Lab Report: Playing the Kore Game with Deep Reinforcement Learning

Nils Eberhardt Hasham Hussain Ramil Sabirov Daniel Tebart
RWTH Aachen University, Germany
{Nils.Eberhardt, Hasham.Hussain, Ramil.Sabirov, Daniel.Tebart}@rwth-aachen.de

Abstract

In this paper we describe our work on the recent Kaggle challenge featuring the game Kore 2022. We designed multiple agents which learned to play the game in a reinforcement learning setting. For that we introduced high-level actions to reduce the large action space of the game, adapted the learning algorithms to output several actions (that is one per shipyard) per state, implemented state-of-the-art deep learning models such as the multi-modal transformer and invented rewards to find a balance between winning and a certain density to train.

Our best result is reached with a version of Proximal Policy Optimization (PPO) and manages to win roughly 80% of the games against the strongest rule-based agent provided by the kaggle environment.

1. Introduction

The underlying task our work tries to address is the recent Kaggle Challenge "Kore 2022"[17]. Generally speaking "Kore 2022" is a 2-player game where both players start with a shipyard on a 21x21 map with the objective to mine as much as possible of a mineral named "Kore" which is distributed across the entire map. This is done by building ships (by spending Kore) at one's own shipyard and then send the ships in packs (as fleets) on a route through the map. Every time the fleet enters a square of the map it collects Kore proportional to the size of the fleet and the amount of Kore in the cell. Once the fleet returns to an allied shipyard the collected Kore is transferred to the player's account and can be used to build further ships. For a graphical presentation of a game state see Figure 1.

If it happens that the fleets of the two players enter the same square or one fleet enters an enemy shipyard, a "battle" happens and the bigger fleet remains victorious collecting the Kore cargo of the defeated fleet. In this way it is possible to "eliminate" the opponent that is by destroying all of his fleets and capturing all of his shipyards.

The game is won by either eliminating the opponent or, if this does not happen, by having more Kore in one's account than the opponent after 400 time steps.

Action Space During every step of the game every shipyard can execute exactly one action. This action is either a building action $build(n)$ or a launching action $launch(n, f)$, where n indicates the number of ships to be build (or to be launched respectively) and $f \in \{N, E, S, W, C, 0, \ldots, 9\}^l$ is a so called "flight plan" describing the route of the fleet. The flight plans are interpreted in the following way: the letters N, W, S, E describe a change of direction (with additional step) to North, West, South, East respectively. The numbers describe how many steps to continue in the current direction. The letter C initiates a creation of a new shipyard if the fleet size has a large enough size. The routes automatically end at the shipyards and if the flight plan is not leading to any shipyard, the fleet continues to idle in the last assigned direction. The length of the flight plan l is bounded by the size of the launched fleet. That is, a larger fleet allows for a longer flight plan. It is important to note that the flight plan needs to be specified at launch time and cannot be changed after that.

The structure of the paper is roughly as follows: After mentioning related work in section 2 we dive into the core of our work featuring our approach 3. In that section we explain the rewards in subsection 3.1, the adaption of the learning algorithms to the dynamically changing amount of shipyards and therefore action outputs in subsection and the reduction of the action space through high-level actions in subsection 3.2. Then after a short introduction of the learning algorithms we used in subsection 3.3 we cover our design of the state representation of the game in subsection 3.4 and give a description of the deep learning models we used in subsection 3.5. Finally, in section 4 we summarize our results.

2. Related Work

Model free Learning The combination of model free learning and deep neural networks has gained much attention in solving video game environments since Mnih *et al.*[19] showed that human-level performance can be achieved on some games from just training on images. Numerous further work built upon this idea, among those prior-

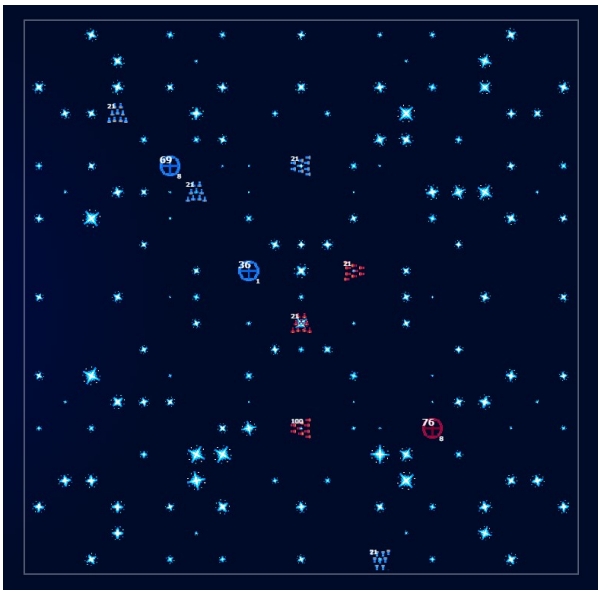


Figure 1: One exemplary state of the game. The blue player has two shipyards with respectively 69 and 36 ships, the red player one shipyard with 76 ships. Both players have three fleets sent out. The light-blue stars on the map indicate the Kore position, where the brightness is proportional to the amount of Kore in the cell.

itized experience replay to increase sample efficacy [21] and double dqn to reduce estimation bias [27]. Hessel *et al.*[11] showed that combining extensions lead to further improving agents. Apart from value learning, [32] showed that complex 3d games can be learned by policy optimization based agents. Especially actor critic agents with shared experience replay can compete with other state of the art agents in atari games [22].

Model based learning Many model based approaches are focused on control tasks [31, 12, 29] or learning board games [26]. Recently, motivated by the benefits of planning, model based methods were also applied successfully on atari games. Kaiser *et al.* [15] used a policy optimization agent on video data to predict the next incoming frame from the previous four frames in pixel space. *MuZero* [23] learns to play by combining a learned model with time-consuming Monte-Carlo Tree Search. Other work was dedicated to developing models that can be trained even on a single gpu in a reasonable amount of time [8].

3. Approach

To let any agent learn to win at a game of Kore, the game itself has to be transformed into a markov decision process (S, A, P, R). The transition probabilities P are given by the rules of the game and the opponent. We chose to train against a balanced agent provided by the kaggle competition [1]. The state space S, the action space A and the reward function R have to be manually specified. Therefore we first introduce our choices of R, next A and then S. At each step we also discuss problems that were encountered. Finally we specify the models as well as the agents which were trained on this markov decision process.

3.1. Rewards

The definition of the reward function plays a vital role in any reinforcement learning task. The reward function is an incentive mechanism that allows us to shape the behaviour of the agent to achieve our goal. An RL agent takes the actions that lead to the maximization of the total reward. Here we briefly go through the five reward functions that we have devised.

3.1.1 Win-Lose Reward

Mathematically the win reward is defined as follows:

$$Reward(state, action) = \begin{cases} 1, & \text{if win} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The agent only gets a reward if two conditions are met: the agent is at a terminal state and the last action leads to winning the game. For every other case, the agent gets a zero reward. Win-Lose reward exactly represents the goal of our task: to win the game.

The obvious drawback of this kind of reward is its sparsity — the reward is zero for most of the steps. An episode can last for up to 400 steps which leaves the agent without any reward for all the intermediate steps. The situation can be particularly bad at the start of training as the agent is unaware of the environment and loses a lot of games leading to no reward for many episodes. It is because of these reasons that training an agent with this becomes infeasible for our case due to the limited resources and long duration of a game.

3.1.2 Advantage Reward

To overcome the sparseness of the win reward, we can attempt to engineer dense rewards using domain knowledge. One such dense reward is the Advantage reward.

To explain this reward, we first need to define the advantage of a state. The advantage is a function of the state and can be defined for each player as a weighted sum of the resources (kore, ships etc). Advantage reward is a ratio of the advantage of the current player to the total advantage available at a state. This incentivises the accumulation of resources which in most cases leads to winning the game.

An obvious downside of this reward is that it does not represent our actual learning goal (win the game), but only

serves as a proxy for the real goal. The proxy rewards sometimes lead to increasing rewards but decreasing win rates during training (eg, when an agent can learns a hack to maximize the reward without winning).

3.1.3 Competitive Kore-Delta Reward

While the Advantage reward takes into account several factors to make the reward more dense it also inevitably includes a certain amount of bias in form of weighting the different resources in a specific way. The idea of the Competitive Kore-Delta reward is to decrease weighting bias by focusing just on one of those resources, namely the Kore. Hereby, for every time step the reward is defined as the difference of Kore successfully collected by oneself and the Kore successfully collected by the opponent. Empirically this reward shows a good correlation to the Win Reward, that is: the total episode reward is positive in most cases if the agent wins but negative if it loses. However, the reward is not bounded. This poses an issue as the agent generally tries to maximize the expectation of the reward and can therefore learn a strategy which wins rarely but with a very high total reward, while the negative reward for the losses has a relatively small magnitude.

3.1.4 Neural Network as Representation of Reward

Instead of hand coding a reward function, we can also let a neural network learn the "states that lead to winning" and use it as a reward signal. We train this neural network in a supervised manner with over 22,500 labeled states. Tensors representing the state of a game are saved at a regular interval during a typical game run along with the winning agent. The neural network is then trained with the objective to predict the winning probability for each player (Figure 2). The winning probability of our agent is then used as a part of the reward function.

When trained against the balanced agent with the PPO algorithm, we get a win rate of about 73%. This further supports the hypothesis that a NN can learn the "winning states" and can output the reward signal that would lead to winning.

3.1.5 Annealing Reward

In the annealing reward function, we can start with a dense reward (eg, Advantage reward) function and then gradually move towards a sparse reward (eg, Win reward).

The equation governing this transition is given below:

$$Reward(R_1, R_2) = (e^{\frac{-t}{\tau}})\, R_1 + (1 - e^{\frac{-t}{\tau}})\, R_2 \quad (2)$$

where R_1 is the starting reward and R_2 is the final reward. τ is the temperature that controls the rate of annealing, t is the number of steps passed since the training has begun.

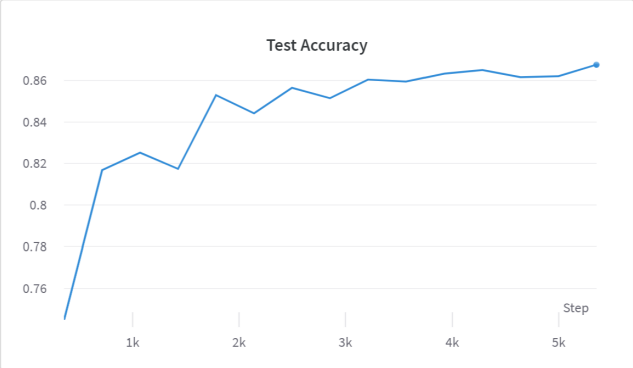


Figure 2: After training, the NN is able to correctly predict the winner 85.4% of the time when tested on the test dataset.

3.1.6 Further consideration regarding Rewards

Penalizing Invalid actions The reward can also be used to incentivise the agent to learn which actions are illegal actions (see 3.2.4) by adding a penalty in form of a negative reward, whenever the agent attempts to perform an illegal action.

We have found that while this method can work reasonably well, one has to be very careful about the choice of the penalty value. If, compared to the reward, the penalty is too big, then the agent's action choices become very conservative (e.g. not expanding as this has a high risk of being illegal) and training slows down. If on the other hand the penalty is too small, it becomes insignificant and the desired effect of avoiding illegal actions is not achieved.

Generally speaking, the masking described in 3.2.4 led to better results.

Trophy Rewards One problem with cumulative rewards as the Advantage reward 3.1.2 or the Competitive Kore-Delta reward 3.1.3 is that the agent is incentivised to drag out the game and might even prefer a long lose over a fast win. This can be addressed by taking one step in the direction of the Win-Lose Reward by adding a "trophy" reward at the end of the game. This trophy accounts for the early finish of the game by for example rewarding a fast win or penalizing a fast loss.

3.2. Transforming the action space

The action space contains a number of different problematic properties that needs to be reformulated in order to obtain a suitable reinforcement learning environment. The game allows each existing shipyard to take an action at each time step, thus resulting in a varying number of allowed actions throughout a game. Each shipyard takes an action for

a varying number of ships and launching ships with a flight plan of varying size, adding further dimensions of variability. Additionally the number of possible flight plans grows exponentially in the size of the flight plan. At last all actions besides waiting are constrained, requiring an agent to adhere to the rules at any time step to be as efficient as possible.

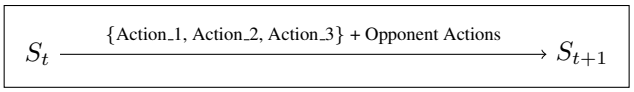
3.2.1 Handling Multiple Shipyards

The Kore game allows to take an action for each shipyard, as visualized in Figure 3a. By simply producing actions for n shipyards in one step, an agent has to choose from the set A^n, where A are all possible actions for a shipyard. This exponential action space blow-up by the number of shipyards makes training substantially more difficult such that we reformulate the problem into only considering one shipyard in each step. Figure 3b shows that we decompose one original game step from state S_t to S_{t+1} into a sequence of substeps $S_t^1, \ldots, S_t^n, S_{t+1}^1$, where always only the action for shipyard i is predicted. In each substep, the previous actions for game step t are accumulated and already simulated on the board. For example, the building of ships directly leads to a reduction of the Kore by the ship cost in the next substep. That way the network does not has to explicitly learn these action consequences and the Markov property also holds for the substep environment. When all shipyards have been processed, the accumulated actions and the opponent actions are simulated on the board such that the sequence of substeps corresponds to exactly one game step. The reward is assigned to the last substep of game step t and all other substeps receive zero reward. Since we use undiscounted rewards, the rewards in the Kore game and the substep environment are equal for the same actions.

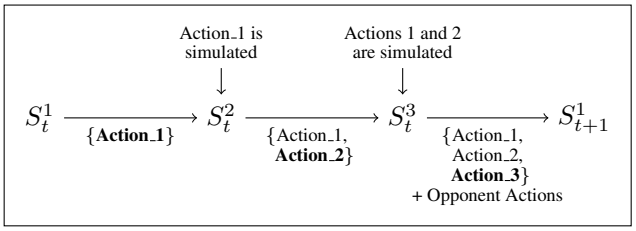
The substep approach is independent of a specific reinforcement learning technique such that it can be easily utilized with different methods such as PPO, DQN or A2C. On the other hand, this approach increases the number of required forward passes by the number of shipyards. Furthermore, the network needs to learn to focus on a specific shipyard, but this can easily be supported by an appropriate input structure, which is outlined in Section 3.4. Additionally, the network may focus too narrowly on the current shipyard and may not produce the globally best solution for all shipyards. Another more efficient solution that overcomes this narrow view would be to use networks that can easily output multiple action predictions. We outline an idea of using a transformer-based architecture for simultaneously predicting multiple actions in Section 3.5.2.

3.2.2 Avoiding varying action sizes

To avoid varying action sizes a translation scheme is implemented. Our policy $\pi : S \rightarrow F$ chooses a function



(a) Kore Game



(b) Substep Environment

Figure 3: A game step with three shipyards in the Kore game (a) and the constructed substep environment (b), where always only the action for shipyard i is predicted based on state S_t^i. The sequence of substeps corresponds to exactly one game step, as depicted in (a).

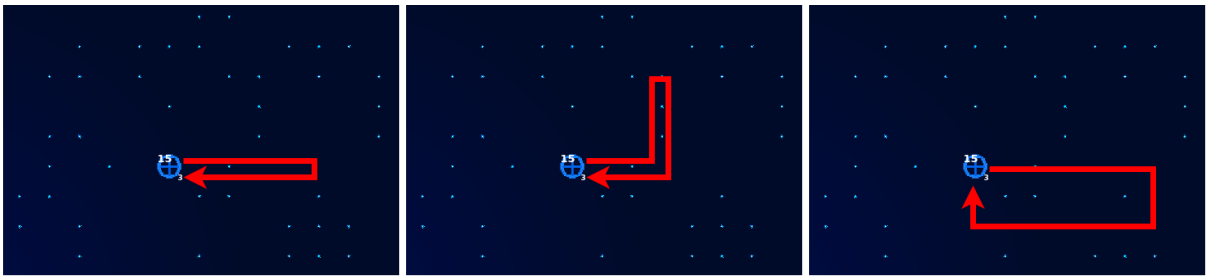
$f \in F, f : S \rightarrow A' \subseteq A$ which leads to π picking actions from A indirectly. The functions itself are then implemented in a rule-based way to compute actions of varying length and delegating it to the game environment.

In total 13 functions were implemented which construct a game action and are divided in these categories:

Attack Attack is the action of computing the shortest path from the players shipyard to a shipyard of the opponent and afterwards calculating the flight plan which will end up at that position.

Mining Mining is a subset of launch actions that will launch ships in order to collect Kore and transport it back to the shipyard. To do that efficiently, a path with the most Kore is calculated and afterwards constructed as a flight plan. The path is searched within a box of radius 9 where the center is the position of the shipyard. Depending on the shape of the path the flight plan length changes and therefore the number of required ships. We divided the mining action with regards to the shape of the path into multiple mining actions. Each mining action requires a different amount of ships in the shipyard. The different actions are shown in figure 4. If there are too few ships in the shipyard, this action is not allowed at the current time step.

Expand Expanding describes the act of launching a fleet and creating a new shipyard at a calculated location. The expand actions differ in the algorithm that computes the location of the shipyard. A Kore based expansion chooses the location with the most Kore within a given radius. Other less greedy and more secure based actions expand relative to the existing shipyard. Circular expansions revolve around expanding in a rotation wise manner on a circle whose origin is the center of the games map.



(a) Axis Mining (b) Boomerang Mining (c) Box Mining

Figure 4: Exemplary paths of each mining action. Mining along an axis (a) requires at least 3 ships. For the boomerang mining (b) and box mining (c) the shipyard needs to hold at least 21 ships since the flight plan has to contain at least 4 changes of direction.

Wait This category describes the action of intentionally waiting. Waiting is necessary to plan actions that can be only executed if some time has passed. One example would be waiting for launched ships to return in order to have enough ships to expand. Additionally if the game gets decided in turn 400 by having more Kore the ability to not build before the end would increase the chance of winning.

3.2.3 Reducing number of possible actions with growing action sizes

Using the translation scheme explained previously, the size of the action space is controlled by $|F|$. F can be designed to be arbitrarily small, in our case $|F| = 13$, shrinking the games large action space significantly.

3.2.4 Handling constrained actions

Taking an action that is not allowed at the current time step by some constraint will be converted by the Kore environment into an action that does nothing. Any action that is converted in this way will be called *illegal action* in the following. According to [13] different strategies affect learning convergence significantly in games with many possible illegal actions like Dota 2 [3]. Our final approach is masking out illegal actions as it returned the highest episodic reward throughout the whole training period compared to other methods as illustrated in figure 5. This could be possibly due to retaining a valid policy gradient [13] while forcing the agent to take legal actions.

3.3. Learning Algorithms

Through the course of our work we used three different types of model-free learning algorithms because they are easier to train than model-based methods. Firstly, we present two policy optimization methods and lastly a Q-learning algorithm.

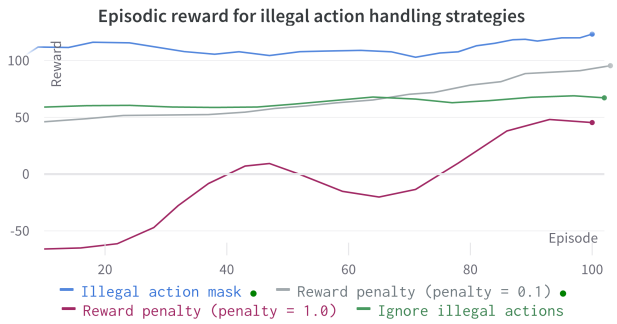


Figure 5: Comparison of training progression over the first 100 episodes when using different strategies for handling illegal actions. A carefully selected penalty in the reward (grey curve) will receive higher rewards than ignoring illegal actions (green curve) after around 50 episodes. Illegal action masking leads to a better policy from the beginning (blue curve).

3.3.1 Advantage Actor Critic

A synchronous version of advantage actor critic, based on [18], was trained in the Kore environment. The advantage function is estimated from a n-step return given by

$$\tilde{A} = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V(s_{t+k}; \theta_v) - V(s_t; \theta_v) \quad (3)$$

where $V(s_t; \theta_v)$ is the estimated state-value of state s_t at time step t from the critic with parameter set θ_v, γ is the discount factor and r_t is the reward at time step t. The parameters θ of π are then updated according to the gradient $\Delta_\theta \pi(a_t|s_t;\theta)\tilde{A}$. Also a regularization term $\beta \Delta_\theta H(\pi(s_t;\theta))$ is added to discourage convergence to suboptimal deterministic policies where $H(\pi)$ is the entropy of π.

Since we have finite length episodes with very delayed rewards $\gamma = 1.0$ was used. Furthermore $n = 5$ and $\beta = 0.01$. The policy was optimized with Adam and learning rate 0.00025.

3.3.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a policy optimization algorithm based on the Actor to Critic (A2C) framework and uses a similar strategy to the Trust Region Policy Optimization (TRPO) [25] to prevent the policy from diverging [24]. TRPO implements KL Divergence to establish a trust region around the current policy. The new policy is only allowed to update in the trust region, effectively removing the possibility of divergence.

Instead of using the KL Divergence to limit the policy explosion, PPO uses a first-order approximation to simplify its implementation. PPO maintains an older copy of the policy and defines the policy ratio as follows:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \tag{4}$$

The goal is to not let the policy ratio grow too much in one iteration to prevent instability. To this end, PPO defines the following clipped objective function which truncates the policy ratio between the range [1-ϵ, 1+ϵ].

$$\begin{aligned} L(s, a, \theta_k, \theta) = \min &\left(r(\theta) A^{\pi_{\theta_k}}(s, a),\right. \\ &\left.\text{clip}\left(r(\theta), 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\right) \end{aligned} \tag{5}$$

where θ is the parameter used to parameterize the policy function (weights of the policy network), s is the current state and a is the action at state s that we are considering taking. A is the advantage function under policy π (it estimates how good an action is compared to the average action at a specific state) and is defined as follows:

$$A(s, a) = Q(s, a) - V(s) \tag{6}$$

In our implementation of PPO, both the Q-value function $Q(s, a)$ and the value function $V(s)$ are modelled by NNs and the hybrid model (section 3.5.1) is used for feature extraction. We used gamma=1, learning rate=0.0008 with RELU activation function.

3.3.3 Deep Q-Learning

The basic idea of Deep Q-Learning is to approximate the q-values $Q(s, a)$ for every state-action pair (s, a) through a deep neural network. Thereby the q-values are the expected return for taking action a in state s [19].

Since the original publication of Deep Q-Learning there have been several enhancements. The ones we are using are double dqn[27] for enhanced stability and a dueling architecture [30].

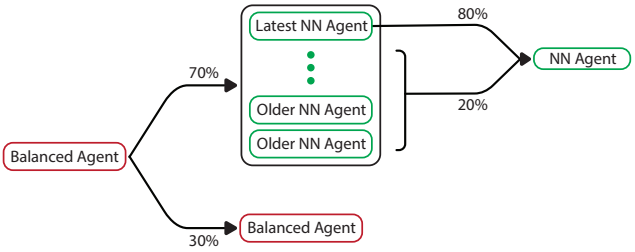


Figure 6: Sampling in self-play: At the start of each game, we pick the balanced latest agent with 30% probability; with the remaining 70% probability, we sample agents from the agent pool with an 80/20 split between the latest agent and an older agent.

Because of finite length episodes we used the discount factor $\gamma = 1$. Also we synchronized target network and train network every 10000 steps. Training was performed with the Adam optimizer and a learning rate of 0.0001.

3.3.4 Competitive Self-Play

In reinforcement learning, self-play is a technique where an agent competes against an older version of itself. This ensures that the task is always of the right difficulty for an agent to improve. During self-play, an agent is expected to can come up with new techniques to compete with an ever stronger competing agent.

In PPO the enemy agent is initialized with the balanced agent (which comes prepackaged with the kore environment) at the start of training. With a certain *dump frequency*, the current agent is added to the pool of available agents. For each episode of the game, we sample the enemy agent from the pool with a certain frequency as described in Figure 6. In PPO, the enemy agent is also able to independently control all the shipyards, thanks to the substep environment discussed in 3.2.1.

3.4. State Representation

The Kore game has many scalar information such as the amount of Kore or the game step but also spatial information such as the Kore distribution or the fleet positions, as can be seen in Figure 1. The board configuration is translated into the current state after each game step. This state representation is utilized by the neural networks in the following section to make the action predictions. The scalar game characteristics such as the amount of Kore, the game step, the number of ships and shipyards for our and the opponent player are included in the state. We represented the spatial information, e.g. the Kore distribution, fleets, shipyards and cargo as multiple feature maps $M_i \in \mathbb{R}^{21\times21}$ for our player and the opponent player in the state. However, some information such as the flight plans are not scalar and

also not spatial information such that we computed a translation of this information into feature maps. Thereby, it may be easier for the network to relate the flight plan information to the other spatial information. We calculate the route of each flight plan with a lookahead of 50 steps and indicate the numbers of steps i until the fleet will be at this position in the feature map by a linearly decreasing number:

$$val(i) = \frac{50 - i}{50} \tag{7}$$

One other representation for the flight plans we utilized takes inspiration from the addition of different noise levels. The idea is that we think of a fleet making a certain amount of "noise" proportional to its size. We further think of this noise level as being exponentially decreasing over the route the fleet is going to take:

$$noise(t) = s \cdot b^t, \tag{8}$$

where s is the fleet size, t is the step of the route (i.e. for $t = 0$ the formula evaluates to noise cast to the current position of the fleet) and b is a hyper-parameter influencing the degree of noise decay. We chose $b = 0.933$ which halves the noise level around every 10 steps which is roughly the radius of the map. Adding the noise levels cast by the fleets for every cell on the map gives us our representation. This map representation of the flight plans tries to combine the two modes of distance and fleet size by giving higher values when either the incoming fleet is very large or close.

Alternatively, autoencoders or techniques from natural language processing could easily be utilized to encode the flight plans. But these techniques would probably require more training as a detailed understanding of flight plans is essential since a flight plan may be assumed as non-returning to a shipyard if the interpretation is just off by one cell.

We have implemented a manifold of scalar and spatial state information features in a Wrapper class for the game board and implemented multiple states from that. Figure 7 gives an overview of the most important scalar and spatial inputs for the hybrid model that is introduced in the following section 3.5.1.

Since our models predict only the action for one shipyard in each step, as described in Section 3.2.1, we focus on that shipyard by recentering all feature maps such that the current shipyard is at the center of all maps. Thereby, the network does not need to explicitly learn to connect the current shipyard to its position in the feature map. Additionally, we provide some specific information such as the number of ships or the maximum spawn rate of the current shipyard. For the other shipyards, these information is only represented on feature maps.

3.5. Deep Learning Models

We introduce a family of hybrid networks consisting of different parts for scalar and spatial information in Section 3.5.1. Subsequently, we introduce a multi-modal transformer architecture and its possible extension to predictions for multiple shipyards in Section 3.5.2.

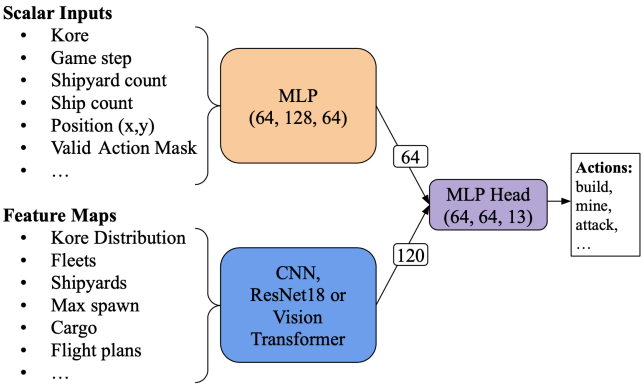


Figure 7: Overview of the hybrid network architectures with the most relevant inputs. The numbers indicate the layer and vector dimensionality respectively.

3.5.1 Hybrid Models

Our first model family uses separate networks for the scalar and spatial information that compute hidden representations for calculating the action prediction, as visualized in Figure 7. The multi-layer perceptron (MLP) for the scalar information consists of three fully connected layers with 64, 128 and 64 neurons respectively. For the spatial information, we utilized different convolutional neural networks (CNN). Firstly, we experimented with a small CNN and upgraded this to a ResNet18 [9] with residual connections, where we only adapted the input channels to the number of feature maps and output a hidden representations from the last layer. Additionally, we experimented with vision transformers [6], where we reduced the network size such that it was computationally more efficient. The MLP head predicts the action based on the concatenated output of the networks for scalar and spatial information. The MLP head consists of two fully connected layers with 64 neurons and an output layer. We use ReLU activation [7] throughout the networks. Due to the manifold of different reward functions, as described in Section 3.1, and the limited computing resources at the RWTH compute cluster, we did not tryout every network and reward combination but focused on the hybrid network with the ResNet18. For more details regarding the neural networks, we refer to the implementation in the git repository under `src/agents/neural_networks`.

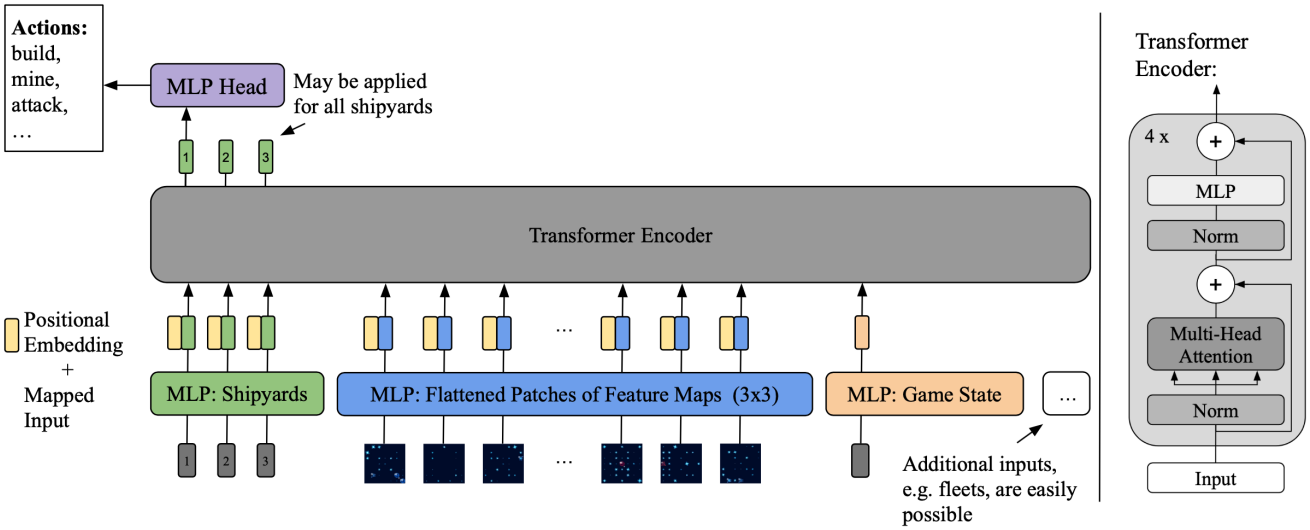


Figure 8: The multi-modal transformer takes the shipyard information (e.g. ships, max. spawn) for each shipyard, the patches of all feature maps and the game state information (e.g. game step) into account by applying a modality-specific MLP. Afterwards the vectors are positionally embedded and fed into the transformer encoder. Finally, an MLP head is applied to the obtained hidden representation of the current shipyard to predict the action.

3.5.2 A multi-modal Transformer

The state information comprises multiple different modalities such as scalar, spatial and textual (e.g. flight plans) information. Hence, a model needs to be able to handle multiple input modalities. Furthermore, if we do not utilize the substep approach of always considering one shipyard, the network should be able to predict actions for all shipyards in one forward pass. Hence, we decided to implement a multi-modal network inspired by the Perceiver IO [14] for arbitrary data. Figure 8 shows that we are considering three different input modalities. Namely, these input types are multiple shipyards, all feature maps (e.g. Kore distribution, fleets, flight plans, etc.) and global game state information. Each of these input types is firstly mapped into a hidden space since each modality is very different. Therefore, we use for each input modality a dedicated tiny MLP consisting of two fully connected layers with 64 neurons and ReLU. The feature maps are processed in a similar way as in the vision transformer [6]. The maps are decomposed into patches of 3×3 game cells leading to 49 patches, which are then flattened and mapped into the hidden space by the specific MLP. Since the network needs to interpret the spatial information correctly and the transformer is permutation invariant, we add a sinusoidal embedding, where we use the center of each patch as its position. Additionally, the network needs to relate the shipyards to the spatial information such that the shipyards also receive a sinusoidal embedding. We utilized the 2D-extension [20] of the classical sinusoidal embedding [28], where the x-coordinate and y-coordinate are encoded in the upper half and lower half of the vector respectively. The embedding is simply added to the mapped inputs in the hidden space. We also experimented with learned embeddings, but this performed worse, which may be due to the difficulty of relating the scalar to the spatial shipyard information. These inputs are then fed into the transformer encoder network consisting of four transformer encoder layers. Each of the transformer encoder layer uses four-head attention [28], layer norm [2], a MLP with GELU activation [10] and residual connections [9], as visualized in Figure 8. The size of the embedding and the hidden size throughout the network is 64. In the transformer encoder network, the information of each shipyard can attend to the other information such that the transformer encoder outputs a hidden representation for each shipyard, which is subsequently utilized by an MLP head to make a prediction for the current shipyard. This is similar to utilizing a class token in the vision transformer [6] or BERT model [5] for classification.

Hence, it would be easy to extend this model to predicting actions for each shipyard by applying the MLP head to the obtained hidden representation of each shipyard. As the change to multiple shipyard predictions would require a substantial modifications of our environment and we would loose the advantages of the substep approach, we only suggest this as future work. Furthermore, since we started implementing this network at the end of the lab, we mainly adapted the input structure of the hybrid network. However, the model could easily be extended to further input modalities such as fleets, which could help the network to

comprehend the game better.

The multi-modal transformer and the Hybrid ResNet achieve similar rewards during training with A2C, as Figure 9 shows for the Advantage reward. However, the Training with the CNN is smoother than with the multi-modal transformer, where the reward is more oscillating.

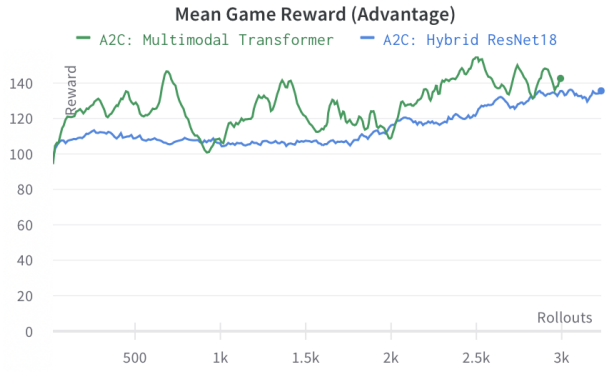


Figure 9: Comparison of the mean Advantage reward during training for the multi-modal transformer (green) and the hybrid ResNet18 (blue) at the RWTH Compute Cluster for 12 hours on one GPU.

4. Results

Finally, we will summarise the experiments and present the results of our agents when played against the Kore agents that come bundled with the environment. These Kore agents are described below:

1. **Balanced**: This is the strongest agent in the group. It can mine Kore, expand by spawning more shipyards as well as attack the opponent's shipyards.
2. **Minor**: This agent only mines the Kore and can not attack or defend.
3. **Random**: This agent just takes a random high-level action at each state.

The training of our agents is briefly described below:

1. **PPO** is trained using Stable Baselines3 and we have trained it with and without competitive self-play. The Annealing reward is used and we start with the Advantage reward and slowly anneal towards the Win-Lose reward. This is trained for 4.2k episodes.
2. **A2C** is trained against the balanced agent using the Advantage reward and multi-modal transformer. It is trained using Stable Baselines3 for 900 episodes. Notably, longer training leads to a decreasing win rate although the reward still increases. Hence, the network probably overfits to the Advanatge reward that is only an imperfect proxy for the win rate.
3. **DQN** is trained against the balanced agent using the Competitive Kore-Delta reward with Trophy. It is trained using the Keras RL library. It was trained for 1.2M steps (which is about 4800 episodes).
4. **Rule Based Agent** (RBA) uses the same high-level actions as the RL agents but the rules to initiate these actions are hand coded.

	PPO	PPO (Self-Play)	A2C	DQN	RBA
Balanced	**80**	77	70	63	72
Miner	**100**	99	**100**	94	97
Random	**100**	**100**	**100**	95	**100**

Table 1: The results show the percentage of episodes that our agent wins against a Kore agent.

The Table 1 represents the win rates when our agents play against the Kore agents.

5. Conclusion

The Kore game has a very large and complex action space compared to other video games where the action space is often only to press or not to press some buttons. To efficiently utilize state-of-the-art reinforcement learning techniques, we had to reduce the action space to 13 high-level actions such as mining, attacking or expanding. Additionally, the variable number of shipyards leads to a variable number of action predictions in each step that we circumvented by sequentially predicting the action for each shipyard in the substep environment. Since the Kore game is very long, we had to experiment with different reward functions that balance the actual goal of winning and a certain density to train efficiently. We experimented with different state-of-the-art deep learning models such as multi-modal transformers and hybrid ResNets that achieved very similar performance. Additionally, we utilized a manifold of reinforcement learning techniques such as A2C, DQN and PPO with and without self-play. Our best agent (PPO) performs well against the rule-based balanced agent from Kaggle with roughly 80% win rate and our agent is also better than a rule-based actor with our high-level actions. However, the best agent [4] in the Kaggle competition was purely rule-based and, at the time of writing, the best disclosed deep-learning solution [16] was on place 13. This deep learning solution uses imitation learning with language modelling and differs substantially from our approach. Hence, the optimal deep reinforcement learning techniques for the Kore game may have yet to be developed. Interesting open research areas for the Kore game include the extension of

the multi-modal transformer to predicting multiple actions, fine-tuning of self-play with PPO and model-based reinforcement learning methods.

References

[1] URL: https://www.kaggle.com/code/bovard/kore-balanced-agent.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[3] Christopher Berner et al. "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680* (2019).

[4] Harm Buisman. *Kore 2022: 1st place solution*. URL: https://www.kaggle.com/competitions/kore-2022/discussion/340035.

[5] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[6] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations*. 2021.

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 315–323.

[8] Danijar Hafner et al. "Mastering atari with discrete world models". In: *arXiv preprint arXiv:2010.02193* (2020).

[9] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[10] Dan Hendrycks and Kevin Gimpel. "Gaussian error linear units (gelus)". In: *arXiv preprint arXiv:1606.08415* (2016).

[11] Matteo Hessel et al. "Rainbow: Combining improvements in deep reinforcement learning". In: *Thirty-second AAAI conference on artificial intelligence*. 2018.

[12] Juan Camilo Gamboa Higuera, David Meger, and Gregory Dudek. "Synthesizing neural network controllers with probabilistic model-based reinforcement learning". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 2538–2544.

[13] Shengyi Huang and Santiago Ontañón. "A closer look at invalid action masking in policy gradient algorithms". In: *arXiv preprint arXiv:2006.14171* (2020).

[14] Andrew Jaegle et al. "Perceiver io: A general architecture for structured inputs & outputs". In: *arXiv preprint arXiv:2107.14795* (2021).

[15] Lukasz Kaiser et al. "Model-based reinforcement learning for atari". In: *arXiv preprint arXiv:1903.00374* (2019).

[16] Fumihiro Kaneko. *Kore 2022: Imitation learning with language modeling*. URL: https://www.kaggle.com/competitions/kore-2022/discussion/337476.

[17] *Kore 2022: Information*. URL: https://www.kaggle.com/competitions/kore-2022/.

[18] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[19] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[20] Zobeir Raisi et al. "2D positional embedding-based transformer for scene text recognition". In: *Journal of Computational Vision and Imaging Systems* 6.1 (2020), pp. 1–4.

[21] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[22] Simon Schmitt, Matteo Hessel, and Karen Simonyan. "Off-policy actor-critic with shared experience replay". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 8545–8554.

[23] Julian Schrittwieser et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[24] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[25] John Schulman et al. *Trust Region Policy Optimization*. 2015. DOI: 10.48550/ARXIV.1502.05477. URL: https://arxiv.org/abs/1502.05477.

[26] David Silver et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[27] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[28] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[29] Tingwu Wang and Jimmy Ba. “Exploring model-based planning with policy networks”. In: *arXiv preprint arXiv:1906.08649* (2019).

[30] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2015. DOI: 10.48550/ARXIV.1511.06581. URL: https://arxiv.org/abs/1511.06581.

[31] Manuel Watter et al. “Embed to control: A locally linear latent dynamics model for control from raw images”. In: *Advances in neural information processing systems* 28 (2015).

[32] Yuxin Wu and Yuandong Tian. “Training agent for first-person shooter game with actor-critic curriculum learning”. In: (2016).