MAd-Chess

Diogo Ramalho 86407
Diogo Faria Fernandes 86410
Rafael Pestana de Andrade 86503
Tiago Oliveira 86517

1. INTRODUCTION

Since the delivery of the first part, our project suffered a lot of alterations. We had planned to use reinforcement learning to develop our chess engine, but the dimension of states and actions was too big for conventional algorithms, such as Q-learning. We thought about using a Deep-Q network to estimate states, but due to the complexity of actions, we'd probably have to use a search algorithm to find the best action. This is the implementation of the most popular chess engines and it seemed to defeat the purpose of a multi-agent approach. What would be the purpose of each agent having its own Deep-Q network if the state is the same for each one? And why would every agent search in a different action space on its own, instead of joining all the search?
So, we decided we'd make each agent get a state as input in a Neural Network and output an action. This way, every agent is different. We also decided to change from 16 to 6 agents, grouping same types of pieces, in order to speed up training; there's now one agent for the rooks (1), bishops (2), knights (3), Queen (4), King (5), Pawns (6) and we added a special one to make a centralized decision about which agent chooses the next move, we called it *King's decision.*
In order to learn, we decided to use supervised learning, observing a better agent called Stockfish **[1]** playing against a random player. Our goal will be to study the performance of a multi-agent approach based on supervised learning against the same random player.
In this report, we first explain some chess rule simplifications. Then we describe some concrete and central implementation concepts utilized. We follow by explaining the whole learning process, from data collection to model parameter decisions. Afterwards, we explain how we observe our system's behavior. We then talk about each agent individually, detailing more about their implementation, learning model and interesting behaviors observed. Finally, we join all the agents together and observe their behavior. We finish this report by discussing the results obtained overall and suggesting further work.

2. SIMPLIFICATIONS

We removed certain chess rules in order to simplify the game. We removed two-steps in pawns, en-passant plays, castling and promotions. We also assumed we're always playing as the white team.

3. IMPLEMENTATION

We implemented the system using Python. To play chess, we used a library called python-chess **[2]**. In order to simplify it, we had to modify it in order to filter all the more complex moves we're no longer considering. To learn, we're going to use neural networks from Tensorflow **[3]** and Keras-Tuner **[4].** The learning was done in a supervised form using the popular chess engine *Stockfish* **[1].**
The part that took the longest was to actually find the system architecture so all agents could function together and code it. There are two main parts of the system: one is the abstract architecture of the agents and the other is the system simulator that unites all of them, known in our project as Agents playground, where samples are collected, and tests performed.
More precisely, the code structure is divided into the following folders: **Players**, where we have the random player, Stockfish and other useful ones; **Agents**, where we keep the agent implementations and the playground; **State representations:** where we transform a chess board into a state for our learning.
We represent the board using piece lists, each with 32 integer slots and with each index in the list being associated with a specific piece; the first half concerns the white pieces and the rest concerns the black ones. The general structure for one team is as follows:

[R1, N1, B1, Q, K, B2, N2, R2, 8*P]

R- rooks, **N** - knights, **B** -bishops, **Q**-queen, **K**-king, **P**-Pawns. **[num] -** number of the piece.
We'll spare the details about how we distinguish between the two rooks, knights, bishops, and eight pawns; it suffices to say that it uses the order of the FEN **[4]** description chess board. Each slot stores board positions from 0 to 64. 0 means the piece is dead. 1-64 are the tiles, ordered from left to right, starting at square A1 and ending in H8. The following is how the initial board in a chess game is represented:
[1, 2 , 3 , 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 57, 58, 59, 60, 61, 62, 63, 64, 49, 50, 51, 52, 53, 54, 55, 56]

4. LEARNING

As said before, every agent has its own multi-label classification model and learns independently from the others. Each uses a dense neural network with few hidden layers, where the input is the state represented as a list and output is an action.

4.1. Data collection

Our approach to learning is conceptually very close to an *on-line* learning. The idea is to play games and, every time Stockfish plays a piece, the agent which is in charge of that piece, updates his knowledge. However, doing this strictly in the code is extremely inefficient, so we play the games first, store them in order and then learn from it. We maintain the order as it is required for online learning.

We used 150 thousand games to learn. In total, we collected 2.659.101 plays, but they're not evenly distributed by all agents: each agent only has the plays where its pieces make a move. The Knights had around 900.000 plays, while the King had only around 45000, being the least played piece. The King's decision - the special agent uses all these samples (but instead of using a move as output, uses the piece which played).

4.2. Hyper Parametrization

In order to learn such a complex game, we'd probably need considerably bigger networks and a vaster amount of games, maybe to play a few millions. Unfortunately, this is a problem we can't solve, and we had to use relatively small networks. Yet, we thought it would be interesting to try to use Keras-Tuner to choose some parameters for us, especially since we don't have much experience with NNs. In this project, it was given the freedom for each one of us developing the agents, to hyper parametrize the parameters they were interested in observes- we won't describe those variations in the report, we refer to the code of each agent for such details - but the main parameters changed were the learning rate of the optimizer, the number of nodes per layer and the number of layers.

We use validation sets (10%) to hyper parametrize and a test set (10%) to check the final accuracy. While ideally training and validation sets have no common data, this tends to happen since frequent boards, such as the initial, are present in every game. However, since we evaluate games rather than individual boards this has no relevant impact and it's done this way to maintain the aforementioned concept of *on-line* learning. The *real* test for the agent's learning has to be done by subjective evaluation.

5. TEST

Fitting errors are good indicators, but they are insufficient in order to assert how good our agents have become at playing chess. Therefore, we created four main tests (implemented in AgentsPlayground), where our agents play against a random agent:

testOneAgentAgainstRandom: Tests any chosen agent (except King's decision). Put it in a team which uses Stockfish to decide whether our agent should play. If not, a move of any other piece is chosen randomly.

testKingsDecision: It plays the normal game between stockfish vs random, but it allows us to see if and when King's decision is guessing correctly.

testAllAgentsAgainstRandom: Tests all our agents except King's decision (only the pieces) forming a team.. Stockfish decides which agent to play.

testAllAgentsWKingsDecision: Tests all our agents forming a team. King's decision chooses which agent to play.

For all of these tests, if an agent suggests an illegal move, we play the second-best suggested move and so forth. If no suggested play is legal, then a random move in the board is chosen (although in practice there's usually always a legal move).

6. AGENTS

We are first going to describe each agent and study it using **testOneAgentAgainstRandom** (or **testKingsDecision**, if the agent is the King's decision). We'll start by talking about the connect between NN outputs and moves, then explain the parameters of each learning model and, lastly, we'll try to understand a little bit of the behaviour of each piece and make some observations. It's important to notice that we can't possibly describe everything we observe, so we limit that part to the observations that seem interesting and useful.

6.1. King's Decision

6.1.1. Output representation

The output of this agent's network has 6 output units, one for each agent. Starting from 0 until 5, the output units correspond to Rooks, Knights, Bishops, Queen, King, and Pawns, respectively.

6.1.2. Learning model

This is the agent that connects the agents into a multi-agent system. Conceptually, this can be thought of as the same agent as the king (so it has representation on the board), but in practice it's a completely different one, so we'll treat it in this report as such. This is the agent that decides which agent moves next. It is also one of the agents with the most promising results in classification, hitting 58% in the final test set, and trained using 2.6 million samples, being vastly more trained than all the others.

Input	Hidden 1	Hidden 2	Hidden 3	Hidden 4	Output
32	480	64	128	128	6

In order to understand what our agent is capable of doing, we will look at its decisions when it is playing against a random player. Note that it will be Stockfish making the concrete moves, so the game is in the same context as the ones where samples were generated.

It guessed the 50% as it was roughly expected (out of 100 games observed).

6.1.3. Behavior

Stockfish seems to have a preference over playing pawns, knights and the queen over the others. Our agent does the same but it exaggerates it, a lot of times suggesting one of the former popular pieces, failing to realize it should have chosen one of the least popular ones. For example: our agent choses knights, when the play chosen by stockfish would be to move **rook h2h7** (Fig.1).

Figure 1

Even though our agent might lack initiative to choose these rarer pieces, it does seem to be able to recognize them when the moves are obvious. For instance, in Fig.2, the bishop was not so obvious to be played, so our agent chose knights. Stockfish played **bishops c4f7**, instead. After the opponent's play, there was an obvious opportunity to capture the rook with the bishop, making our agent break its pattern of choosing queen, knights and pawns, to choose bishop and allow this "obvious" conquer.

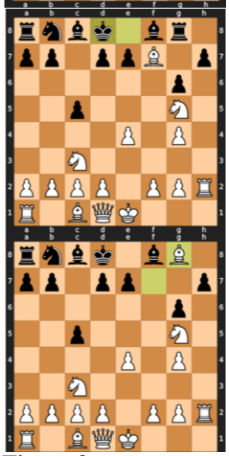

Figure 2

There are many interesting behaviors in this agent, but what seems to be happening is that our agent does choose the good plays at 1-step distance into the future, but it seems to fail a lot in any move that is part of a larger play at the game. A very clear example of that, starting with board of Fig.3 the queen is moved by Stockfish **d1d3**; our agents chose pawns. Then, Stockfish chose queen again to move diagonally **d3h7**, while our agent chose pawns again. Only when there was an obvious and very valuable for the queen, killing the rook in **g8**, our agent chose to move the queen.

Figure 3

Our agent, as inferred, does seem to make moves based on killing, but completely misses checkmates (maybe because checkmate is actually quite complex to assess). One example of this, it should have chosen rook so it could checkmate moving **h1h8**, but it chose pawns in Fig.4

Figure 4

To summarize, our agent does good 1-step choices and guesses the right ones when it comes to killing pieces. When there isn't an obvious play, it chooses one of the most popular pieces (queen, knight, pawns).

6.2. Knights/Horses

6.2.1. Output representation

According to the rules of chess, a knight can move to 8 different positions at most; and therefore, this agent's neural network features 16 outputs, 8 for each knight. The neural network output association with plays is given by the order of the figure for the first knight. For the second knight you had 8 to those values. So, for a given state, the output 15 of the learning model means moving the second knight 2 steps forward and one to the left.

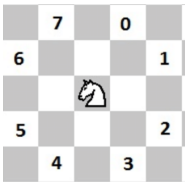

Figure 5

6.2.2. Learning Model

The Knights got a bit over 50% accuracy when fitting with our test set. When joining this agent with the testOneAgentAgainstRandom team described before (and averaging over 100 games) we hit 28% of the plays stockfish would have played. Which is not such a bad score, considering that the board games we trained were developed using stockfish vs random, so we expect these new boards to be considerably different. We're training the knights with 855532 samples and using the following NN structure (defined by hyper parameterization)

Input	Hidden 1	Hidden 2	Output
32	256	352	16

6.2.3. Behavior

If we look at our agent in overview, there are certain traits that are interesting to notice. StockFish almost always plays the horses to the middle (to **f3** or **c3**) in the first plays of the game (Fig.6); our agent does the same all the time. This might explain why the accuracy is so high. If we look at moves that are bad, it usually is about advancing towards the center, which is

Figure 6

a usual "general" guideline of chess. When Stockfish played these moves, it had set up other pieces to protect the knights or set up piece *trade-offs*; our agent doesn't seem to understand this connection. The knights seem to have learned the distinction between dead and alive knights, thus not suggesting plays for the dead one. The only time that might happen is when the alive knight is surrounded by many illegal movements and fails at classifying a couple legal ones. In Fig.7, our knight doesn't have many legal moves allowed, so it tried 3 illegal movements and then suggested a movement for a dead knight at the 4th try. But in normal circumstances, it never does.

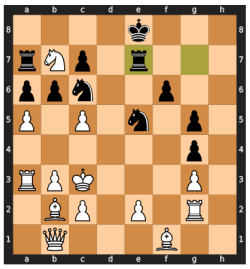

Figure 7

We observed two more relevant behaviors: the knights don't seem to understand the concept of checking. A lot of times when Stockfish would have put the king in check, our knights choose other moves, usually bad ones. When it comes to eating pieces, it's complicated to assess. Sometimes the plays are right on spot with Stockfish, other times are completely off.
The knights alone are not enough to win the game. There's no "closure" and the game usually goes on for more than 200 rounds, when we interrupt the game.

6.3. Bishop

6.3.1. Output representation

The bishop can move in the 4 diagonals and a maximum of 7 squares per diagonal, that gives a total of 28 moves per bishop. As this agent controls 2 bishops, there are a total of 56 units. The output between 0 and 55 is given by the expression **28*bishopNum + 7*val + dist -1**, such that **val** is a value between 0 and 3 telling the direction (NE,NW,SE,SW respectively), **dist** is a value between 1 and 7 representing the distance, and **bishopNum**, a binary value telling which bishop is playing.

6.3.2. Learning model

When testing against our testing set it got 15.86% of accuracy, but when we joined the Bishop with the random player (and averaging over 100 games) it got 13.63% of the moves Stockfish would have made.
It used a model with 2 hidden layers with 1056 ReLU activation units, each.

Input	Hidden 1	Hidden 2	Output
32	1056	1056	56

6.3.3. Behavior

In Fig.8, the bishop was in **c6** and there are 2 smart options. Either the selected bishop captures the rook in **a7** or the other bishop captures the queen with **f1h3** (the move Stockfish chose). However, it chooses to move the selected bishop to **d6**, not capturing any piece, but in a position where it can be captured by a black pawn, showing non-intelligent behavior.

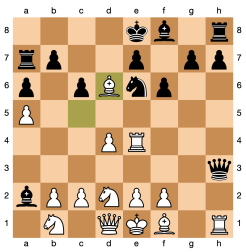

Figure 8

In all the observations that were made the agent only captures the opponents if they are at 1 square in distance. In Fig.8 the rook in **a7** was not captured and was at **square distance 2** and so was the queen from the other bishop.
Also, in Fig.9 the bishop should have captured the Queen in d3 in the first attempt like Stockfish suggested but decided to move only 1 step closer and capture it in the next attempt, when distance was 1.

Figure 9

In Fig. 10, we can see the bishop checking the King at the first attempt, just as Stockfish would have played. On more occasions, the bishop tries to check the king. However, it never seems to be part of a team plan, and so it never results in a checkmate.

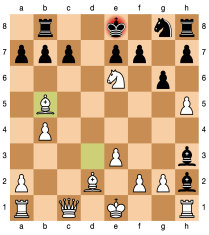

Figure 10

6.4. King

6.4.1. Output representation

We can think of the king's movement as 8 different directions with a **value** for each direction (0-7). The agent can move a maximum of **1** square in each of those directions. The encoding function returns that **value** (between 0 and 7) which encodes the 8 possible moves for the king.

6.4.2. Learning model

When testing against our testing set it got 64.12% of accuracy, but when we joined the King with the random player in **testOneAgentAgainstRandom** (and averaging over 100 games) it got 35% of the moves Stockfish would have made.
The high accuracy in the testing set may be explained by the fact that in Stockfish strategy the king is seldom asked to defend itself alone, it's a team effort. In **testOneAgentAgainstRandom**, the lack of team strategy leads the king to be in more complicated set-ups, explaining the drop to 35% and the exaggerated attempts of illegal moves.

Input	Hidden 1	Output
32	110	8

6.4.3. *Behavior*

First of all, the king often tries many illegal moves before choosing a legal one. When the King is put in check, it is able to go away usually after some illegal attempts. It is the least used piece and it is usually used when it is put in check and it has to run away, like in Fig.11.

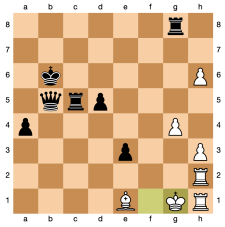

Figure 11

6.5. Pawns

6.5.1. *Output representation*

The representation is relatively straightforward: Each pawn has 3 moves: advancing forward and capturing in the forward diagonals, resulting in 24 possible actions; for instance, output of 0 means go forward, 1 is left capture, 2 is right capture for the first pawn.

6.5.2. *Learning model*

Pawns obtained 28.58% test accuracy using only 2 hidden layers with 1024 and 64 sigmoid activation units, respectively; and in trials against random player under the direction of Stockfish got 21.5%.

Input	Hidden 1	Hidden 2	Output
32	1024	64	24

6.5.3. *Behavior*

Despite the low accuracy, some patterns emerged. First, it tries to perform captures by moving in diagonal, even when it is not a legal move. Also, it tries to move the 2 outermost pieces before advancing the center as a block. For example, in Fig.12 is a staged scenario against only random pawns. The first moves we see a quick movement from the laterals. Then, it has two good moves, resulting in two captures of black pawns in ranks **a** and **b**, due to the fact that the network is biased for captures for the outermost pawns, even when they are not legal. This duel ends in a convoluted way. The pawns do not seem to possess great spatial awareness towards enemy pieces, and they make blunders in what are simple situations (for us). In another of these random duels (Fig.13), we arrived at the following situation: when the black pawn captured the white pawn at **e3**,the natural move would be a counter-capture; instead, the pawns decided to move from **g2** to **g3**, which was also not a good move because it removed support from the pawn at **f3**. Also, this situation lasts for 9 moves, when the black pawns finally captures pawn **d3** and in the process claims a check.

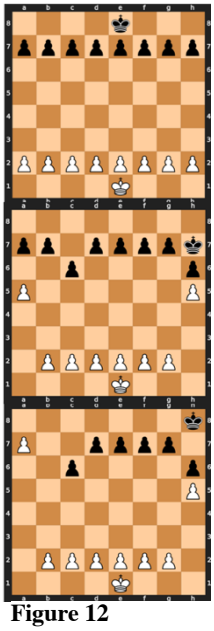

Figure 12

Figure 13

6.6. Rooks

6.6.1. *Output representation*

The representation of the rook movement consists in 14 possible actions per rook, that represent the 7 horizontal and 7 vertical different squares the rook could move to if the board was empty; as this agent controls 2 rooks, the output layer has 28 units. To make an horizontal move for the leftmost possibility, the output is 0; for the rightmost, 6; vertical moves vary between 7 (the bottom-most possibility) to 13 (the top-most); for the second rook the logic repeats, by adding 14.

6.6.2. *Learning model*

Hypertuning rooks landed them with a network with a single hidden layer with 64 sigmoid units (chosen by hyper-parameterizing and random trials), which is clearly too few, but it was able to get 28.5% accuracy on our test set; but when confronting random player, it only got 11.99%.

Input	Hidden	Output
32	64	28

6.6.3. *Behavior*

Figure 14

This resulted in simplistic rooks that have a behavior that can be summarized as go forward with the leftmost rook (and if impossible, go the leftmost possible position), which in our network actually corresponds to always strongly activating one of the neurons. By looking at the data distribution, all the datasets (train, validation and test set) featured 14% of the labels of the same class, and all the other outputs were in the single-digit percentage or lower; and this is the reason why more complex networks did not improve performance.

As a matter of fact, in general it is good to advance the rooks the most you can and make large movements, but this agent lacks attention to the board state; 36 moves later, it still is at the corner.

6.7. Queen

6.7.1. Output representation

We can think of the queen's movement as 8 different directions, like a wind rose, with a **value** for each direction (0-7). The agent can move a maximum of **7** squares in each of those directions (**distance**: 1 to 7). The uciToY function returns a value between 0 and 55 which is calculated as such: **7 * value + distance - 1**. This encodes the 56 possible moves for the queen.

6.7.2. Learning model

When testing against our testing set it got 16.99% of accuracy, but when we joined the Queen with the random player (and averaging over 100 games) it got 13.88% of the moves Stockfish would have made. It used a model with 2 hidden layers with 160 and 480 ReLU activation units for the first and second layer, respectively.

Input	Hidden 1	Hidden 2	Output
32	160	480	56

6.7.3. Behavior

It can be seen that the queen attempts to check the adversary king as often as possible, ignoring better possible moves. Also, it seems to prefer taking enemy pieces only when they are one-square away (just like the bishop). We can assume that this agent only thinks of one move at the time without considering the bigger picture.

In Fig. 15 we can see that the queen piece is able to safely check the adversary king piece despite not being the ideal move; it should have moved to **c8** to have better options in posterior plays.

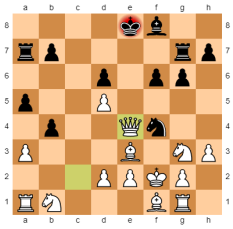

Figure 15

In Fig.16, we see that the queen goes again for the check but this time it puts itself in danger as it can easily be taken by a pawn; instead it could've just taken the knight and played safe.

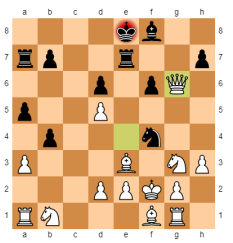

Figure 16

In Fig.17, we see the queen, again, managing to check the king, but this time it ends up being taken by the rook in the following play; a better play should've been either taking the rook or the bishop, or simply retreating to a safer position.

In Fig.18, it should've moved to **g7** in order to simultaneously check the king as well as threaten the knight, the rook and the pawn, but instead it puts itself in danger of being taken by the king.

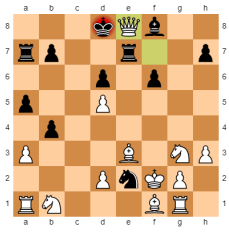

Figure 17

In Fig.19 (left) and (right), in both situations the adversary queen could've been taken but our agent decides to avoid it, leading to it being taken in the following play by the other queen.

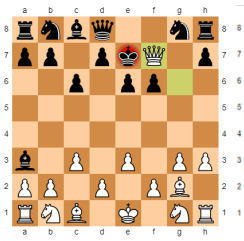

Figure 18

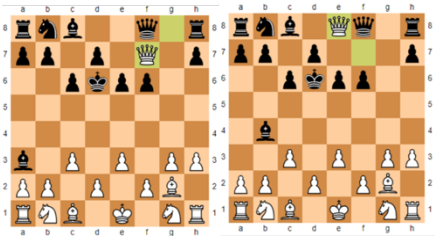

Figure 19

6.8. Accuracy summary

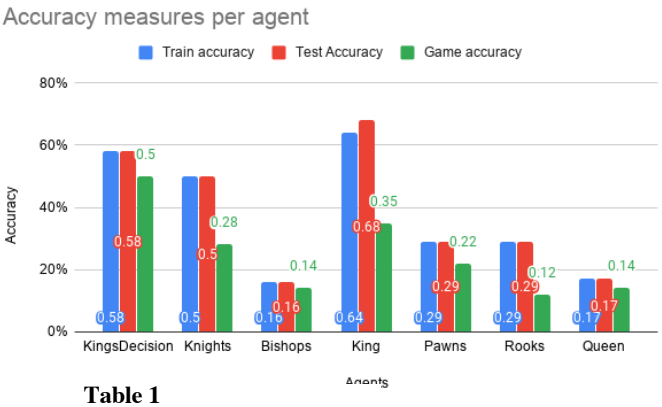

Table 1

The accuracy in the training and testing sets is very similar for all Agents, which indicates that the neural networks did not significantly overfit our dataset. However there is a significative difference between the performance on the classification test and the ability to play the game, measured with the percentage of times our agents chose the same actions as Stockfish.

7. AGENT SYSTEM

Now that we've studied our agents independently, it's time to study them all together. We're going use the following tests here **testAllAgentsAgainstRandom** and **testAllAgentsWKingsDecision.**

7.1. testAllAgentsAgainstRandom

The games usually start with a similar pattern as Stockfish. The two knights are played to the center and

then the rightmost pawn advances as much as it can. In one instance, this pawn takes the piece on the left, forcing the random player to either choose to lose the knight in **h6** or the bishop in **a3**. Their knight backs up to **g8** and a better play is performed by our agent. It captures the pawn at **h7**, now forcing the same choice again, either the knight or the bishop and, if the rook tries to kill our pawn, our rook will take it. So far, our agents seem to be honoring stockfish's plan.

Figure 20

However, as the game advances, when their knight plays **g8h6** (Fig.20), Stockfish chooses our rook to play with the intent to kill the knight, but our rook agent fails to perform that action and wastes an opportunity by playing to **h3.** This is, in a nutshell, what happens in every game. The first couple of plays seem to go well and according to stockfish's plan, but eventually one of our agents messes up an obvious play and the whole system stops showing smart decisions. Our team gets to do between 5 and 7 good plays before ruining it.

7.2. testAllAgentsWKingsDecision

Just like in **testAllAgentsAgainstRandom**, the games usually start with a similar pattern to Stockfish, like described in the previous subsection.

When we advanced in the game, sometimes strange behaviors were observed. For example, the horses, rooks and bishops would loop over the same 2 actions and would keep the chosen ones by KingDecision until either the king was in check or the piece was captured. This can be seen in Fig.21 where the horse repeats this pattern for around 50 iterations.

It was interesting to see that, even though the agents are independently deciding the move, KingDecision still chose around 42% of the times (averaging 100 games) the same agent that Stockfish did, which is very close to the 50% observed when Stockfish was the one making the decisions. So, the chaos originated by the agent's decisions in the board does not influence a lot KingDecision ability to select the agent correctly. Also, when KingDecision chooses the same agent as Stockfish, the chosen agent guesses around 20% of the times the same move as Stockfish would. This means that our final system given by the KingDecision deciding the agent and by the agents deciding the move has a final mean accuracy around 8.4% (20% of 42%), when compared to Stockfish. So, as we can observe, even if the king's decision is working reasonably well, the malfunction of the other agent's creates a bottleneck in the performance of the system. One agent alone can't pull the team back up; they depend on each other.

Figure 21

7.3. Final Considerations

Every agent that controls more than one piece did understand the link between that piece's death and its plays, never suggesting plays for dead pieces.

Some basic, simplistic patterns were found in every agent. Usually, the longer the game, the more repetitive the patterns become.

Some pieces do seem to be able to target the king, especially in early parts of the game, sometimes recklessly, ending up getting eaten.

The games take, in general, an exaggerated amount of turns, usually ending in draw (which is possible according to the rules), or checkmates without relevance.

The pieces can't coordinate themselves to achieve checkmate; there's no coordination between them in order to lead to rational plays.

8. DISCUSSION

In general, our agents did not learn how to play the game or strategize. The game and *concepts* seem to be too complex for our model. Our agents perform better in the beginning because this is when the boards are closer to the samples they learned. Patterns were indeed learned, our agents move very distinguishable from random plays, but the patterns were simplistic and repetitive. As a matter of fact, the more we advanced in the game, the more the board diverged from the learning samples and, as a consequence, the more our agents would take refuge in these loops and small routines they had learned.

Concepts like being under attack, checkmate, defending pieces and exchange of pieces are way too complex for our models. The only pattern every agent seemed to have learned was the meaning of 0 in the state (in agents that control two pieces). Agents don't suggest plays for dead pieces, only if a couple plays have been tried, but were illegal.

One pattern of capturing that some agents understood was the 1-step-square kill move (described in some agents before), where the agent would only kill pieces if they were 1 square distant from them, even if they could kill them from a higher distance. This reveals a lack of understanding about the meaning of the numbers passed in our state. There's no notion of space and mapping of positions in our models. Maybe bitmaps and convolutional networks could have been used to teach the agent about the concept of board and relative positioning, but we would, most likely, still need more

than just a couple layers and considerably more samples. Something else worth noticing about the accuracy while training is the correlation between the complexity of the model and its performance. The knights and king are the ones with the least amount of possible moves and are the ones who perform the best. Pawns are the second ones with least number of outputs and were also the next in line in terms of accuracy in the test set. The Bishops and Queens have both 56 output states in their Neural performed considerably better in the dataset than these. We can't conclude with certainty that it is the number of outputs that led to that result, so we'll just notice that observation and leave it here.

If we had more time to do the project and more resources, based on these results, we'd extend the size of our networks and the size of our samples. Playing 150 thousand games seems a lot to a human being, but it probably isn't to a machine when faced with such a complex game. We'd also try the mentioned convolutional NN approach.
Lastly, something we're curious about is if what we did here could speed up a Reinforcement Learning approach. When we changed from RL to Supervised learning in our approach, it was due to fearing that when starting with 0 knowledge about the game, it would take far too long until the agent could give *his first steps.* But what if we started with this current knowledge and small patterns we obtained? Could Reinforcement learning start from these, correct and extend them into more complex behaviors? Would it be faster to extend our patterns or create them from scratch?

9. CONCLUSION

We can confidently say we underestimated completely the complexity of our task. We now have an idea of why chess is the most popular game to study in AI and why so much engineering and research has been done about it. However, those challenges were welcomed and made us learn quite a bit! First of all, chess is such a complex scenario, where the state and action space is so big, that we had to figure out a way to address this problem, in our case using Neural Networks, which we didn't have much experience with. It was also interesting to realize how simplistic approaches can be problematic when facing these complex problems. Secondly, unlike many problems we had solved so far, learning and training even the most simplistic models is a very time consuming task, especially if we want to use hyper parametrization. Thirdly, one of the biggest problems was the fact that we don't know what's *going on* inside a neural network. We had to try to decipher patterns by looking at the system playing, which makes it very difficult to understand our problems and correct them. And finally, it was interesting to watch a decent - at least when compared to the other agents - performance, King's decision, be completely irrelevant to the networks and, not only their performance was the worst, but in the experiments, it was visible that some of the more complicated plays that are natural to do with these, simply weren't performed. But most interestingly is the Rooks. The rooks have, in theory, the same complexity as the bishops and queen - 56 possible moves. However, we found a way to encode these moves into 28 output states in its neural network and, even though the rooks only had half the samples as the bishops to train, they performance of the team as a whole when everyone else is doing poorly. In other words, one agent alone can't *save the day!*

REFERENCES

[1] https://stockfishchess.org/
[2] https://python-chess.readthedocs.io/en/latest/
[3] https://www.tensorflow.org/
[4]https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation