resources where the player has the least of. For example, in environment U, the player can use at most 1 Wood per step because of the costs of the buildings. If that player has 5 Wood, it will trade

the 4 excess Wood automatically for the resource it has the least of, with the bank.

5.2 Configuration

Each player in this environment consisted of 2 parts, the 'executor' and the 'brain'. The executor only existed in the Unity game environment. It collected the observations and executed given actions. The collected observations were used as inputs for the brain and will therefore be called inputs from now on. The brain only existed in the Python scripts and contained the neural network, among some prepossessing and utility code. The inputs given by the executor were used as the inputs for the input-layer of the neural network. The neural network then processed these inputs, which leads to a certain output (of the Actor part of the neural network). This output is interpreted as the action, which is communicated to the executor and executed. The following subsection; 'Input Space' contains the properties of the collected observations (collected in-game by the executors) and how this was used by the brain. The subsection 'Action Space' contains information about how outputs of the neural network (the Actor layer specifically) were processed and interpreted as actions. The last subsection contains information about the setup of the Actor-Critic algorithm and how it relates to the environment.

5.2.1 Input Space

The input data was as raw as possible and was a vector with length 437. The first 5 inputs were the amounts of resources of each type currently 'in-hand' by the current player, while the 432 following inputs were 54 stacked vectors of length 8. (see figure 5).

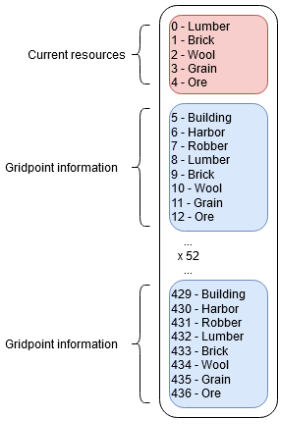


Figure 5: Visual representation of the input space.

Each of those 54 vectors represent a gridpoint (GP) on the board. These GP's are the crossroads between tiles on the board, as can be seen in figure 6

and figure 2a. The following data is represented in a given GP vector; Firstly, whether there is a building present. Secondly, whether or not the GP is a harbor. Third, whether the Robber is affecting this GP. The fourth until eight element of the GP vector are the resources values of this GP, calculated by summing the probabilities of each connected tile using its associated tile number. See figure 6 for examples. The input data is raw, as this information is directly coming from the board and none of the inputs given can be inferred from the other inputs. As can be seen from the visual representation, the input provided to the agents given a certain state only possibly differs in input 1 for every GP, which has a value of 0 when there is no building, a value of 1 when the village or city on the GP is owned by the current player and a value of -1 otherwise.

Next to this input vector, the agents also receive an action-mask in every state which is a Boolean vector with the same length as the action-vector (see 5.2.2). This action-mask contains for each action whether or not it's legal or illegal (masked). For agents with hard-coded behaviour, these action-masks are used directly for decision-making, while for the Actor-Critic agent they are used in post-processing of the network-output. More on this in the Actor-Critic section 4.

Figure 6: GP value example. In this example the resource values of yellow gridpoint are Lumber:$\frac{1}{9}$, Ore:$\frac{1}{18}$ and Grain:$\frac{5}{36}$.This means that the resource-value part (element 4 through 8) of the vector in the input space becomes: $(\frac{1}{9}, 0, 0, \frac{5}{36}, \frac{1}{9})$

5.2.2 Action Space

The action space of every agent consists of a pass / no-op action along with actions for all possible combinations between GP's and building types. This means that in Environment U, the action space consisted of 163 different actions. Every agent outputs a vector with probabilities for every action. Outputs were re-normalized after taking into account the action mask. This means that the probability of every illegal action is corrected to 0 and the rest of the probabilities is normalized. This was proven by Vinyals et al. [3] to be quite effective

when dealing with large action spaces where conditional actions are common. The rules of Catan do not allow building streets on GP's. Therefore in this context it means that building a street 'on' a GP actually results in building a street towards that GP. The game logic automatically connects the other end of the street to the closest village or road owned by the same player.

5.2.3 Setup

At first, the training of the agent was step-based. This meant that the main agent (the agent that is controlled by the Actor-Critic neural network) trains its neural network every step, using the observation and reward received said step. In this context, a step is single environment step in which the main agent is 'in-turn'. An agent only receives input and is only requested output when its the turn of said agent. Step-based training generally has a reduced training-time compared to episode-based, but had a huge drawback; It did not fit Catan as a game. Catan is a strategic game and can't be played at a decent level when played step-based. Applying this concept to humans would be like erasing your memory of the current game every time you roll the dice. This would be unplayable and this step-based learning method was therefore scrapped in an early stage of the research. The training was then converted to episode-based. This means that observations and rewards are collected over the course of a whole match[12] of Catan. When the match has concluded, the neural network is trained using these observations and rewards. In theory, this would allow the agent to develop a real strategy / policy that spanned over a whole match. However in practice, the more the agent learned (i.e. showing a decreasing trend in the losses), the less VP it would gain in a match. See the next subsections for an analysis. All the experiments were done using 4 players, one of which was an Actor-Critic agent, while other three were Random-Agents. The Actor network used a softmax activation, while the Critic network was activated linearly. Both used mean-squared-error as their loss-functions.

Measurements

In general, graphs that show results in this paper will all use the same format unless stated otherwise, that is; In blue the running average of a certain metric with next to it in red the sliding average of that same metric using a window of a thousand episodes. The running average (RA) is defined as :

$$RA(X) = \frac{1}{N} \cdot \sum_{x \in X} x \tag{5}$$

Here, X is the set of all data points up until a certain time. The size of set X is expressed by N. If data-points are collected on a regular interval, as is done here, N is equivalent to the current interval (which is the current episode in this case). This RA is used to show the overall trend of the entire run.

[12]The terms match and episode mean the same thing here. The term 'episode' is more common in literature in context of training neural networks.

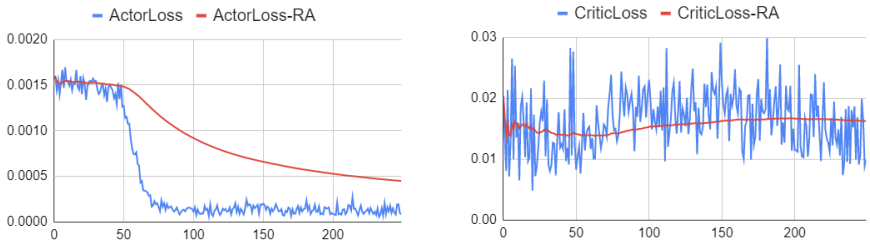
The sliding average (SA) is similar to the running average, except that it can show trends on a smaller scale and can thus be more detailed. It shows trends over a configurable 'window'. The definition of SA is as follows:

$$SA(X) = \frac{1}{\min\limits_{N,w}} \cdot \sum_{i=\max\limits_{1,N-w}}^{N} X_i \tag{6}$$

Here, X and N have the same definitions as in the running average 5. An indexed episode of X is given by X_i with index i. The window of the SA is given by w. A window of 1000 was used in all experiments. The max statement is used to prevent the indexes from going into the negative values when $N < w$, i.e. when X is too small for the given window. The min statement is used to correct the averaging whenever the $N < w$.

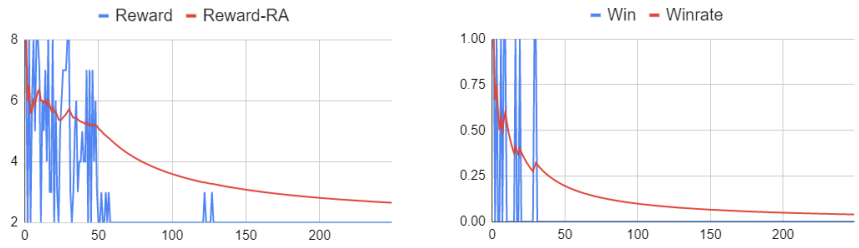
5.3 Results

On average, the number of steps in a single match/episode was between 370 and 390. The loss of the Actor was decreasing quite fast, as can be seen in Figure 7a. It can be seen to diminish after 50 episodes, in contrast to the Critic, which does not show any meaningful decreasing of loss. A first intuition would be that the agent is actually learning because the Actor-loss is decreasing. However, the Critic and Actor are deeply connected and as such, this cannot be concluded. As explained in section 4.2, the Critic output determines the weights of the samples used in training the Actor. As can be seen in figure **7b** the loss of Critic is very ecstatic. That is indicative of randomness/noise in the output of the Critic. The weighting of the samples for training the Actor will therefore be quite random as well. As a consequence this the Actor learns that passing is the 'best' action because that's the safest way to handle a random (weighting) distribution of states. In figure **8** the result of this behaviour can be easily seen; the Actor does not attempt to score points anymore.



(a) (b)

Figure 7: The losses of the Actor and Critic with their respective running averages (RA) in red. For a definition of RA, see equation **5**.



(a) (b)

Figure 8: On the left the Reward / VP per episode and on left the wins in blue and both their respective running averages in red. The RA in the graph that displays the RA is indicated by win-rate as a running average of wins and win-rate are equivalent.

Of course, 250 episodes can hardly be called a training[13]. However, the behaviour of the Actor-Critic agent collapsed into only passing / no-ops after 50 episodes, even when the resources for building where there. This was consistent over dozens of runs and the trends were comparable to those in figure **8a**. This result was independent of the learning rates, which ranged from $1.0 \cdot 10^{-4}$ to $1.0 \cdot 10^{-7}$ during testing. In the figures, a learning rate of $1.0 \cdot 10^{-7}$ was used. A 4-player environment meant 1000 step episodes on the regular, which is not feasible when it takes 5 seconds per episode on average. For reference, the Monte-Carlo Tree Search algorithm used in [1] could play 300 games / episodes per second. It therefore was decided to re-create the environment in Python. This would eliminate the extra communication layer and processing done in the Unity Engine and thus speed up training.

[13]When Deepmind trained an Actor-Critic agent to play Starcraft II, it took in the tens of billions (as in 10^{10}) of episodes to train to perform similarly to human experts.

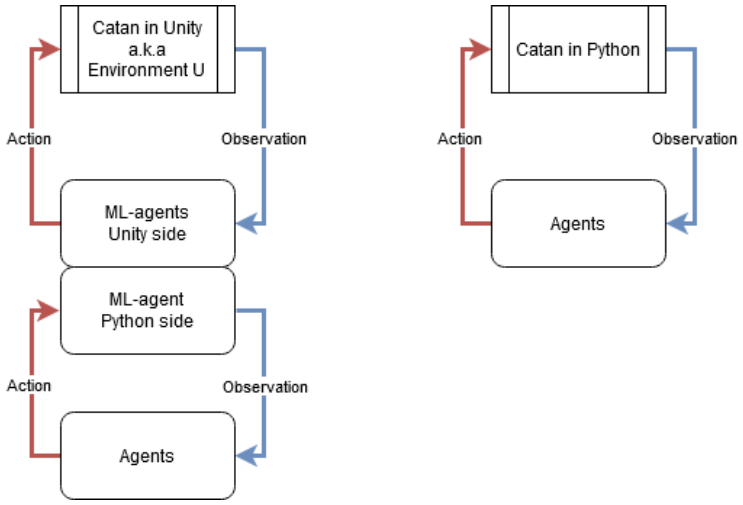


Figure 9: A high-level representation of the environments. The main difference between the two environments is that; In environment P, everything is handled in Python while in Environment U the game-logic is separated from the agents' 'brains', which necessitates an extra communication layer. This communication layer adds overhead but also provides the game with a visualisation, where Environment P is all in code and command-line. There are also differences between the rule-sets used in the environments, see **6.1**.

6 Environment Overhaul

Environment U yielded unsatisfactory results. This was mainly due to the setup of the environment. Episodes played using Environment U took too many steps (and therefore too much time) to conduct any type of meaningful iterations on the algorithm and/or analysis within the time constraints. As such, the Catan game was re-created in pure Python. The high-level difference between the 2 environments is visualized in figure **9**. In short, remaking the environment in Python eliminated the required communication layer of environment U, which increased time-efficiency. The following adjustments and (further) simplifications were made to the setup used in Environment U. This modified/simplified environment will be called Environment P.

6.1 Simplifications

- **1.** The game was reduced to a 2-player game. This means that in the experiments, 1 Actor-Critic agent and 1 RandomAgent were used. As a consequence, it reduces the game from a multi-agent problem to essentially a single-agent problem, as a single RandomAgent can be considered as an extension of the environment. In a multi-agent 4-player game this cannot be done as the other agents are also working against each other.

- **2.** The amount of Victory Points needed to win was reduced to 3.
- **3.** Cities were removed from the game. Cities are essentially a direct upgrade to the Village and their removal has no influence on the core of the game.
- **4.** The amount of resource-types was reduced to 3. This meant that Wool and Ore were removed from the game. As a consequence, the cost of a Village was reduced to 1 Brick, 1 Wood / Lumber and 1 Grain.
- **5.** Harbors were removed entirely. Harbors are not part of the core of the game, they are used only to speed up progress of players by providing better trading ratios. To compensate for this, the default trading ratio with the bank was reduced to 3:1 from 4:1.
- **6.** The map lay-out was changed from a 3-4-5-4-3 layout (vertically) to a 2-3-2 basic honeycomb layout to cater to a 2-player setup. See figure 10 for a visualisation. As a consequence, there could only be 5 different numbers on the tiles, which meant that the 2 'dice' got 3 sides each.
- **7.** The Robber was removed from the game to increase training time. It has been observed that removing the Robber halves the number of steps per episode.

Figure 10: Visual representation of the board used in environment P. The number 4 is not present on the tiles because the number with the highest probability is reserved for the Robber. Because the Robber is removed, rolling 4 does nothing.

6.2 Configuration

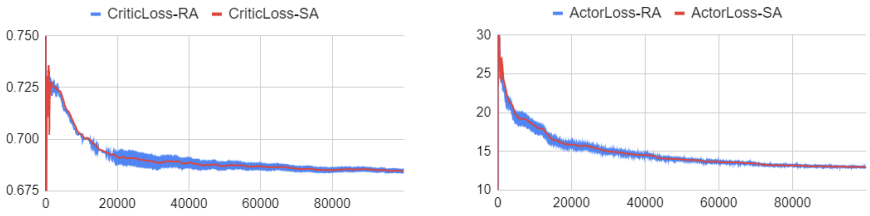
The input and action spaces were handled the same as Environment U, but were both reduced in size. The current resources, which were input nodes 0

to 4 (inclusive) in Environment U, where reduced to input nodes 0 to 2 in the input vector because of the simplification of the resource types. The gridpoint-information was also reduced from 8 to 5 input nodes, because of the removal of harbors, resource-types and the simplification to a 2-3-2 map layout. The action space was also reduced to 49 actions, 2 actions for every of the 24 gridpoints, plus a no-op / passing.

A few setup changes were made during the training in Environment P. Both the Actor and Critic used a RMSprop[14] optimizer with a learning rate of 0.001. The maximum number of streets was limited to 10 to prevent players from completely crowding the board with streets (a similar limit of 15 exists in the original board game). A small negative reward (called Standard Step Reward, or SSR for short) was introduced every step, to stimulate the agent to end the game as quickly as possible. Building buildings and thus progressing is not enough to consistently win games; placement and timing is also important. In these experiments, an agent would get -10 reward for losing (unless stated otherwise) and +10 reward for winning, while receiving 1 reward per village.

6.3 Results

First, an experiment was done of 100.000 episodes. The environment had no overhead in contrast to Environment U. This meant a huge (over ten-fold) speed-up in training, reducing the average steps and taking overall less seconds per step and episode. The results are given in the graphs in **11**. On average, an episode took 240 steps and 0.3 seconds. As can be seen from the losses in figure **11**, the agent is definitely learning.

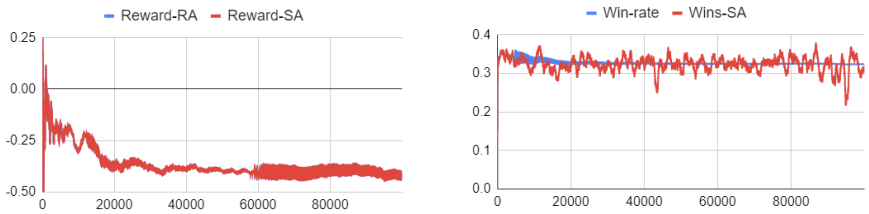


(a) (b)

Figure 11: The losses of the first experiment with 100K episodes. See equations **5** and **6** for explanations regarding RA and SA.

However, the rewards and wins tell another story. The reward shows a decreasing trend, which contradicts the intuition that a decreasing loss should yield 'better' results. This contradiction can have different causes. At the time, the reward function was considered to be the problem.

[14]RMSprop is infamous for never being proposed in a paper but was instead proposed in a lecture by Geoff Hinton[8]. It is a generalization of the Rprop algorithm, adjusted to work with mini-batches.



(a) (b)

Figure 12: The rewards- and wins running- and sliding averages. In the left graph that represents the rewards, the running and sliding averages overlap.

The reasoning was as follows: Every episode takes around 240 steps, which is split evenly the among the players, resulting in 120 steps per player per episode. This results in 40 steps on average per village per player. As the SSR was only -0.01, it did not provide enough stimulation to the agent for building a village as quickly as possible. Decreasing the SSR to -0.02 or -0.05 did not seem to improve the results[15]. Results using a SSR of -0.02 yielded approximately the same results as -0.01, just with 0.75 less average reward (-1.12) over 25000 episodes, while using a SSR of -0.05 yielded an average reward of -6. This practically means that the agent almost exclusively received negative rewards. However, eliminating a losing reward entirely and/or doubling the winning reward did not yield any change in trend regarding the rewards, winning rates or losses. This meant that the reward design wasn't an explanation for the behaviour of the agent. You could argue that tinkering with the reward function further could produce the 'expected' behaviour. However, modifying the reward design to get the desired result shouldn't be considered, because of the enormous designer / research bias associated with that decision.

After this initial experiments, 20 consecutive runs where done of 10.000 episodes. The results were averaged over these 20 runs for a more stable results. In figure **13** the running- and sliding averages can be seen of the Actor and Critic losses of an Actor-Critic agent that was pitted against a RandomAgent. These results will be used in the analysis going forward. No losing reward was used during these runs.

7 Further Exploration

The graphs in figure **13** show a clear decreasing trend in the losses of both the Actor and Critic. This indicates that the agent is learning and thus a formation of a certain policy / strategy. Also, analysis of individual episodes revealed that the probability of passing goes down during a run. This indicates that the agent

[15]"Improving the results" here means; eliminating the contradiction that is: reduced loss is associated with decreasing rewards and/or win-rates.

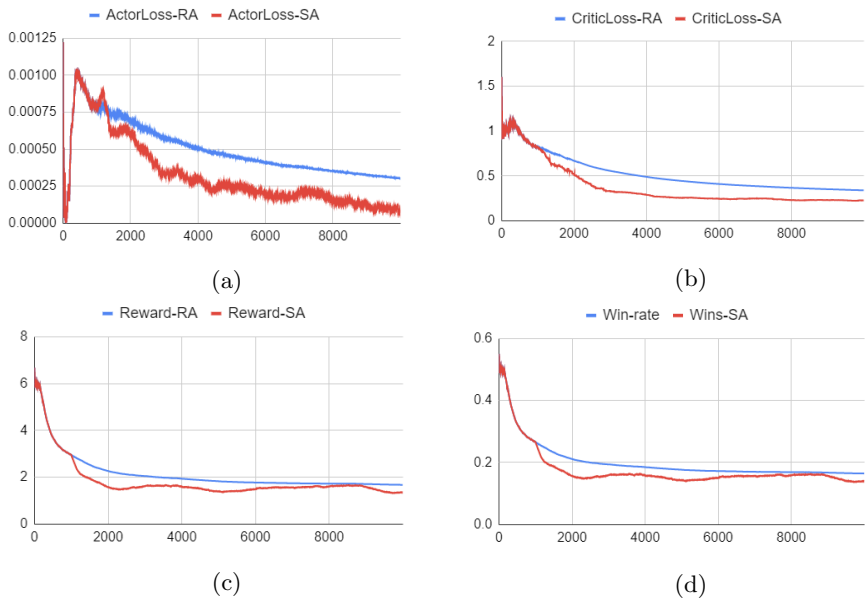


Figure 13: The running- and sliding-averages of the Actor- and Critic-losses, averaged over 20 episodes. Definitions of running- and sliding averages are given in equation **5** and **6**.

learns to avoid passing and thus spend its resources. However, the performance of the agent decreases over the course of a run, as can be seen by the decreasing reward and win graphs in **13**.

In the next sections, these results will be explained and discussed. However, some background information is needed to understand these explanations.
The average number of steps between generating resources is around $5\frac{1}{2}$. This is calculated by averaging the probabilities of the Grid-points generating a resource.

$$\frac{1}{N} \cdot \sum_{i=0}^{N} P(g_i) = \frac{1}{24} \cdot (4 \cdot \frac{1}{9} + 8 \cdot \frac{2}{9} + 8 \cdot \frac{3}{9} + 4 \cdot \frac{4}{9}) = \frac{1}{24} \cdot \frac{60}{9} \approx 0.28$$

In this equation, N is the total number of gridpoints on the board, g_i is the i-th gridpoint and $P(g_i)$ represents the probability that a resource is generated by this gridpoint. On average, a single initial village generates approximately 0.28 resources per step, which equates to approximately 3.6 steps for a single resource. There is also a $\frac{2}{3}$ chance that the dice-roll does not generate a resource at all (when rolling a 4). The total number of steps between resource-generation therefore becomes approximately 5.4. This corresponds with the experimental results.

7.1 Critical Steps

In any particular episode, the Actor-Critic agent will only have about 5 to 10 steps in total where non-passing actions are available (called 'critical steps'). This accounts only for 2-5% of the total steps. When such a critical step occurs, the Actor-Critic agent will have to wait at least approximately $5\frac{1}{2} - 16\frac{1}{2}$ more steps for that extra resource to be able to build a village instead of a street. This is because, it was found in the inspection of individual episodes that an agent will receive 1 resource approximately every $5\frac{1}{2}$ steps on average. Also, if the agent only owns 1 village on the board, there is a good chance that the agent only receives one resource type (more on that later). This results in an extra waiting time of around $16\frac{1}{2}$ steps for certain resources if they are not resources the agent can obtain without directly. This decreasing further to -0.17 when accounting for the SSR. Waiting to have enough resources to build a village instead of a street thus requires the agent to overcome the -0.17 'waiting' penalty. This is not aided by the fact that during all of those steps, there are a lot of actions with a higher probability than passing, that result in the agent building a street. If the agent has only 1 village that is only connected to 1 tile , the agent needs at least 7 resources[16] to build a new village. This will take $7 \cdot 5\frac{1}{2} \approx 40$ steps, which is a -0.4 'saving' penalty. This is a pretty hefty cost for only a reward of 1 and thus a total reward of only +0.6.

7.2 The Next Village

Furthermore, none of the previous accounted for the 2-street requirement between villages. Any player will need to build at least 1 street, or 3 streets in a worst-case scenario, to even be able to build a second village. To sketch this impact, consider a scenario in which an agent has just placed its first village and adjacent street.

Best case scenario: The agent has placed its first village in such a way that the village is connected to two tiles with different resources, preferably Wood and Brick. To build its next village, the agent will need to build a street and acquire the resources for a village. Using the same average resource-gather rate as before it can be easily calculated that the agent will build the street in 11 steps. For the village, the agent will need at least 5 resources, of which 3 to trade in for the one resource the agent cannot acquire directly from the board. This results in another $5 \cdot 5\frac{1}{2} = 27\frac{1}{2}$ needed steps. In total, in the best case[17], the agent will need $27\frac{1}{2} + 11 = 38\frac{1}{2}$ steps to build a village, which results in a final reward of +0.615.

However, now consider a worst case scenario where the agent builds its first village in such a way that it can only generate a single type of resource, as depicted

[16] One of any resource, plus six op top of that to trade in for the two other resources.

[17] This only accounts for placement, the actual resource gathering is of course random. Therefore, an average is used. In the data, there are cases of it taking 30 steps for an agent to even acquire a single resource. Also, this only considers scenarios where the agent builds exactly the right building directly after getting the resources.

by the red and black GPs in figure 14. The agent will build 2 streets, each costing 4 resources[18] (of which 3 are to trade in). This results in $2 \cdot 4 \cdot 5\frac{1}{2} = 44$ needed steps on average. The village costs 7 of the same resource, which in turn results in $7 \cdot 5\frac{1}{2} = 37.5$ steps. A total of $81\frac{1}{2}$ steps is therefore needed in the worst case. A final reward of around +0.095 is acquired.

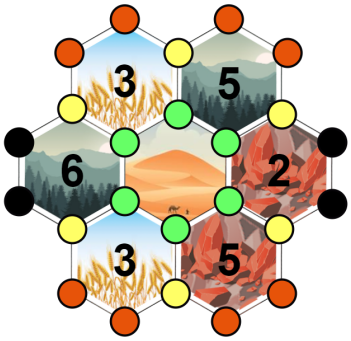


Figure 14: Visual representation of the simplified environment. The colour of the GP represents their value for a (first) village. The order from worst to best is as follows: Black → Red → Yellow → Green. Placing the first village on a black grid-point will result in a worst-case scenario, while placing it on green results in a best case scenario as described. The Green and Yellow points only differ in the fact that there are more options to expand toward other 'good' points when starting at Green than Yellow. The difference between Red and Black is the probability of the adjacent tile generating a resource, which is lower at Black (half compared to Red).

7.3 Initial Bias

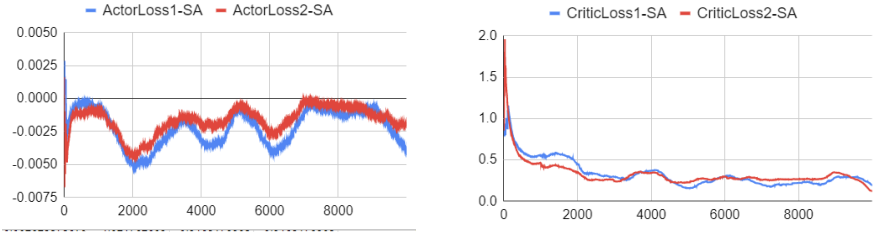
Thus, we have a best-case scenario of +0.615 and a worst case of +0.095 total reward. However, the truth does not lie perfectly in between those scenarios. The best and worse scenarios both assume that the agent will take the right action at the appropriate times. However, an agent can also choose sub-optimal actions, such as passing in a critical step. Building a street instead of a village could also be considered a sub-optimal action, depending on the current state of the board. Furthermore, the Actor-Critic agent also suffers from Initial Bias. Because the agent immediately receives a reward of 1 when placing its first village in its first step, the agent (specifically the Actor) generates a bias for a certain 'village-building-action' after a while. Whichever specific action it is is

[18]The point can be made that you could place a village next to a Grain tile, in which case you would need 6 resources to build a single street. However, in the used configuration of the board, the Grain tiles actually both have a probability of $\frac{2}{9}$ of being rolled, which would half the time needed to collect the needed resources compared a Wood or Brick tile that only have a probability of $\frac{1}{9}$ of being rolled.

purely based on chance[19], but this action will than be chosen more and more as the initial action through the whole run, until that particular action is always the initial action. This practically means that after a certain (relatively small - around 500 to 1000) amount of episodes, the agent will always choose the same initial action. In at least 50% of the runs, this results in poor placement of the first village, see **14**.

7.4 Additional Findings

In contrast to the 4-player environment used by Szita et al. [1], this 2-player environment did not give any significant benefit to the starting player. When two RandomAgent's faced off in 10K episodes, it was found that the starting player won 51% of the matches, which is within margin of error. Also, interesting behaviour occurred when two Actor-Critic agents played against each other. As can be seen in figure **15**, the losses reported by the two networks are strikingly similar, in trend as well as in peaks and troughs.



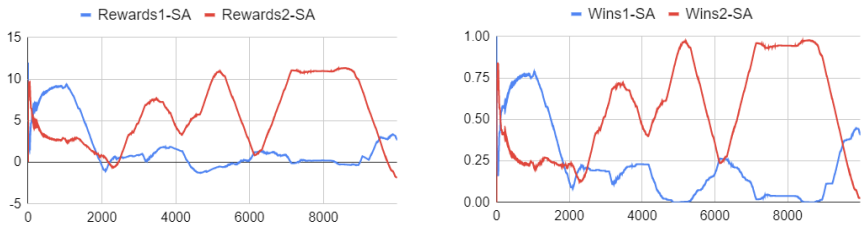
(a) (b)

Figure 15: The sliding average of the losses of two Actor-Critic agents facing off for 10K episodes. See equation **6** for definition of sliding average.

However, when looking at the rewards and win-rates of the two agents in figure **16**, huge discrepancies can be between the two. Both agents show multiple collapses in their behaviour, resorting to *only* passing during such a collapse. Looking at the episodes around which the collapses occur it can be seen that around 2000, 4000 and 6000 agent 2 seems to 'recover' while agent 1 shows the opposite, actually collapsing into losing around those episodes. This is remarkable considering that the peaks and troughs in the rewards and win-rates correspond to peaks in the Critic losses and troughs in the Actor losses. Because of time-constraints this could not be investigated further and will need to wait until further research.

Sharp readers will see that the win-rates do no always sum to 1. This is because of two reasons: First, these are sliding averages, which means that they are not a perfect representation of the win-rates, but an approximation and only useful for describing trends in the data. Second, it's also possible for an episode

[19]The weights of the network are initialized with random values, which results in random actions in the beginning.



(a) (b)

Figure 16: The sliding average of the rewards and wins of two Actor-Critic agents facing off for 10K episodes.

to result in a draw if the step-count reaches the maximum. This maximum was set to 400 steps because 99% of the episodes ended below 350 steps during all tests with Environment P.

7.5 Final Results

Inspection of the direct outputs (i.e. action probabilities) of the Actor revealed that the probability of non-passing actions peak when the AC-agent has enough resources to build something. Also, the probability of passing decreases over the course of a run, i.e. every episode the probability of passing is less overall than the previous episode(s). This means that the agent learns to make use of its resources to build something and that passing is generally not beneficial. However, because the village cost is the same as that of a street plus a Grain, the AC-agent shows impatience and builds streets almost exclusively, instead of waiting for resources needed for a village. This makes intuitive sense. If passing is learned to be associated with a negative reward (a.k.a the SSR of -0.01), then the probability of passing will of course go down. As a result, all other probabilities will go up. However, because the agent has so little opportunity to get feedback on building actual buildings and of certain biases caused by the setup of the problem, the agent will never learn within reasonable time that it can be beneficial to wait for the extra resources to build a village instead of a street.

8 Discussion and Future Work

This section will discuss the limitations that were encountered during this research. Below, critiques and areas for improvement are stated, in no particular order.

8.1 Dilution of Input

The input space in the used environment is actually quite small, but can be difficult to interpret correctly by any neural network. This is because a small portion of the inputs are very meaningful, namely the first 3 or 5 inputs (depending on the environment) that represent the current resources. The rest of the input nodes (around 150 or 400, again depending on the environment) have less of an influence on the possible actions in a given state. This distinction can take a long time too learn. A recommendation for future research is to split the agent into two completely different networks. The first network could handle the higher level decision making surrounding building. Practically it would mean that it chooses which building type to build based on the resources and (probably) certain heuristic inputs. The other network could make the lower level decisions such as choosing the position of the building based on the GP input. This GP input is combined with the input concerning the current resources in the currently used NN. This is not unlike the approach taken by Pfeiffer [2], which used heuristic high-level decision making. This approach would probably increase convergence speed because of the divide-and-conquer approach to the problem. In the overarching problem of Catan in general, a single neural network that should tackle every aspect of the game would become massive and hugely sample inefficient. One option within the Actor-Critic networks would be decentralizing the actors and keeping a centralized critic. This works in competitive and cooperative environments as shown by Lowe et al. [4], of which Catan is a healthy mix. An example concept of a network that would 'play' Catan in the proposed configuration is visualized in the figure **17** below.

8.2 Starting Small

The used network itself is also very basic and not iterated upon, which is mostly because of time constraints. For future research, starting with an even smaller network and environment and iterating from there would result in more consistent results. This would prevent the process of simplifying the environment step-by-step to cater to the agent. Furthermore, the choices for the layers were based on rules-of-thumb of AI-enthusiasts. This can be a good starting point, but shouldn't be used as final choices for the network. In extension, the part[20] shared by the Actor and Critic is also not iterated upon, which has a large impact on the performance of the network overall. The current setup assumes that the patterns observed and used for determining the policy can also be used directly to 'value' the state.

[20] In this sentence, the "part" mentioned is actually almost the whole network. See the visualisation of the network in the Actor-Critic section 4.

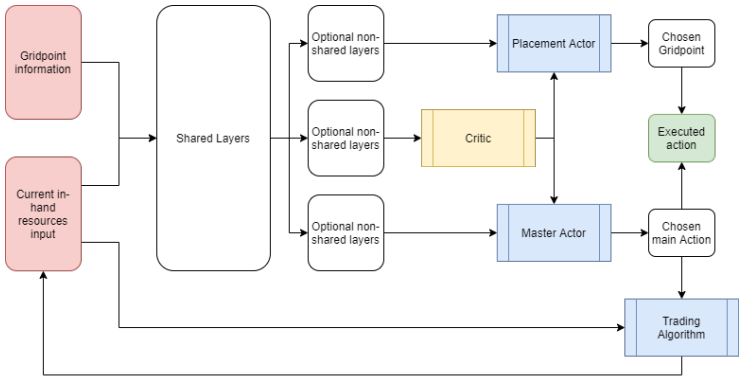


Figure 17: A diagram of how a network with multiple actors and a centralized critic could look like. In this example, the 'Main' / 'Master' actor decides which building to build or whether to pass. The 'Placement' actor decides the position of the building (if that's relevant). The outputs of these actors are then combined into a final action. The critic provides feedback for both of them with a single value estimation. A separate trading algorithm tries to trade resources such that the chosen action can be executed.

8.3 Simplification Justification

As stated in the Environment U and P sections, a justification of the simplifications of the environments is in order. For most of the simplifications holds a single overarching reason: Focus on the core of the game. In Catan, the core is to build settlements in the right positions and use resources in the right ways. Take trading between agents for example; Trading is important part of Catan but not necessary for the progression of the game. Trading only speeds up progress and adds a lot of (mathematical) complexity to the game. As Catan is made for humans, trading can be a fun way to show alliances and stir up rivalries, but not so much for an AI player. That said, having an AI that could use these underlying psychology to its advantage could be very interesting but is way beyond the scope of this research. The same reasons hold for removing the harbors and development cards. On the other side of the fence resides the Robber, which only exists to impede players progress. The Robber is essentially nothing more than a chance-based annoyance that can undermine whole strategies if given enough time. Removing it was therefore a straightforward decision. Fixing the map generation (and reducing the size in Environment P) was a decision made out of time constraint issues. A downside of fixing the map is the decreasing generalisability of the AI. Building on the first justifications, removing the cities and reducing the number of different resources does not alter the core of the game either. Five different resources is a perfect number when playing with humans, because a higher number would increase the complexity (and in extension the difficulty and length) of the game and would reduce fun. The latter also holds for reducing the number of resources, which was necessary

in Environment P to reduce training-time and cater to the smaller map size (A 2-3-2 map with 5 resources just doesn't make sense). Cities are direct replacements for villages and introduce no important key game-mechanics, at least not in the base-game version of Catan. In future research, a single player/agent environment could be considered to create a network that could learn the key game-mechanics and iterate from there.

8.4 Sparse Rewards

It is clear that this research has suffered under time constraints. This is not aided by one of the biggest drawbacks of Actor-Critic methods, sample inefficiency. In previous sections there has been touched on this subject before. In this research a deliberate choice was made to include real victory points as the direct positive feedback to the agents, instead some heuristic reward function. This however opens up problems surrounding learning, because the RL agent depends on these sparse rewards as feedback from the environment. A low sample efficiency combined with sparse rewards and sub-optimal network design can be devastating for learning. There is however an interesting way of mitigating the sparse rewards problem. An add-on for Actor-Critic called 'The Curiosity Module' (ICM for short) can be used to better arm the network for sparse reward; Proposed by Pathak, Agrawal, Efros and Darrell in 2017 [9] the ICM enhances the abilities of a standard A2C or A3C[21] network to handle sparse rewards. The games used as training and testing environments in their paper are not at all comparable with Catan but prove the potential of the ICM. For example, an agent was trained on the popular first person game *Doom*, having only the final outcome of the game (reaching a certain in-game location or object) as feedback. Promoting "Curiosity" in the agent and learning to give intrinsic rewards, rather than depending on extrinsic feedback from the environment, enabled the agent to consistently find the way to the goal. This could be extended into Catan, because of the relative long times between actual available actions, let alone rewards.

[21] A3C is an extension of A2C where multiple instances of the environment are played by different actors. These actors share a centralized critic and parameters.

9 Conclusion

The main agent in this paper, controlled by an Actor-Critic neural network, clearly showed a learning process, indicated by decreasing loss metrics. This decrease indicates that a policy / strategy had been formed. Also, analysis of individual episodes revealed that the probability of passing went down over the course of a run / episode, while probabilities of non-passing action spiked during so-called critical steps. This revealed that the agent learns spends its resources and that passing is generally not beneficial. However, the rewards and win-rates were decreasing. This was due to various biases, such as the so-called Initial Bias, and the sparsity of the rewards. The agent was greedy and preferred building streets over villages and did not show any indication of learning where to place its buildings. This behaviour was also strengthened by concessions that were made during development because of time- and resource-constraints. The latter meant that almost no iteration could be done on the network design or reward structure. However, this research has at least shown the pitfalls that come with tackling such a complex game and provided insight for future research. This includes the split of the input-space to provide a more delimited area for Actor networks to thrive in and promoting Curiosity to cope with sparse rewards.

All in all, this research has not succeeded in creating an Actor-Critic agent that could play a (simplified) digital version of Settlers of Catan on a human-like level. However, the observation that learning occurs indicates that such an agent might be possible, given state-of-the-art reinforcement learning techniques.

References

[1] Istvan Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*, volume 6048 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2009.

[2] Michael Pfeiffer. Reinforcement learning of strategies for settlers of catan. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.

[3] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[4] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, pages 6379–6390, 2017.

[5] Catan GmbH. Catan official rules. `https://www.catan.com/service/game-rules`. Accessed on 01-05-2020.

[6] PythonLessons. Advanced actor critic algorithm (a2c) with pong. `https://github.com/pythonlessons/Reinforcement_Learning/tree/master/09_Pong-v0_A2C`.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Geoff Hinton. Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, 2014. Accessed: 10-06-2020.

[9] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17, 2017.