STRESZCZENIE

Celem niniejszej pracy inżynierskiej jest stworzenie systemu pozwalającego na porównywanie agentów uczonych ze wzmocnieniem w środowisku Google Research Football (GRF), którzy nigdy przeciwko sobie nie grali, ale dostępne są mecze z graczem pośrednim. Zaproponowane rozwiązanie polega na jednorazowym stworzeniu reprezentacji agenta na podstawie meczów, jakie rozegrał on z graczem pośredniczącym. Taka reprezentacja może potem zostać użyta do porównywania go z przeciwnikami, z którymi wcześniej się nie spotkał. Rozegranie meczu jest kosztowną operacją. Funkcja określająca liczbę meczów, jakie należy rozegrać między wszystkimi zawodnikami ze zbioru graczy w celu ich porównania ma złożoność kwadratową względem mocy zbioru graczy. Liczba meczów wymaganych w przedstawionej metodzie rośnie liniowo pod względem liczby zawodników, ponieważ grają oni tylko z graczem pośrednim.

Pierwszy rozdział zawiera kontekst problemu oraz opisuje zakres pracy inżynierskiej i podział obowiązków zespołu. Drugi rozdział skupia się na wprowadzeniu i wytłumaczeniu narzędzi, które zostały użyte w opracowanym algorytmie. Trzeci rozdział szczegółowo opisuje środowisko GRF. W dwóch kolejnych rozdziałach autorzy przedstawiają opracowane przez siebie metody służące porównywaniu agentów na podstawie meczów z graczem pośredniczącym. Szósty rozdział został poświęcony opisowi szczegółów implementacyjnych zaproponowanych metod. W siódmym rozdziale została opisana aplikacja oraz graficzny interfejs użytkownika. Ósmy rozdział przedstawia przeprowadzone eksperymenty i porównanie zastosowanych podejść do problemu. Ostatni rozdział podsumowuje wyniki badań oraz możliwe kierunki rozwoju.

Przewidywanie wyniku meczu tylko na podstawie rozgrywek z graczem pośredniom okazało się skomplikowanym problemem. Zarówno agenty uczące się ze wzmocnieniem jaki środowisko cechują się dużą wariancją co powoduje, że nawet mecze tego samego gracza znacząco się od siebie różnią. Pomimo tego zaproponowane rozwiązanie osiąga obiecujące wyniki, chociaż ma problemy z graczami cechującymi się odmiennym stylem gry.

Podział pracy:

- Dominik Grzegorzek
  Pracował na infrastrukturą potrzebną do stworzenia danych treningowych. Wyselekcjonował agentów do konstrukcji zbioru treningowego i nadzorował jego generację. Napisał klasy opakowujące dla środowiska Google Research Football odpowiedzialne za zbieranie statystyk z meczów. Zaimplementował skrypty treningowe. Trenował różne rodzaje architektury uczenia głębokiego. Przeprowadził eksperymenty z wytworzonymi modelami oraz porównał otrzymane wyniki. Pracował nad graficznym interfejsem aplikacji.
- Szymon Żebrowski
  Stworzył skrypty i infrastrukturę potrzebną do wygenerowania zbioru treningowego. Zarządzał magazynem danych na google cloud platform, w którym znajdowały się wygenerowane dane. Pełnił nadzór nad procesem kolekcjonowania danych, pracował nad jego zrównole-

gleniem oraz kontrolował ilość zasobów zużywanych przez serwer. Zaprojektował i zaimplementował architekturę oraz stronę serwerową aplikacji, przygotował aplikację do uruchomienia jako kontener na klastrze Kubernetes.

- Jan Pliszka

  Napisał klasy opakowujące dla środowiska Google Research Football służące do zbierania statystyk z meczów. Czytał artykuły naukowe, zebrał literaturę oraz opracował pomysły na eksperymenty do przeprowadzenia. Zaprojektował i implementował architektury głębokich modeli sztucznej inteligencji oraz skrypty do ich trenowania. Prowadził i nadzorował treningi sieci neuronowych. Analizował zbiory treningowe oraz walidował wytworzone modele. Koordynował pracę zespołu.

Słowa kluczowe: neuronowe sieci splotowe, regresja, uczenie ze wzmocnieniem

Dziedziny nauki i techniki zgodne z wymogami OECD: Nauki o komputerach i informatyka

ABSTRACT

The purpose of the thesis is to create a scoring system that allows comparing different reinforcement learning agents who have not played games with each other in the Google Research Football environment. However, games with the proxy player are available. The proposed solution relies on creating agent representation just once based on the matches played against a proxy player. This representation can be then used to compare it with different opponents which agent never played with. Playing the match between agents is a costly operation. The function that determines the number of matches to be played between all players in the set of players for the purpose of comparison has a square complexity in relation to the cardinality of the set of players. The number of matches needed for the presented solution grows linearly in terms of the number of agents as they are played only against a proxy player.

The first chapter contains the background of the problem, describes the scope of this thesis, and segregation of duties. The second chapter focuses on the introduction and explanation of the tools used in the algorithm. The third chapter covers the GRF environment in depth. In the next two chapters, the authors present methods they developed that allow comparing agents based on their matches with the proxy player. The sixth chapter explains the implementation detail of both methods. The seventh chapter describes the application and graphical user interface for comparing the agents based on the proposed algorithm. The eighth chapter discusses experiments and comparison of different approaches. The last chapter shows the result of research, directions of development, and possible use-cases of the created solutions.

The task of predicting the results of the match based only on the matches with proxy player turned out to be a complex problem. Both reinforcement learning agents and environments show a large variance causing even the same player's matches to vary significantly. Despite that proposed solution achieves promising results, but had problems classifying agents with an unusual style of playing.

Assignment of works:

- Dominik Grzegorzek
  Worked on the infrastructure needed to create training data. Chosen the agents for dataset creation and supervised the process of generation. Wrote statistic-collecting wrappers for the Google Research Football environment. Implemented the training scripts. Trained different types of deep learning architectures. Conducted experiments with created models and compared results of the used methods. Worked on the application frontend.
- Szymon Żebrowski
  Programmed the scripts and infrastructure for dataset generation. Managed storage on the google cloud platform for storing generated data. Took care of the data generation collection process, worked on the process parallelization and training server resources control.

Designed and implemented the application's architecture and backend, prepared the application to be run as containers on a Kubernetes cluster.

- Jan Pliszka

  Worked on the wrappers for collecting match statistics in Google Research Football environment. Reviewed scientific articles, collected literature and ideas for the experiments. Designed and implemented deep neural architectures and training scripts. Ran and monitored training of the neural networks. Analyzed the datasets and validated model outputs. Coordinated work of the team.

Keywords: computer science, convolutional neural networks, regression, reinforcement learning

OECD field of science and technology: Computer and information sciences

CONTENTS

LIST OF ABBREVIATIONS

- CNN - Convolutional Neural Network
- GPU - Graphical Processing Unit
- GRF - Google Research Football
- GUI - Graphical User Interface
- JSON - JavaScript Object Notation
- MAE - Mean Absolute Error
- MSE - Mean Squared Error
- NN - Neural Network
- OS - Operating System
- PV - Persistent Volume
- PVC - Persistent Volume Claims
- RL - Reinforcement learning
- SGD - Stochastic Gradient Descent
- SVM - Support Vector Machine
- SVR - Support Vector Regression
- ML - Machine Learning
- RWX - Read Write Many
- API - application programming interface
- NFS - Network File System

1. INTRODUCTION (DOMINIK GRZEGORZEK)

Recent progress in the field of reinforcement learning (RL) has been accelerated by virtual learning environments such as video games, where novel algorithms and ideas can be quickly tested in a safe and reproducible manner. Google Research Football (GRF) [15] is a reinforcement learning environment based on open-source, physics-based 3D game Gameplay Football. In this environment, an agent is controlling one football team and plays against the second team, which can be controlled by another agent or human. The purpose of the thesis was to create a scoring system that allows comparing different RL agents who have not played games with each other in the Google Research Football environment.

The additional requirement for the algorithm of comparing the collection of agents is to have possibly the best computational complexity in terms of the size of that collection. The need arises from the potential use in reinforcement learning. Setting with more than one RL agent acting concurrently, each having their own policy, is called competitive reinforcement learning. The currently leading solutions in this field, which can achieve superhuman results in games such as Starcraft[39] and Dota 2[22], are based on an approach of self-play, where an agent is continuously playing against its previous policy. Attaining satisfying results demands an opponent to be as competitive as it can be. Choosing the most recent previous version of an agent to play with is not the best option. It can lead an agent to be able to win only with a specific opponent. To give the model an ability to generalize, the perfect solution is to play against the opponent with the greatest probability to win the match. To accomplish this, there is a need to compare all policies held, what can be done by the scoring system being the purpose of research.

1.1. Assignment of works

The team consists of Dominik Grzegorzek, Jan Pliszka and Szymon Żebrowski. The division of labour was as follows:

- Dominik Grzegorzek
  Worked on the infrastructure needed to create training data. Chosen the agents for dataset creation and supervised the process of generation. Wrote statistic-collecting wrappers for the Google Research Football environment. Implemented the training scripts. Trained different types of deep learning architectures. Conducted experiments with created models and compared results of the used methods. Worked on the application frontend.
- Szymon Żebrowski
  Programmed the scripts and infrastructure for dataset generation. Managed storage on the google cloud platform for storing generated data. Took care of the data generation collection process, worked on the process parallelization and training server resources control. Designed and implemented the application's architecture and backend, prepared the application to be run as containers on a Kubernetes cluster.

- Jan Pliszka

  Worked on the wrappers for collecting match statistics in Google Research Football environment. Reviewed scientific articles, collected literature and ideas for the experiments. Designed and implemented deep neural architectures and training scripts. Ran and monitored training of the neural networks. Analyzed the datasets and validated model outputs. Coordinated work of the team.

2. BACKGROUND AND RELATED WORK (JAN PLISZKA)

In this chapter, the authors explain the fundamentals of reinforcement learning. They are also presenting the regression analysis method, describing the Transformer model architecture, its attention mechanism, and convolutional networks, which will be used in further work.

2.1. Reinforcement learning concept

Reinforcement learning[33], as a subfield of machine learning, is a method in which models are trained empirically. The key concept of reinforcement learning is that agents exist in the environment and can interact with it. The environment has a dynamically changing over time state s, which usually part of can be observed by an agent as an observation at in each interaction step (In fully observed environment s is equal to o). Iteratively, an agent performs an action a_t, basing on the current observation and its own policy. Actions affect the state of the environment. The agent also perceives from the environment a reward r_t which numerically describes how good his step was. The reward is an abstract value, if it is positive it means that the action taken has a positive effect, otherwise it is a negative scalar. An agent's goal is to maximize cumulative reward, which is called a return: $G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$, where T is the final step. One step, which consists of a sequence of state, action, reward, and the next state is presented in Figure 2.1.

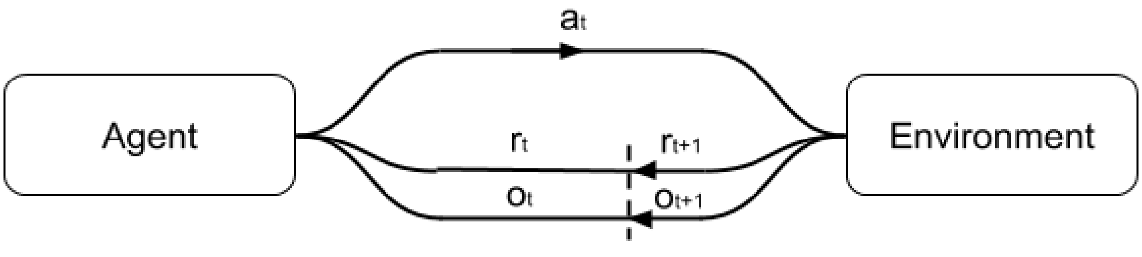


Fig. 2.1. Reinforcement learning problem framing

Series of transitions form a trajectory. The trajectory that starts in the initial state s_0 and ends in a terminal state s_T create an episode. The whole training often requires a vast number of episodes to find out the expected effect.

RL is a learn-by-doing approach. The agent is ought to conclude from this how his own actions interfere with the reward. For that reason, an agent has its own policy, which defines how it should act depending on the data it has got. A policy is a state to action mapping, which specifies the agent. It is usually mathematically denoted by μ when it is deterministic, or π when it is stochastic. Usually parametrized policies are used in reinforcement learning. Those parameters are adjusted and optimized during the training. Action computation equation is denoted by $a_t = \mu_\theta(s_t)$ or $a_t \sim \pi_\theta(\cdot|s_t)$, respectively for deterministic and stochastic policy.

2.2. Regression analysis

Regression analysis [46] is a statistical method that allows finding patterns between dependent value, further called outcome, and independent variables usually named predictors. Regression is widely used for forecasting and predicting using one or more predictors. In practice the use of regression comes to two stages:

- Function construction - a regression model preparation, which will map the relationship between the outcome and predictors. The function itself depends on the used regression algorithm, not only can it be a simple equation, but also sophisticated machine learning (ML) model like Random Forest Regression(2.2.3).
- Scoring - usage of a calculated model to find out the dependent value on data containing only features.

In formal notation regression takes the form of the following equation:

$$Y = f(X, \beta) + \epsilon$$

Where: Y is the dependant value, X is a set of predictors, β is the set of regression coefficients, which are usually real numbers, ϵ refers to an additive error term, which represents all other factors influencing the dependant value.

2.2.1. Linear regression

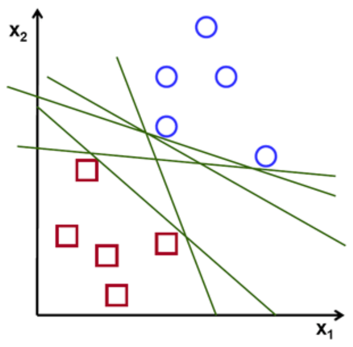
Linear regression [13] assumes that there is a linear relationship between the dependent variable y and p sized vector of regressors x, taking into account an error ϵ - an unobserved random variable. In case that in researched data is only one predictor we are dealing with simple regression, when the set of independents consists of more than one variable the process is called multiple linear regression and is expressed by a matrix equation:

$$y = X\beta + \epsilon$$

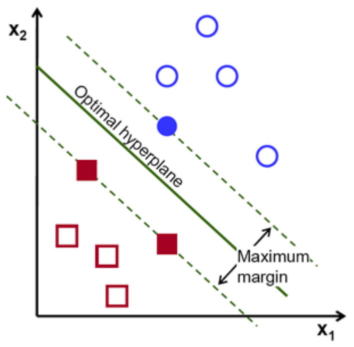
Where matrices have the same marking and are respectively:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{pmatrix}, \quad \boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}.$$

The most commonly used approach for solving overdetermined systems of equations i.e those existing in linear regression is the least-squares method[43]. Overdetermined, in this sense, means that the system has more equations than unknown values. In the least-squares method, the sum of squares of residuals created in each equation is minimized.



(a) Possible hyperplanes splitting classes[9]



(b) Optimal hyperplane calculated using SVM[9]

2.2.2. Support Vector Regression

Support vector regression(SVR) [8] is an extension of support vector machines(SVM) [37], concept used in classification problems, to perform regression. The intention of those machines is to find a hyperplane that optimally separates samples in N-dimensional space belonging to different classes 2.2b. Optimally means that the plane has a maximal distance to both classes.

As the margin increases, the classifier generalization error(the measure that describes how accurately the model classifies an unseen data) decreases. For the regression tasks, the objective of the original support vector machine is changed. It aims to divide hyperspace in such a way that for each sample the predicted value lies not further than epsilon from the true value and at the same time keeping the created hyperplane as flat as possible. Where epsilon is the hyperparameter choosen for the problem.

2.2.3. Random forest regressor

In the paper from 1990 “The Strength of Weak Learnability” [29] RE Schapire theoretically proved that a group of weak learners can achieve an arbitrarily low error showing that they are equivalent to one strong learner. Random forest is a machine learning method firstly introduced by Tin Kam Ho in a paper “Random decision forests” [36] in 1995. It uses an ensemble learning technique to perform the regression. The ensemble consists of many decision trees which are tree-structured models used to perform regression or classification on the data. The algorithm of creating such a structure is described in the “Induction of decision trees” [24] article. Decision trees are usually trained on different subsamples of the dataset to keep independence between them. Even though individual trees may perform badly the majority voting ensures that the predictions of the whole ensemble are on a sufficient level.

2.3. Deep learning

Deep learning [16] is a machine learning technique using artificial neural networks to learn to solve tasks based on the data. This solution is inspired by the neurons of the living organisms. Artificial neural networks consist of neurons that form structures called layers. The word deep comes from the number of hidden layers. Networks with more than one hidden layer are consid-

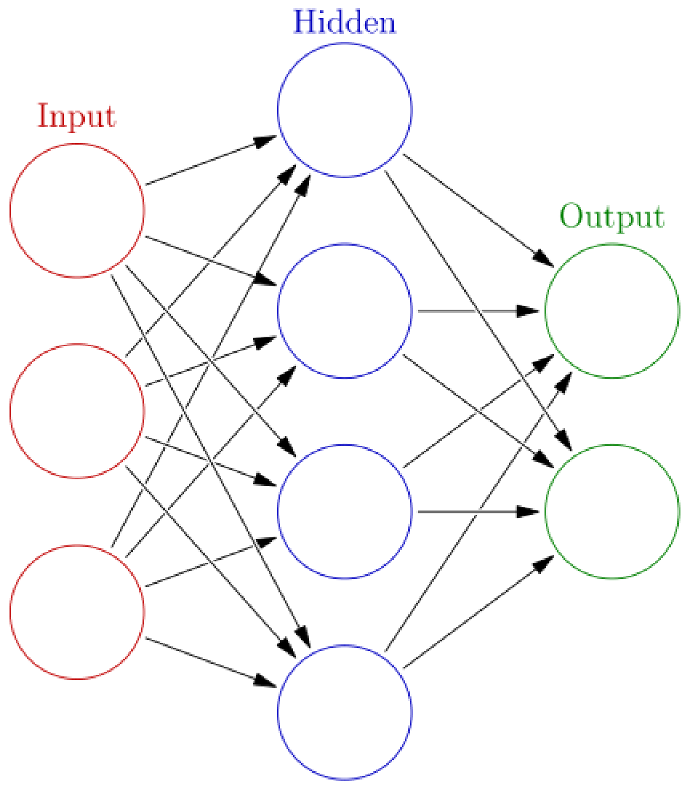


Fig. 2.3. Fully connected neural network [10]

ered to be deep neural networks. In the simplest type of layers, a fully connected layer activation of each neuron is calculated by multiplying the activations from the previous layer by weights on the connection between them as shown in Figure 2.3. Weights of the network are updated using the gradient methods to minimize the cost function of the given problem. Specific types of deep learning architectures used in this research are described in the following sections.

2.4. Self-attention mechanism

Self-attention is a mechanism that transforms one sequence into another where the transformation for a given position in sequence takes into account the other positions of that sequence. The simplicity and possibility to parallelize this operation allowed for the displacement of the recurrent models in the natural language processing areas such as text translation, generation, and a few more.

There are different methods to calculate attention [17]. The popular example is the Dot-Product attention described with formula:

$$Attention(Q, K, V) = softmax(QK^T)V$$

The query Q, key K, value V matrices are created by multiplying the vectors of the input sequence by learnable weight matrices W_q, W_k, and W_v which usually transform the input to

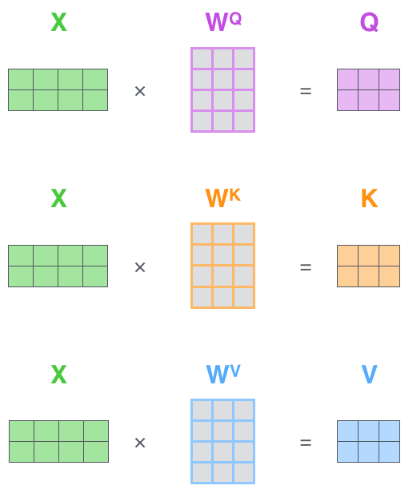


Fig. 2.4. Calculating attention components [1]

lower dimensions. Visualization of the process is shown in Figure 2.4. The word self is in the name because queries, keys, and values are computed for the same input sequence.

2.4.1. Transformer model

Transformers are deep learning architectures that rely on the self-attention mechanism. One transformer layer consists of the self-attention layer described in the previous section, fully connected layer, residual blocks, and normalization. The exact model is shown in Figure 2.5 from the paper “Attention is all you need” [38] from which the original Transformer architecture comes from. Transformer models usually consist of a few Transformer layers stacked on top of each other.

2.5. Convolutional neural networks

Convolutional neural networks [3] are a type of deep learning architecture that in most cases consists of many: convolution, pooling, and non-linear layers which create a representation of the input that then is processed by fully connected layers. This kind of network was inspired by neurons in the visual cortex. ConvNets are state of the art solution for computer vision problems. They are also used for some areas of natural language processing and tasks involving time series. The main building block - convolution layer consists of small filters that slide over the input producing activations map. One step of this process is shown in the Figure 2.6. The filters can be thought of as graphic filters e.g Gradient-Sobel filter for edge detection but with learnable parameters.

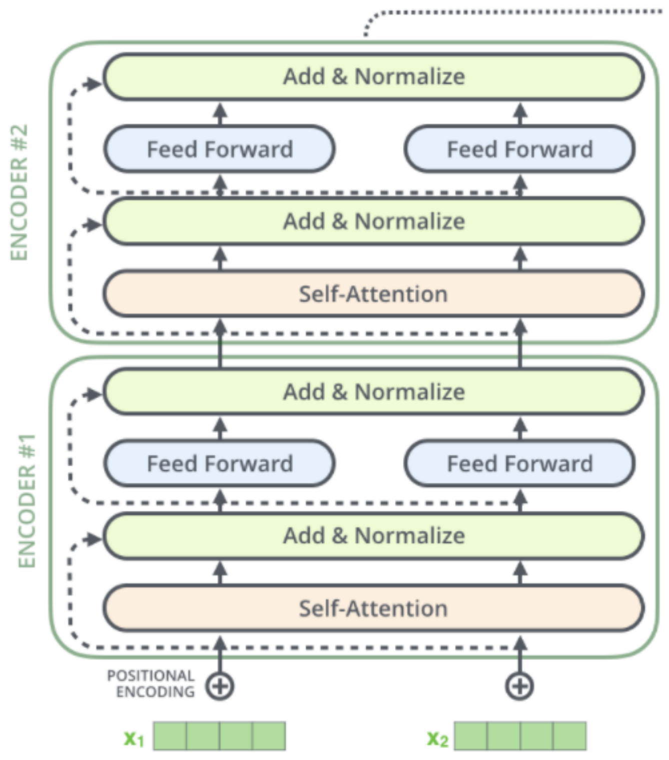


Fig. 2.5. Architecture of the Transformer model [38]

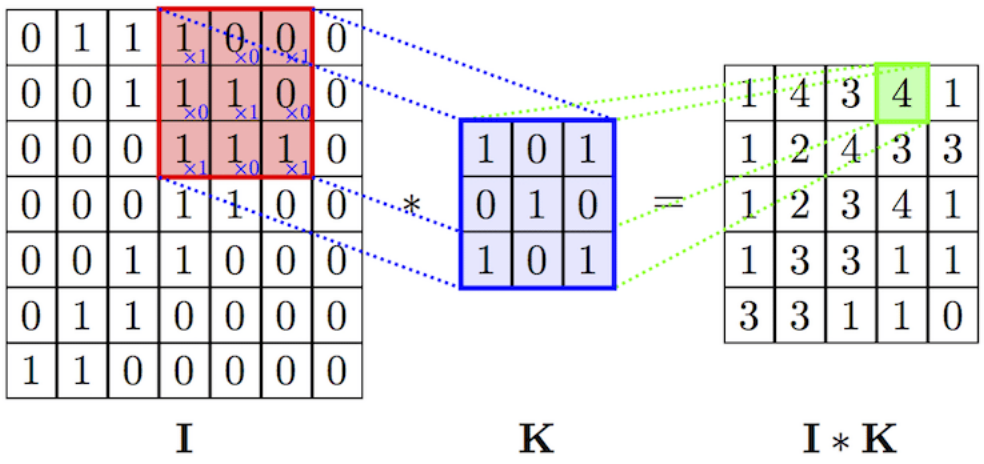


Fig. 2.6. Visualization of the convolution operation [28]

3. DESCRIPTION OF THE ENVIRONMENT (JAN PLISZKA)

In this chapter, the authors describe the Google Research Football environment and match representations it provides. They also explain the choice of the specific representation used in the experiments.

The Google Research Football project contains a reinforcement learning environment based on the engine of the open-source game Gameplay Football [31]. The goal of the game is to win the football match by controlling the players of the team. The project was developed in Google Brain's research team for studying reinforcement learning.

The environment implements the Env interface from the OpenAI Gym module [4] which is commonly used in the reinforcement learning field. The Env interface requires the implementation of three methods:

- reset(self): Reset the environment's state and return observation on which agent decides which action to take.
- step(self, action): Execute passed action in the environment. Returns (observation, reward, done, info) tuple. The 'observation' is the next state observed by the agent. A 'reward' is an advantage in the latest state and 'done' is a boolean flag indicating the end of the current episode. The 'info' usually contains a dictionary with any additional data that did not fit into the previous variables. This method represents the concept of reinforecement learning shown in Figure 2.1.
- render(self, mode='human'): Returns or shows present state. The default “human” mode will render a graphical representation of the environment.

This simple interface enables to easily create the reinforcement learning training loop and is widely used in many other environments.

3.1. Match representation in Google Research Football

A match in a GRF environment, like a real football match, consists of two halves. Each half consists of 1500 steps for a total of 3000 steps. Figure 3.1 shows the initial state of the environment. The starting point for the second half looks analogical but the other team starts with the ball. Compared to turn-based games or games where a psychological factor may be important, there were no signs that the starting player had an advantage [42].

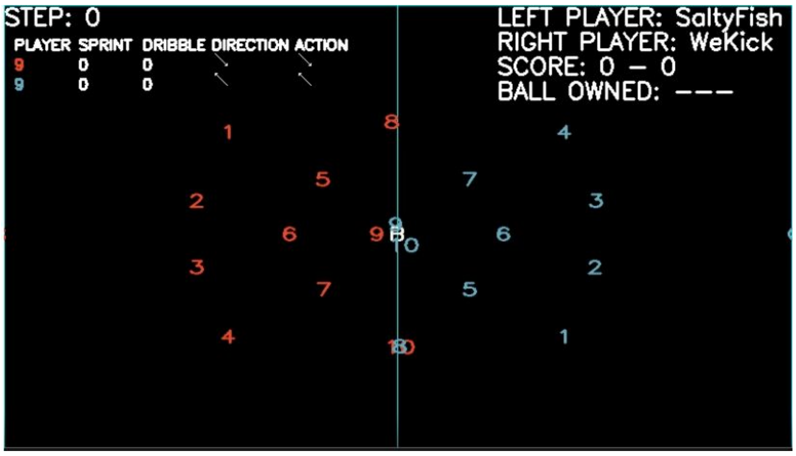


Fig. 3.1. Initial state of the GRF environment

At each step the environment returns the observation. The raw observation from the environment is represented by a dictionary containing information about:

- position and the velocity of the ball and both team players,
- the current owner of the ball,
- score,
- number of frames until match ends,
- screen represented as RGB 1280x720 frame.

And a few other auxiliary pieces of information can be found in the project's documentation.

The Google Brain team also wrote three observation wrappers that can be used to convert raw observations into a simpler representation which is more suitable for most RL algorithms. The most unsophisticated is the "Simple115StateWrapper" which represents the observations from the game as a single vector of length 115. The vector contains the most important information such as positions and velocity vectors of players and the ball. The simplicity of that approach allows the use of more unsophisticated deep learning models and accelerates the RL agent training.

Second "SMMWraper" returns four 72 x 96 binary planes as a game observation, where "1" corresponds to the presence and "0" to lack of the object at the position. The first two planes show the positions of the players from both teams. Third plane stores the ball position and the last position of the currently active player.

The last wrapper returns the observation closest to the observation we receive as humans playing this game. It is a grayscale representation downscaled from 1280 x 720 to 96 x 72 pixels.

The latter two representations require the model to be able to process spatial data. What is more, since they represent only the current frame there is no information about the motion of the environment. There are mainly two solutions to this problem. One way is to use the model that can process the sequential data and capture the change between the observations. The other approach is to return the concatenation of the few frames as an observation so the model can reason about the motion in the environment.

For the purposes of this work, all experiments were conducted using the “Simple115” representations to keep deep learning models less complicated and to reduce overall training times.

4. HAND-CRAFTED AGENT REPRESENTATION (SZYMON ŻEBROWSKI)

In this chapter, the authors in detail described the implementation of the handcrafted GRF agent representation and they were looking closely at statistics obtained.

The concept of this method is to create an agent representation by hand - by gathering the statistics of the game between chosen players, similar to those collected in real football games. Then create a regression model able to predict the probability of winning the game by one of the created agents using concatenated statistics for both in such a way that inverting the vector will give the probability of winning the game by the other player. The description of creating real probabilities needed for regression is presented in Section 6.1. The method also assumes checking various regression methods to find the one most suited to the collected statistics. Regression training specification can be found in Section 4.1. Algorithms of particular statistics collection are presented by Table 4.1 with statistics and the description of what it is and how could be gathered from GRF match representation.

The main drawback of that method is the lack of scalability, in the sense that it cannot be transferred to a different environment, except for other football environments. It also requires both compared agents to play with proxy-player some number of matches, because one can be not enough. But undeniably it has a great advantage. A once created representation can be used for comparison with every other agent, which makes the comparison efficient when it is a need to compare a group of agents.

Table 4.1. Statistics description

Statistic name	Description
win/loss/draw	Percent of matches won/lost/draw with the proxy player.
goals scored	The mean number of goals scored by an agent in a match.
goals lost	The mean number of goals lost by an agent in a match.
on dribble	Mean percent of the time a player spent on dribbling. Observation contains a boolean field describing if an agent in this step uses dribbling. Frames, when the value was set, were counted and divided by total match duration.
on sprint	Mean percent of the time a player spent in a sprint. Observation contains a boolean field describing if the player-controlled by an agent currently is in a sprint. Frames, when the value was set, were counted and divided by total match duration.
red cards	The mean number of red cards an agent got during the match.
yellow cards	The mean number of yellow cards an agent got during a match.
fouls	The mean number of fouls an agent made during a match. This is obtained by calculating how many times a player is taking a free or penalty kick. Offsides, as it is in football rules, are also counted in.
shots on target	The mean number of shots that ended up as goals or were saved by the goalkeeper during a match.
shots off target	The mean number of shots that ended up as a corner in a match.
passes	The mean number of successful passes made by a player during a match. Observation describes which team is owning the ball. We assume that a successful pass is a sequence of ball-owning states where at the beginning ball is owned by player A of the first team, then for some time nobody owns the ball, and then player B of the same team owns the ball.
failed passes	The mean number of failed passes made by the player during a match. Observation describes which team is owning the ball. We assume that a failed pass is a sequence of ball-owning states where at the beginning ball is owned by the first team, then for some time, nobody owns the ball, and then an opposing team takes the ownership.
possession	The mean percent of the time a player is owning the ball. Statistics gathered from observation. Frames, where nobody owns the ball, are not taken into account when counting the percentage.

A detailed description of the implementation of collecting features(statistics) can be found in Section 6.3.1.

4.1. Training process

Three different regression model types were trained on the pairs generated from the subset of 15 agents. Then models were evaluated on the pairs created from the rest of the 5 agents to measure how well the model can generalize. The linear, Random Forest, and Support Vector models described in subsections of section 4.1 were trained in that manner. For comparison purposes, the dummy regressor that predicts always the mean was also evaluated on the validation set. The results of these experiments are described in chapter 8.1. The random forest and SVR models were trained with hypermarameters shown in the Tables 4.2 and 4.3. There are no hyperparameters for linear regression method.

Table 4.2. Hyperparameters used for random forest model

Hyperparameter	Value
number of estimators in the tree	1200
minimum samples per split	2
minimum number of samples at leaf nodes	1
maximum number of features took into consideration for a split	square root of all features
maximum depth of the tree	30
use bootstrapping	False

Table 4.3. Hyperparameters used for SVR model

Hyperparameter	Value
epsilon	0.05
kernel function	radial basis function kernel
regularization parameter	1.0

4.2. Dataset analysis

Before the usage of collected data in practice, the exploratory data analysis to verify if the prepared dataset is valid was made. The authors tried to understand those statistics and find out which one carries the most information.

To increase the volume of training data, the dataset has been mirrored - training data contains not only matches of player A against B but also player B against A. It means some metrics can be understood as collected at home and away (further described with the suffix ”_home” and ”_away”). The model task is for predicting home player winning likelihood, so the correlation with this metric will be subjected. Wanting to look at the relationship between collected features and home win probability a heat map was created - a visualized representation of information with a marked absolute correlation between standardized characteristics. As one can observe in Figure 4.1, there was some correlation between searched probability and collected statistics, but was not very large. The statistics denoted with the prefix ”opponent_” are describing the proxy player.

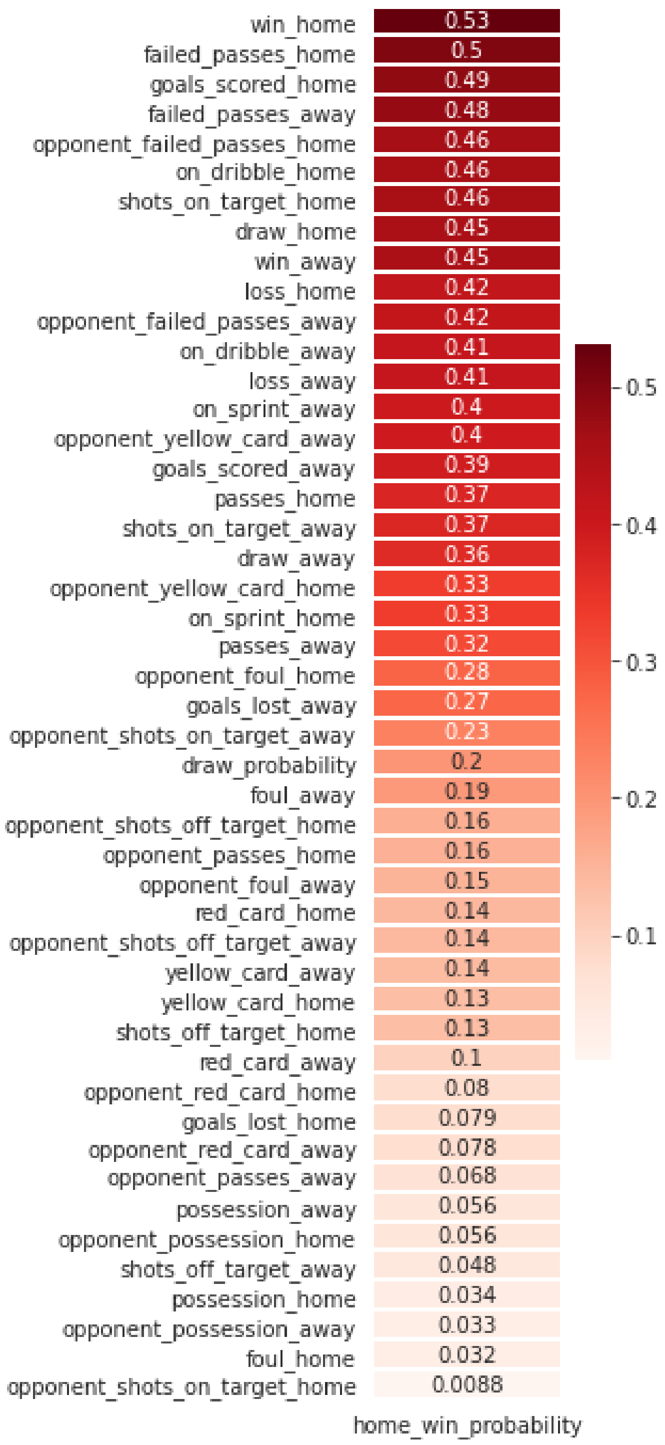


Fig. 4.1. Heatmap representing the absolute correlation between collected metrics and winning probability

Surprisingly, the metrics that seemed important like an opponent's shots on target and those connected with ball possession, turns out to have a small correlation with home win probability. Also as was not expected statistics such as failed passes for both agents and both their proxy opponents, have great potential to be useful in prediction. On the other hand, in the correlated statistics group, unsurprisingly first place took *win_home*, which refers to the percent of matches won with the proxy player.

Example relation of two the most correlated values with the subjected probability for each possessed agent was presented in the Figure 4.2.

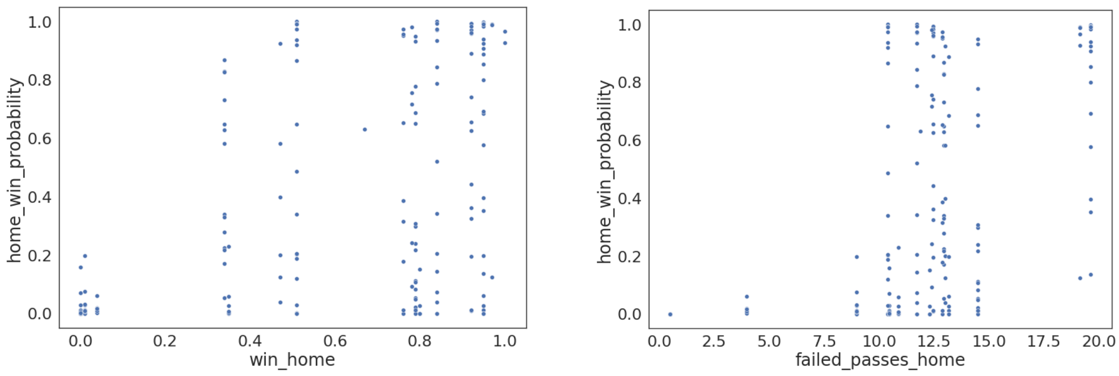


Fig. 4.2. Example relation with home winning probability

The charts showed that the correlation, although it did exist, was not easy to spot. It was seen that if an agent constantly loses with the proxy it would probably lose also against an opponent agent. Interesting was that agents with a low number of failed passes also were losing. What is more, those with the greatest number of failed passes are likely to win. This fascinating relationship, as confirmed by empirical research, may result from that the better agents more often decided to pass, as result they often missed.

Moving further, the authors decided to check how the most relevant (correlation with home win probability greater than 0.45) features were linked with each other to verify the potential of prediction.

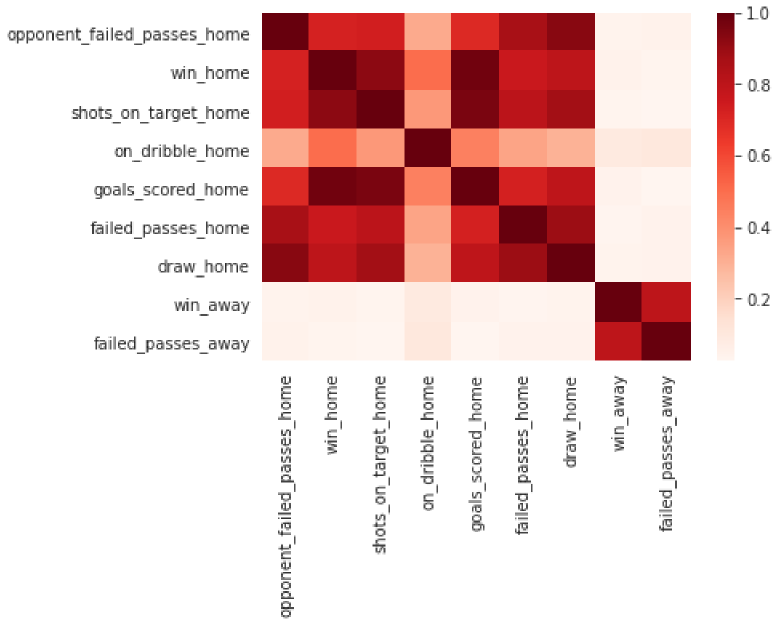


Fig. 4.3. Heatmap of correlation between most relevant metrics

Unfortunately as could be seen in Figure 4.3, the majority of those relevant statistics were strongly correlated with each other, which means that they carried similar information. But there were also some like *win_away* and *on_dribble_home* with a much smaller correlation, which could have a positive impact on the quality of prediction.

Summing up, this analysis showed that the calculated statistics have sense and carry meaningful information. What is more, some statistics i.e. those with absolute correlation lower than 0.05, can be marked as irrelevant, giving the least information and potentially not used for prediction to not generate noise.

5. END-TO-END WINNING PROBABILITY PREDICTION (JAN PLISZKA)

In this chapter authors presented two methodologically and architectonically different neural network models, which were able to learn how to predict winning chance probability and achieve satisfactory results in the trial training.

This method is straightforward in terms of the process, but complicated if it is about the designed model's architecture. The idea is to develop a neural network that takes the match of both of the compared agents played versus the proxy player and returns the correct probability of winning the match by one of them. Two different architectures for that purpose were designed.(5.1.1)(5.1.2) General outlook is presented in Figure 5.1.

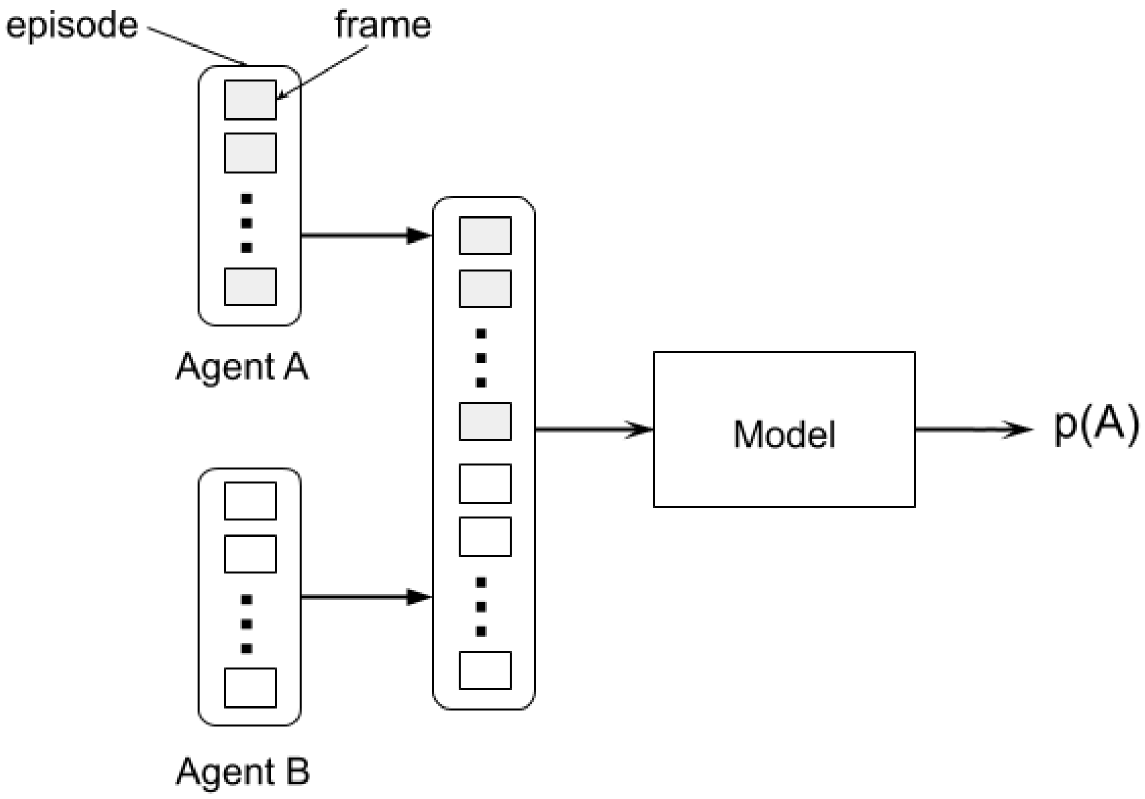


Fig. 5.1. Architecture of end-to-end solution

What is important, the method, if successful, can be transferred to any different competitive environment. It also has the feature that the number of matches needed for comparison grows linearly in terms of the size of compared set.

5.1. Models architectures

5.1.1. End-to-end ConvNet architecture

The model consists of two parts. Encoder based on the ConvNet, responsible for encoding and embedding the match to a smaller dimension. The Google Research Football match is a sequence of frames. Using one-dimensional convolution layers enables to build abstract information from the smaller parts of the match. Because of the pooling layers the data also shrinks in size. The second part is the classifier which outputs the winning probability based on encoder output. Before these parts, there is also a concatenation operation that joins the matches of the agents for which the winning probability is predicted. The detailed specification was presented in Tables 5.1 and 5.2, for encoder and classifier respectively.

Table 5.1. CNN encoder model architecture specification

Component	in channels	out channels	kernel size	stride	padding
Convolution	115	64	2	1	0
ELU activation	—				
Convolution	64	64	2	1	0
ELU activation	—				
Average pooling	—		3	3	0
Convolution	64	128	5	2	0
ELU activation	—				
Convolution	128	128	5	2	0
ELU activation	—				
Average pooling	—		3	3	0
Convolution	128	256	5	2	0
ELU activation	—				
Convolution	256	256	5	2	0
Average pooling	—		3	3	0
Tanh activation	—				
Flatten layer	—				

Table 5.2. CNN classificator part architecture specification

Layer	**input size**	**output size**
Fully connected	3328	512
ELU activation	512	512
Fully connected	512	512
ELU activation	512	512
Fully connected	512	512
ELU activation	512	512
Fully connected	512	256
ELU activation	256	256
Fully connected	256	128
ELU activation	128	128
Fully connected	128	32
ELU activation	32	32
Fully connected	32	1
Sigmoid activation	1	1

5.1.2. End-to-end transferred learning Transformer architecture

Initially, the Transformer in the architecture was to be used identically as CNN described in Section 5.1.1. But all attempts to verify the operation of the model by overtraining a significantly reduced data set ended up as a failure. In most cases, the model was only able to predict a dataset's mean probability value. Due to the fact that training end-to-end Transformers being able to predict values close to an expected likelihood turned out to be too demanding, the decision to use a transfer learning was made.

In the pre-training task, the issue of feature extraction from an agent was compared to the problem of finding an image similarity. Technically the point was to create a projection of training samples (images or in this problem matches) in multidimensional space where contextually similar data will be placed close to each other and those divergent will be spaced apart. In image similarity problems commonly it is achieved by training CNN using triplet loss A. The process is presented in Figure 5.2. The idea behind this method was to take advantage of triplet loss based training in agent representation creation. In the next step, the pre-trained model was taken and at the end of it, a regressor added. Then performed end-to-end training. It was expected that previously trained encoders would be able to achieve expected results much easier.

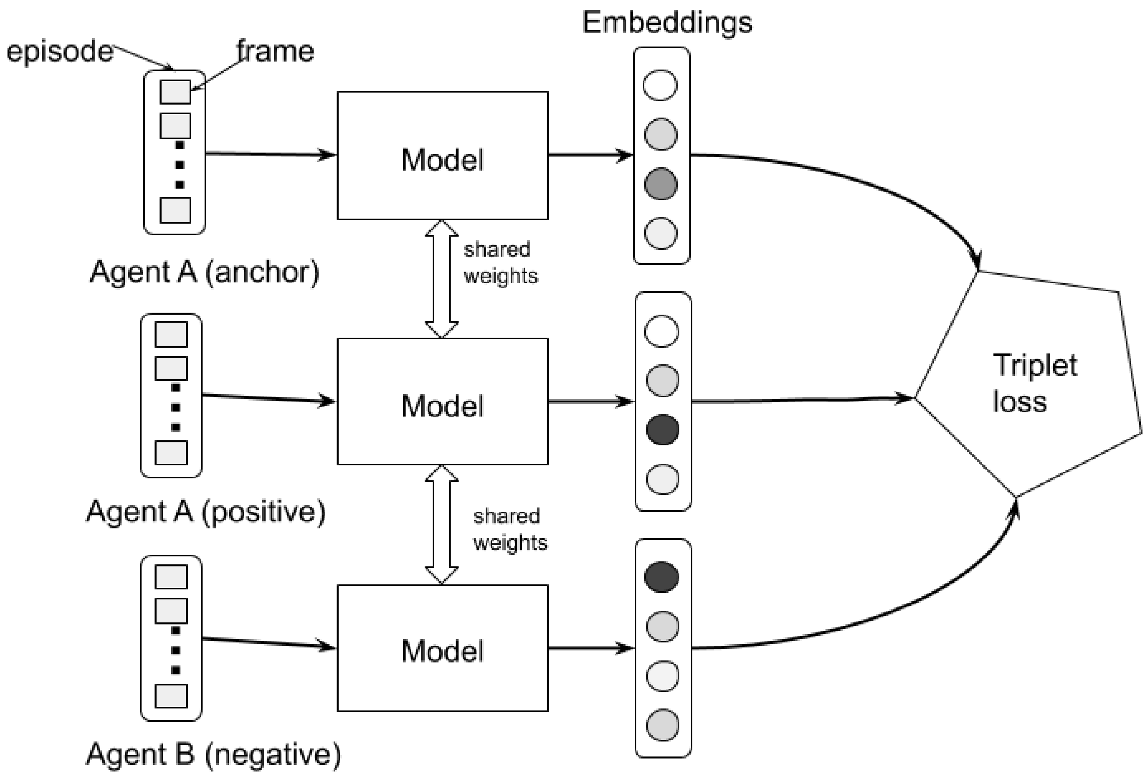


Fig. 5.2. Feature extraction process

Due to the fact that transfer learning was used, the model had been divided into two potentially separated parts. The first - an encoder responsible for embedding creation, only this part is subjected to preliminary training, and a second - regressor, which takes merged outputs from an encoder and returns a probability. An entire process was presented in the Figure 5.3.

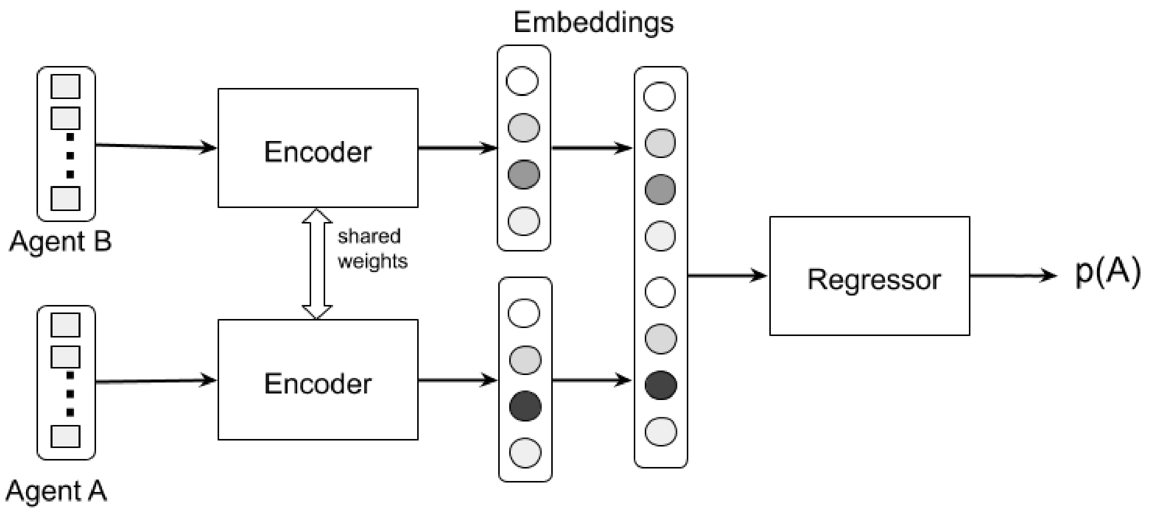


Fig. 5.3. Transformer model outlook

The core of the whole design is an encoder, which is presented in Figure 5.4. It was made of four components, each having different functional purposes. The first was sinusoidal positional encoding(A) which adds to each frame position-dependent signal. It was needed because the Transformer has no notation of the word order in a sequence. Then there was a Transformer encoder itself, which task was to turn the match into a representation with the size of the episode, which describes an agent's capabilities. Finally, the representation obtained was reduced by two last components. Both neural networks with a few linear transformation layers, one of them shrinks a frame and the other brings the vector with reduced frames to the given size output size.

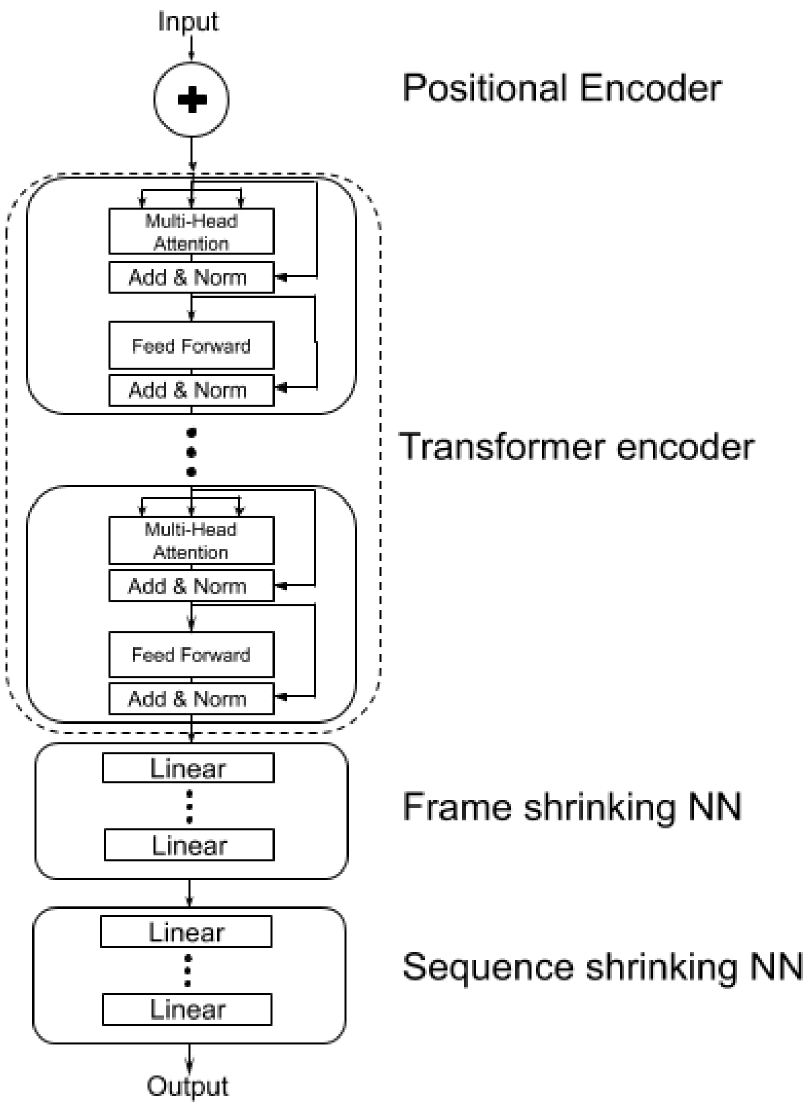


Fig. 5.4. Transformer model - encoder

Regressor in this method was another deep neural network that takes two agent embeddings and predicts the winning chance of the agent whose representation is in the first half of the input vector. Both encoder and regressor detailed specifications including empirically selected parameters are presented in Table 5.3.

Table 5.3. Transformer model architecture specification

Component	Input size	Output size	Param	Value
Positional Encoder	(3000, 115)	(3000, 115)	Dropout	0.3
Transformer Encoder	(3000, 115)	(3000, 115)	Attention heads	5
			Feedforward dimension	2048
			Dropout	0.3
			Transformer encoder layers	5
Frame shrinking NN	115	1	Hidden dimensions	256, 128, 64, 16
			Activation function	Leaky ReLU
Sequence Shrinking NN	3000	64	Hidden dimensions	1024, 1024, 256
			Activation function	Leaky ReLU
Regressor	128	1	Hidden dimensions	2048, 2048, 256, 16
			Activation function	Leaky ReLU, sigmoid(last)

5.2. Training process specification

Because of the large size of the models and the input data, a batch size during the training had to be limited to even one sample for the Transformer. The training with such a small batch would be ineffective. Due to that a gradient accumulation method was applied, in which instead of updating the model weights after each batch, the gradients are accumulated, and the model is ultimately updated after a specified number of batches based on cumulative gradient. Worth mentioning that collected datasets turned out to be too large to finish an epoch in a reasonable time and what is more crucial the model probably would be already overtrained at the end of that epoch. The decision was made to validate the model more often than at the end of the dataset. The period between evaluation will be hereinafter called an epoch. Used in the training loss functions were a sufficient metric that allowed evaluation of the quality of operation of the models.

Training of the Transformer-based architecture requires the usage of transfer learning. The step preceding the actual training was to teach the model how to create an agent representation. It is called pre-training further. There was not any specified target loss that the model should achieve, because it will be retrained at further steps. The point of pre-training is only to facilitate regression and its result is defined by the result of the entire study. Furthermore, a key thing in the transfer learning approach is to not spoil a partially working model at the beginning of proper training. A gradient formed as a result of random regressor weights would negatively affect the previously trained encoder. To overcome this issue the pre-trained part should be frozen for some time in the final training. Detailed specifications of all end-to-end models training were placed in Table 5.4.

Table 5.4. End-to-end models training specification

Training	Batch size	Gradient accumulation period	Evaluate period	Initial learning rate (LR)	LR step period	LR step gamma	Optimizer	Loss function
CNN	32	-	5907	0.001	10000	0.9	SGD	MSE
Trans. pre-training	1	32	5000	0.001	1000	0.9	SGD	Triplet (margin = 1)
Trans.	1	16	12800	regressor $7 * 10^{-6}$ encoder $5 * 10^{-6}$	3000	0.9	Adam	MSE

Periods are expressed in a number of steps.

6. IMPLEMENTATION OF PROPOSED METHODS (DOMINIK GRZEGORZEK)

In this chapter authors presented how dataset creation was performed and revealed implementation details of both presented methods.

6.1. Target probabilities

Agents used in the research mainly were shared with us thanks to the courtesy of Football ZPP team[34]. What is more, the set was enlarged by a couple of less intelligent agents trained for the needs of this work.

Not only for regression but also for validation purposes a real, target value of probability was required. The only way to gather it is to calculate the relative frequency of winning matches by one of the agents. To gather the targets, probabilities of winning the match, 400 games between each pair of agents were played. The number of matches was picked to ensure with 95% confidence that this probability lies in a $p_{win} \pm 0.05$ interval, where p_{win} is the percent of 400 games won by an agent. The collected at this stage likelihoods were used in every approach introduced. The number of needed matches was calculated with the formula for binomial proportion confidence intervals[40] using the normal approximation. Normal approximation assumes that the errors about the probability of winning follow a normal distribution. Then probability p is estimated by the formula:

$$p = \hat{p} \pm z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}},$$

where $\hat{p}$ is estimated probability, n is the number of played matches and z is the $1-\frac{e}{2}$ quantile of a standard normal distribution, where e is targeted error rate. For 95% confidence z is equal to 1.96. The estimated interval is the widest when $\hat{p}$ equals to 0.5. Inserting that value into the equation and 1.96 for z it is now possible to equate the formula to the wanted interval size and then solve it for n.

For the reason of dataset creation, a decision was made to select 20 agents for the task of data generation. This number is big enough to create a dataset consisting of 380 objects and small enough such that time of running needed simulations would not take longer than one week.

6.2. Leaderboard

Due to the need for player comparison, leaderboard util was created. It is a system allowing automation of match playing and collecting results from them. A leaderboard is represented by Google Cloud Storage Bucket, which is storing players' configuration files and trained models, optionally dumps from matches. The core of the leaderboard is a JavaScript Object Notation file with results from all played matches and the player's Elo rating. A leaderboard was based on the fork of the ZZP Team private leaderboard [34] and expanded.

Leaderboard module consists of following files:

```
leaderboard/
├─ leaderboard-private/
│  ├─ all_vs_all.py - used for playing matches, further described in next
│  │  chapter
│  ├─ collect_statistics.py - collecting statistics from matches using
│  │  statistics trackers
│  ├─ convert_dumps_to_simple115.py - converting directory with GRF dumps into
│  │  representation used by Transformer model
│  ├─ evaluate.py - playing matches between two given players
│  ├─ generate_dataset.py - generating .csv file with data used by handcrafted
│  │  predictors
│  ├─ init_leaderboard.py - creating .json file for leaderboard or statistics
│  ├─ one_vs_all.py - for playing matches, described further
│  └─ play_all_vs_all.py - used for playing matches, described further
└─ utils_leaderboard/
   ├─ common.py - utils for leaderboard and interaction with cloud storage and
   │  environment
   ├─ merge_leaderboards.py - script merging leaderboard and statistics .json
   │  files
   └─ upload_dir.py - upload given directory (or file) to cloud storage
```

Leaderboard script fetches data from the cloud, plays matches between players and uploads results back to the bucket. It can be run in four different versions:

- evaluate, which plays a match between two given players. The time complexity of this script is $O(1)$. Can be used in two modes - standalone=True (which works normally) or standalone=False (as a part of play_all_vs_all.py - a modified version of all_vs_all).
- one_vs_all, which plays matches between specified player and all the other players existing in the leaderboard. Time complexity is linearly proportional to the number of players in the leaderboard, thus the script runs in $O(n)$, where n is the number of players. The script is useful when adding a new player to the existing leaderboard and needs to play a given number of matches with other players to have an equal number of collected data.
- all_vs_all, which plays matches between all players. The time complexity of this script is quadratically proportional to the number of players on the leaderboard.
- play_all_vs_all, a modified version of the script above. The idea behind the script was to make data collection faster by playing more matches at once. The script finds appropriate pairs suitable for playing games and runs a few evaluate scripts as its subprocesses.

The whole leaderboard can be compared to a graph, where vertices are players and edges are matches. In the case of collecting data for each player by all_vs_all, it creates a complete graph, where $\frac{N(N-1)}{2}$ matches need to be played (N is the number of players). Leaderboard architecture makes us assume that each model cannot play more than one match at a time. But at one moment, it is technically possible to have few matches - if it will not be the same players. The problem of choosing appropriate players can be compared to the edge coloring of a graph. It does not care about selecting players already involved in the game, it wants to choose edges in a way that no two incident edges have the same color (played at the same time). Parallelizing the games by finding proper (not necessarily the most optimal) edge coloring resulted in a slight increase in the time of a single match but decreased overall time to play the whole tournament.

6.3. Hand-crafted representation

6.3.1. Statistics

As input data for the estimators the decision was made to use metrics similar to these gathered during real football matches i.e. the number of passes, shots on target, possession, and many more. Metrics were collected during the 100 games played between an agent and built-in bot, which imitates the process of RL agent training. To achieve statistics trackers are needed which will be used as a wrapper on the RL environment based on each step observations. Google Research Football environment does not support collecting matches' statistics, so there was a need to develop it. The principle of operation of the script was the same as this included in the leaderboard (6.2). The Google Cloud Storage Bucket containing all the players for whose matches were collected statistics should be prepared at the beginning (technically that was the same bucket used in the leaderboard). Then the statistics gathering was performed by running the collect_statistics.py, which takes as a parameter the name of a bucket, the name of the result file also placed on the bucket storage, and the number of matches to be played. The algorithm downloads all players, for each pair creates the environment and sequentially plays a given number of episodes for that pair, saving the statistics for each match locally, and after the games of one pair are finished, the data is uploaded to google cloud.

Over a dozen statistic trackers that are used as a wrapper on the RL environment were written. Each statistics tracker is responsible for the calculation of one statistic. The script lets you collect a dump (a full record of the episode) for played matches and uploads it to the remote storage. This functionality is not needed for this method, but it is required for the further described approaches in which predictions were based on those dumps. Both metrics and dumps were collected at once during the 100 games played between an agent and built-in bot which imitates the process of RL agent training.

6.4. End-to-end solution

6.4.1. Dataset preparation

Creation of a large part of the needed dataset - probability values of winning the match for each pair in a twenty element set of agents was already described in Section 6.1. It remained to collect a set of predictors - a representation of a match played against a proxy for each agent. The script collect_statistics.py described in Section 6.3.1 allowed researchers to save a dump of matches during which statistics were collected. Match records had to be made readable for the model. Due to that, each frame of an episode was represented in a simple115 notation in the second version which is present in the Google Research Football environment. The decision was dictated by the size of this representation, which was the smallest possible. The conversion was done by dump_to_simple115.py script, as a result of which dump is saved to a file that consists of a pickled two-dimensional python list of size 3000x115, where the first dimension size represents the length of a match and the second a length of a simple115 representation. For all agents, 100 matches were saved, which creates our dataset.

6.4.2. Regression dataset

One training sample that can be used in this regression task consists of a left player match, a right player match, and the probability that the left agent wins the match. The number of samples existing in the dataset can be calculated by an equation:

$$N = C_p^2 * C_q^2 * 2$$

, where p is the number of agents in the dataset, q is the number of matches every agent played with proxy and C_y^x denotes a number of x-elements combinations in y sized set. Factor 2 in the equation exists only if mirroring is used. In this case, it means changing the order of matches transferred to the model, and calculating the inverse probability. Having a hundred matches for each of twenty agents, when applying a mirroring allows to create exactly 1'881'000 samples, but the key thing was to split the dataset when selecting the agents. It allows ensuring that a model learns how to extract valuable features from the match instead of simpler classifying the agents. It would be less demanding to judge the probability if the model already saw an agent in any match. So the dataset was split into a training, containing 15 agents and a validation one that contains the rest 5 of agents. Agents were divided consciously, making sure that each set was differentiated in level. The procedure finally made the final training and validation dataset, respectively 1'039'500 and 99'000 samples large.

6.4.3. Triplet loss dataset

Taking to the dataset every agent that later will be used in a regression task could affect the validation result, on the other hand, a dataset consisting of only fifteen players may turn out to be too small. Because of that, for this task, there was a need to create a completely separated training dataset based on previously unseen agents. The sample in this process requires to include two matches of agent A, which serve as an anchor and positive value, and a single match of agent B - the negative input. The collection size can be calculated using a formula:

$$N = V_p^2 * C_q^2 * q$$

, where exactly like in the previous equation p is the number of agents in the dataset, q is the number of matches every agent played with proxy, V_y^x and C_y^x denote a number of x-element variations without repetitions or combinations respectively, in y sized set. A natural language equation can be expressed as follows: for each ordered pair of players, take all pairs of first player matches and combine them with all the matches of the second. As it collected about 54 agents and exactly 10 matches of each playing with proxy, the whole dataset consisted of 1’287’900 samples. Finally, the dataset was limited by taking only one randomly picked a negative match for each pair instead of all ten, so the data size decreased 10 times.

6.4.4. Training implementation

The deep neural network models were designed and trained with the PyTorch library, currently the most popular choice among deep learning researchers. The library provides an implementation of all commonly used layers, cost functions, optimizers, or other tools useful for the developers such as schedulers, metrics, dataset interface, and many more.

The simplified version of the training loop is presented in Figure 6.1.

```
for epoch in epochs:

    for i, batch in train_dataset:
        output = model(batch"["features])
        loss = criterion(output, batch"["targets]
        optimizer.step()

        if i % log_frequency:
            evaluate(model)
    logger.log_metrics()

    scheduler.step()
    evaluate(model)
    logger.log_metrics()
    save(model)
```

Fig. 6.1. End-to-end models training loop

This generic version was used for the training of the models based on the Transformer’s encoder as well as those using the CNN. It was also suitable for the setups with different cost functions by setting the criterion parameter, TripletMarginLoss for the pre-training and MSELoss for the end-to-end solution.

Worth mentioning is the evaluation inside the dataset iteration. This choice was dictated by the size of the datasets described in previous sections and slow processing time due to the large model size which resulted in sporadic logging about the current training status. Adding more frequent logging and evaluation enabled better monitoring of the training process.

Models were evaluated on the validation dataset described in the Section 6.4.2. The agents used in that set were independent of the training ones. Models that were tested on the dataset created from unseen combinations of matches of the agents which the model was trained on lacked generalization. Using different agents helped to find the overfitting problem which may be observed in the plots in the next section.

7. APPLICATION AND USER INTERFACE DESCRIPTION (SZYMON ŻEBROWSKI)

In this chapter, the authors describe the application made for evaluation of their prediction algorithms divided into two parts: web application overview and deployment environment. They present used technologies and workflow connected with the production of software.

7.1. System requirements

The application should allow users to upload their trained Google Research Football player models into a system based on leaderboard (6.2). With a given player, the user should obtain prediction results for matches between his and other players existing in the system, basing on matches with the proxy player, as well as get real results from matches played between them. The application should also be able to play match between players and display video dump from them. Functionalities should be wrapped in friendly and simple user interface.

7.2. System overview

7.2.1. Technologies

The application was written using Django - an open-source Python framework for creating web applications. Django allows users to conveniently create websites realizing model-template-view architectural patterns. Main framework distribution provides modules such as authentication system, administrative interface, and built-in mitigation for popular web attacks such as SQL injection, cross-site request forgery, and cross-site scripting. Default Django project can be extended with applications, each with its own URL routing system. The framework provides object-relational mapper mediating between data models written as Python classes and relational databases. Django officially supports five databases: PostgreSQL, MariaDB, MySQL, Oracle, and SQLite. The system is relying on other technologies such as:

- PostgreSQL - advanced, open-source relational database system supporting both relational and nonrelational querying [23].
- Redis - efficient and highly scalable in-memory key-value storage which can be used as a database, message broker, or cache [26].
- MinIO - cloud storage system and object store capable of storing unstructured data in buckets [20].
- Celery - open source asynchronous task queue used as a mechanism to distribute tasks across multiple threads. Thanks to that, web applications do not have to handle requests that take a long time such as data processing and fetching data from external services by themselves [5].
- Docker - one of the most popular container runtime interfaces [7].

The application was designed to be containerized, split into microservices, and run using Docker on the Kubernetes platform. One of the key assumptions was to reuse the leaderboard system for collecting data and playing matches between players. To limit external network traffic and provide better portability, the authors decided to switch from previously used Google Cloud Storage to MinIO storage placed inside the cluster.

7.2.2. Containerization

Containers are applications isolated from the host's OS without the need for emulating the whole hardware layer and operating system in contrast to pure virtualization - they use resource isolation mechanisms provided by the Linux kernel such as cgroups and namespaces. Containers run only processes of given application which increases their efficiency in hardware resource utilization which in the case of distributed applications operating on many virtual machines gives noticeable savings. Another benefit of using containers is the ease in application deployment: by having a container image - a static file including executable code which can be compared to a recipe for running a container, the user can deploy it to any machine without having to configure the environment and install application dependencies. Each container has its own space in memory and a separate network interface with its own private IP address. Containers work independently from each other as long as user does not explicitly indicate the dependencies between them.

Containers are stateless by their definition. That means that between container restart, all data inside that container is gone. In some cases, applications should keep their files to remain persistent. Docker has a concept of volumes - directories on disk which are mounted to containers and used to persist data while the container is gone. Such a directory can be mounted to a new container providing all the data saved to that volume.

7.2.3. Kubernetes

As the number of containers and independent projects grows, maintaining and administering running applications can become problematic. Additionally, if you want to expose more instances of the application on new machines, manual configuration of connections between containers can be quite a challenge. In 2014 Kubernetes project was made available to the public. Kubernetes is a platform for managing, automating, and scaling container applications based on many years of Google's experience in managing high-scale services combined with best practices and ideas developed by the community. It connects available machines in a cluster, a group of interconnected computer units that work together to provide an integrated work environment. In Kubernetes there are 2 types of nodes (computers working in clusters):

- Master - its role is managing cluster, making major decisions about the cluster, detecting, and responding to various events. Should only run containers which are crucial for the cluster.
- Worker - its role is to run user applications in containers.

Master nodes consist of the following control plane components:

- API server - extensible module which is a classic REST API allowing for interaction with cluster. Cluster components communicate with the API server by performing some action or checking status. It uses the Role-Based Access Control (RBAC) mechanism for assigning roles to components giving access or restrictions to certain actions on API.
- Controller-manager - deals with determining the state of our cluster - when the user reports, for example, a desire to scale the container from 1 instance to 3, the controller will lead to such a state. From the level of logical division, each controller is a separate process, the following controllers can be distinguished:
  - Node Controller - responsible for recognizing and responding to situations, where the node becomes unavailable.
  - Replication Controller - responsible for upkeep the correct number of Pods (Kubernetes object described in further part) for each object of Replication Controller type in the system.
  - Endpoints Controller - it joins Pods and Services together creating Endpoints objects.
  - Service Account & Token Controllers - it creates default service accounts used by Pods and API access tokens for the new namespaces.
- Scheduler - responsible for assigning nodes to new pods based on resources available on these machines and user preferences.
- Etcd - critical element of the cluster, distributed key-value storage. While the rest of the cluster is stateless, etcd stores all data about the cluster. It uses the Raft [25] consensus algorithm, where few machines try to agree on some value. It works when over half of the machines in the cluster are available.

On each node in the cluster additional components are running:

- Kubelet - agent responsible for running containers created by Kubernetes inside Pods.
- Kube-proxy - network-proxy managing network rules on nodes thanks to which networks inside and outside the cluster can communicate with Pods.
- Container runtime - software responsible for running containers, for example, Docker.

General outlook of Kubernetes cluster components is presented in Figure 7.1.

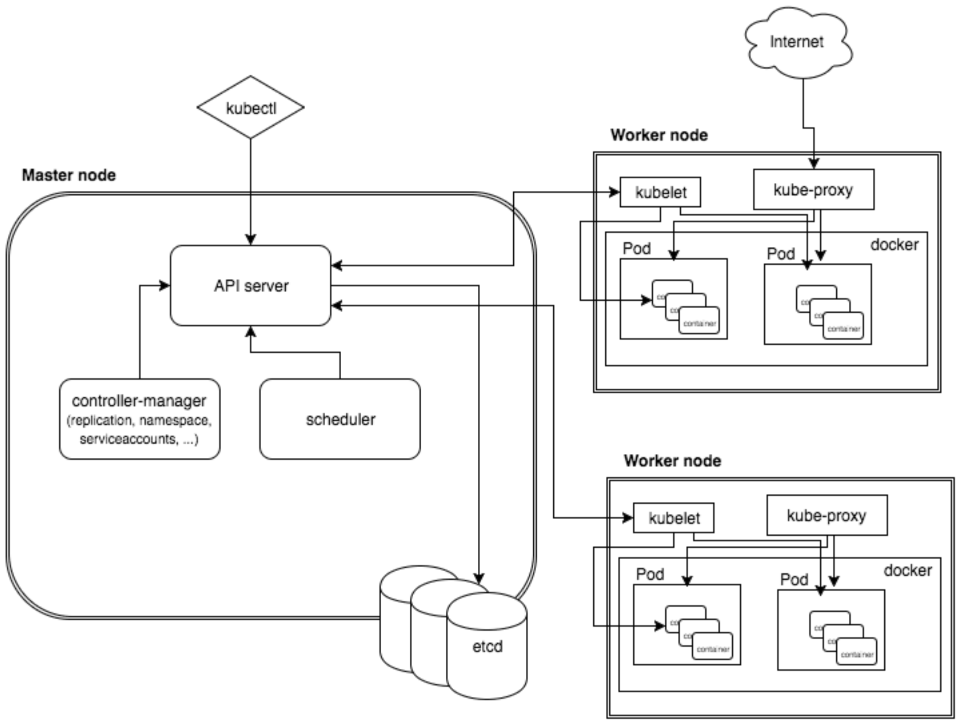


Fig. 7.1. Kubernetes cluster components [35]

Kubernetes can be deployed on bare metal platforms, public cloud, or for developing purposes on one node using minikube.

Applications running on Kubernetes are called workloads. Containers running on the cluster are packaged in Pods - deployable units running one or more containers. Pods can be managed by other workload resources such as Deployments, ReplicaSets, StatefulSets, or Jobs:

- Deployment - object providing declarative updates to applications. Manages the application's life cycle - such as the number of desired pods and way of updating the container's image.
- ReplicaSet - object maintaining a stable set of Pods running at any moment, it guarantees the availability of a given number of Pod replicas.
- StatefulSet - similar to Deployment, but designed to manage stateful applications, for example, databases. Guarantees persistent identifier for Pods (in case of Deployments, names are unique but not persistent - after rescheduling, Pod gets a different name).
- Job - creates one or more Pods and ensures that they successfully finish their tasks.

Kubernetes applications are exposed using Service resource. It is a way to enable network access to a Pod. Inside a cluster, Pod can access Service using its name - it is resolved by DNS plugin. To access Service outside the cluster without having LoadBalancer which will redirect external traffic, Service has to be NodePort type - ports on a node will be proxied into Service.

Kubernetes introduces a few abstractions on the storage layer, supporting manual and dynamic provisioning. In the manual approach, PersistentVolume (PV) - a piece of storage used for creating smaller volumes for workloads - is created by cluster administrator and later consumed by PersistentVolumeClaims (PVC) - request for specific storage. PersistentVolumeClaim is similar to Docker's volumes. They define the size of storage needed by workload resource and access mode in which they will be mounted, e.g. ReadWriteOnce (mounted inside one Pod, allowing reading and writing operations), ReadOnlyMany (mounted inside many Pods, allowing them only reading operations), ReadWriteMany (mounted inside many Pods, allowing each of them both types of operations). In dynamic provisioning, the administrator creates StorageClass, which is used for creating volumes using external provisioners. Provisioner determines what plugin should be used for provisioning PersistentVolumes. Plugins represent various storage providers, e.g CephFS, Network File System (NFS), Ceph's RADOS Block Device (RBD), and public cloud providers' internal storage types, e.g. Google's GCEPersistentDisk or Microsoft's AzureDisk.

For more information about Kubernetes, visit [2].

7.2.4. Architecture analysis

The main component of the application is a web application written in Django. It uses the PostgreSQL database for storing its models. The core of the application's prediction system is reusing the authors' leaderboard system 6.2 and Google Research Football application. Both of these run in a single container. For playing matches and collecting data for predictions Celery is used. It allows running long-lasting background tasks without suspending the main thread of the application. When it comes to performing such an action, the main application thread produces messages describing work to be done in the background which are stored in Redis (Celery requires some kind of message broker which in case of application is Redis) and fetched by Celery workers. Some of the data - for example, player's configs and match results are stored on Minio. This data needs to be accessible for the leaderboard system which is based on bucket storage. Furthermore, the leaderboard and GRF environment are using it to persist data they produce, which is later fetched by the web application and used for generating predictions or displaying information. The principle of operation of the application is presented in Figure 7.2.

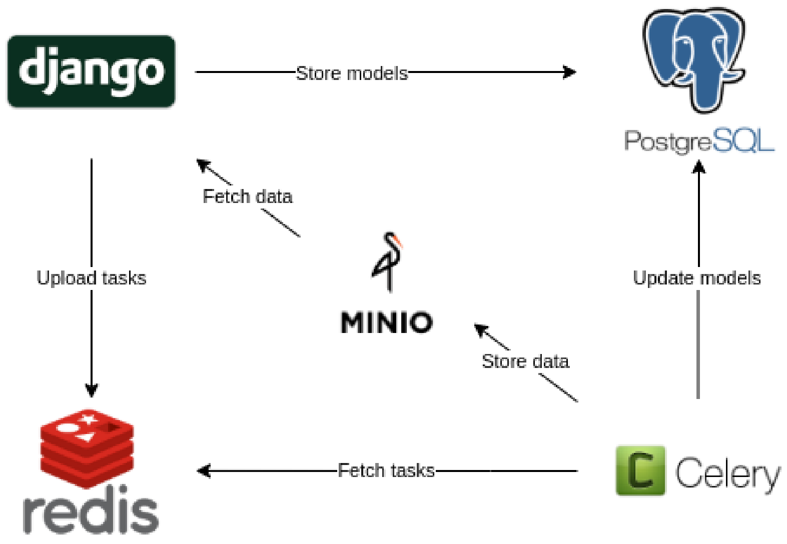


Fig. 7.2. Schema of application workflow

The application operates on three data models which are presented in Figure 7.4 and 7.3.

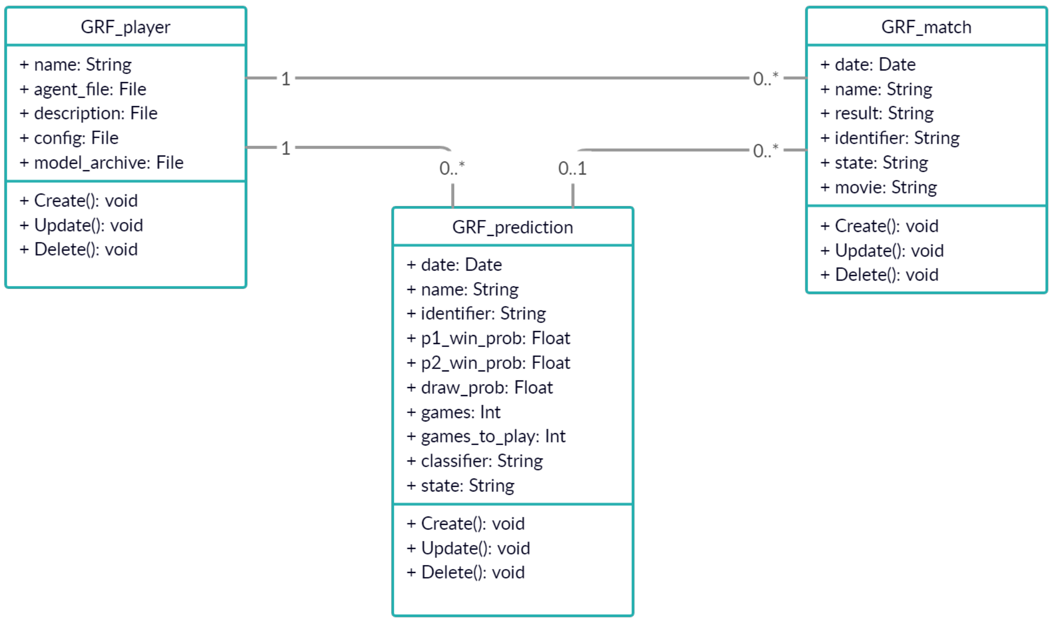


Fig. 7.3. Class diagram

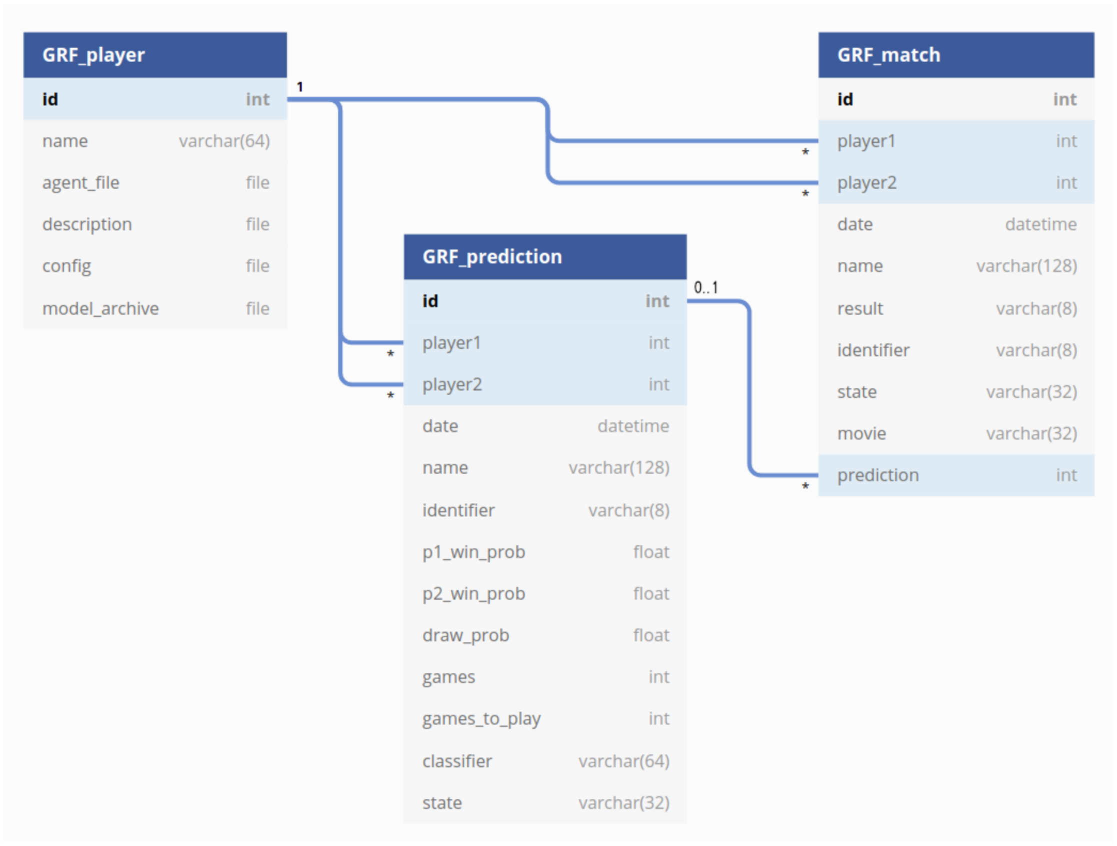


Fig. 7.4. Database diagram

7.3. Project overview and implementation details

Django imposes certain application structure, below is a short description of the application's files:

```
grf-app/ - container for project named "grf-app"
├── grf-app/ - Django project directory
│   ├── asgi.py - an entry point for WWW servers compatible with Asynchronous
│   │   Server Gateway Interface
│   ├── celery.py - module integrating Django with celery
│   ├── settings.py - configuration of the Django project
│   ├── urls.py - URLs for the Django project
│   └── wsgi.py - an entry point for WWW servers compatible with Web Server
│       Gateway Interface
├── grf_predictions/ - application's directory
│   ├── static/ - directory containing Cascading Style Sheets and Images for the
│   │   web application
│   ├── transformers/ - directory containing scripts and checkpoints for
│   │   Transformer model
│   ├── admin.py - registration of application's models to the admin site
│   ├── apps.py - application's config
│   ├── forms.py - forms used for model creation
│   ├── model_utils.py - various functions connected with models used in the
│   │   application
│   ├── models.py - application's models
│   ├── tasks.py - function run by Celery
│   ├── urls.py - URLs for application
│   ├── utils.py - functionality connected with running matches and MinIO access
│   └── views.py - application's views
├── media/ - directory for various uploaded or generated media
├── templates/ - HTML pages templates using Jinja
├── manage.py - Django command-line script
├── model_rf.pkl - Random Forest regression model for sklearn
└── model_svr.pkl - SVM regression model for sklearn
```

Application has three main features: uploading players, creating matches between players and creating predictions. These functionalities will be further discussed in this paragraph.

Uploading players

The first action taken by the user should be uploading the player in leaderboard format which is used for further evaluation. The Player's uploading form expects one or more files to be uploaded. Each file should be a .zip archive containing player checkpoints, config, agent script, and description, which are saved as a player model to the database as well as normal files in the media/ directory. File upload to bucket storage is done as an asynchronous job using Celery. Model's detail view displays the content of its files.

Due to cascade delete set on foreign keys containing player's id, before deleting player, list of all resources connected with that object is shown, asking the user for confirmation of his decision.

Each player model archive should have the following structure:

```
player_name.zip - archive file with model, named after a player
└─player_name/ - directory containing data, named after a player
    ├─checkpoint/ - directory containing model weights
    ├─description.json - file containing model's specification and any
    │ additional data (should contain URL to neptune.ai experiment from
    │ training if applicable)
    ├─config.json - file with the configuration for the model (should contain
    │ the location of checkpoints)
    └─player_name.py - agent script used by GRF environment
```

Creating matches between players

User can choose two players from the list of existing players and create a match between them. Each user-created match saves the match result and video dump from the environment. Playing matches takes dozens of seconds, it can not be realized in the main thread of the application because the application user could feel the hang of the website. Running matches is handled by the controller supervising Jobs creation and deletion. Jobs are creating Pods running GRF environment image with commands. For each match running in Pod following actions are taken:

1. Run init_leaderboard.py script to initialize JSON with leaderboard.

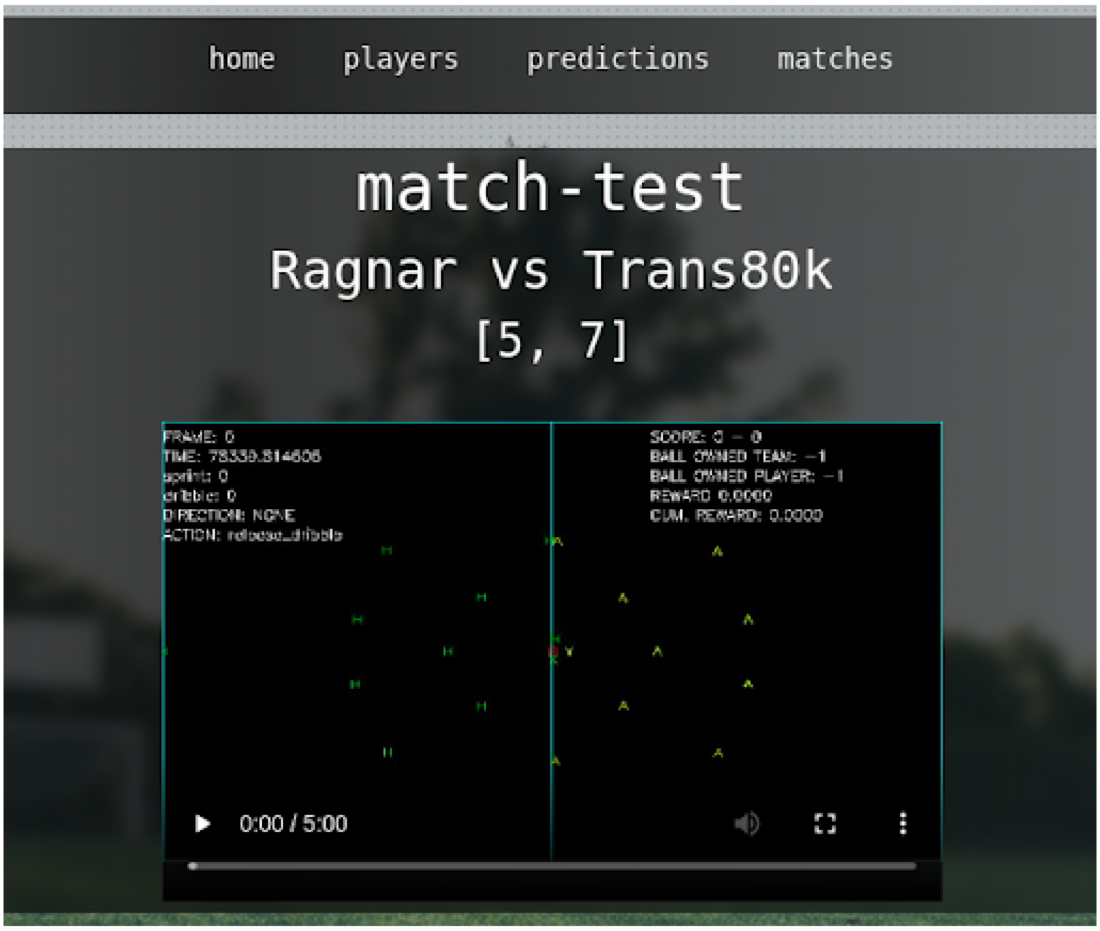
2. Run evaluate.py script to create a match between two given players and save results to leaderboard file created in the previous step.

3. Run GRF gfootball/replay.py script to create video dump in .avi format based on dumps saved from a match.

4. Upload video dump to bucket storage.

With the completion of these steps, Job is finished and Pod is being terminated. Later on, JSON with results is downloaded from bucket storage and parsed for getting a match result. Fetched video in .avi is converted to .mp4 format using FFmpeg library and saved in the application's media directory.

In match detail view user can notice the match result and video generated by the environment, what is presented in Figure 7.5.



match-test

Ragnar vs Trans80k

[5, 7]

```
FRAME: 0                              SCORE: 0 - 0
TIME: 78339.814606                    BALL OWNED TEAM: -1
sprint: 0                             BALL OWNED PLAYER: -1
dribble: 0                            REWARD 0.0000
DIRECTION: NONE                       CUM. REWARD: 0.0000
ACTION: release_dribble
```

Fig. 7.5. Match detail view

Creating prediction

The user selects two players for which system creates win probability using statistics collected from a given number of matches (1-100) with proxy players. User can also verify results by providing a number of matches which will be played between these players returning the actual number of wins, draws, and losses with a calculated confidence interval. User can choose between available algorithms: Random Forest regressor [36], SVR [8] and Transformer [38]. Transformer (5.1.2) and handcrafted models (4) require different input data, so the user has to decide which type of predictor will he use at the moment of prediction creation because of the procedure of collecting data and making prediction. With saving prediction to the database, two Jobs are created in a similar way as for playing matches - because we have two players for which we collect statistics. In each of these jobs the following steps are done:

- Run init_leaderboard.py script to initialize JSON with leaderboard.
- Run collect_statistics.py script to gather statistics from matches with the proxy player.

If the chosen classifier is the Transformer, additional actions are taken:

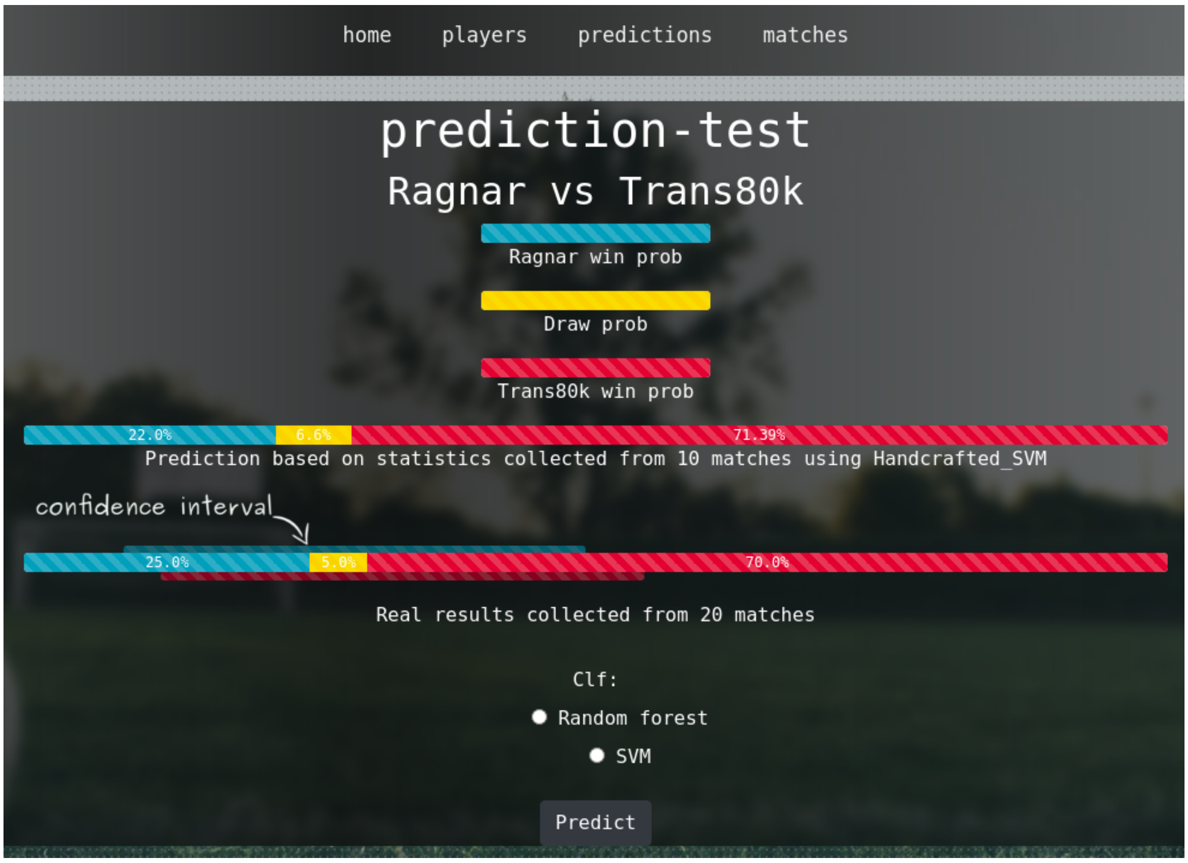
- Run convert_dumps_to_simple115.py script to prepare data for a prediction model.
- Upload these dumps to bucket storage.

The next step is to generate a dataset from collected metrics. Another Job is created, in which the following actions are taken:

- Run merge_leaderboards.py script to connect leaderboards with collected data into one file.
- Run generate_dataset.py script to prepare .csv file which will be used for prediction.
- Upload created .csv with data to bucket storage.

The last step is making prediction based on the chosen model. If user chooses any of hand-crafted predictors, .csv data consisting of rows with results of two players are loaded into pandas dataframe, its columns are reorganized to be in the same order as during predictor's training, and each row is loaded into prediction model (to achieve both players win probability against each other). If the user decided to use Transformer, downloaded dumps with simple115 observation's representation are loaded into the Transformer model to return a tuple with win probabilities. Then, in both ways, results are parsed to generate draw probability. For making predictions, the winning probability for each player is calculated. If their sum is less than 100%, the difference between that and 100% is draw probability. If the sum of probabilities is greater than 100%, both of them are scaled with the following formula $p_x = p_x/(p_1 + p_2)$ where x is the number of player.

If the user decided to create matches for result verification, they are handled in the same way as usual matches, except they do not generate video and do not show up on the list of created matches. Choosing any of the handcrafted models allows the user to switch the current regressor and rerun prediction with a different model, what is visible in Figure 7.6.



Prediction based on statistics collected from 10 matches using Handcrafted_SVM

Real results collected from 20 matches

Fig. 7.6. Prediction detail view

When hovering on the progress bar with a confidence interval, a popup with precise values of interval appears. The confidence interval was estimated using Python statsmodels library based on results from played matches. The calculation is done using the ‘beta’ method from statsmodels - Clopper-Pearson interval based on Beta distribution often called an ‘exact’ method, which works accurately when the probability of an event is equal to 0 or 1, what could happen during evaluating matches (when one player wins all matches) and is more precise than Normal approximation method [19].

7.3.1. Deployment description

The application is designed to be run on Kubernetes using 5 Deployments:

- Grf-app - containerized application running server.
- Celery - containerized application running celery worker as one process.
- Postgres - PostgreSQL database for application.
- Redis - by default one instance of database.
- MinIO - one instance of bucket storage.

Due to the application's design, more precisely downloading and uploading files in the background, Pod with grf-app should have common disk space with Celery workers. It could be done by putting these two containers inside one Pod, but it blocks an important advantage of application - scaling Celery workers. For that, nfs-server-provisioner was installed using Helm - package manager for Kubernetes [11]. It creates the NFS-server pod and NFS StorageClass, allowing to create PersistentVolumeClaims with ReadWriteMany (RWX) access mode. With this shared volume mounted to applications and Celery's Pod, changes to the filesystem in one of them causes changes in another.

Django makes connection with database, Redis, and MinIO using Services they expose. Postgres and Redis Services are ClusterIP types, which means they are available only within the cluster. MinIO service is configured as NodePort, making it available from outside the cluster. The same applies to the application's service.

By default, Celery Pod cannot create necessary API resources because of Role-Based Access Control settings. To change that, ServiceAccount and appropriate rules to make operations on Job and Pod resources were created.

Application scales very well depending on resources. With every 2 CPUs and 2GB of memory, another Celery worker can be added - what means faster execution of predictions and matches because they do not wait in a queue so long.

Figure 7.7 pictures application's schema inside Kubernetes Namespace.

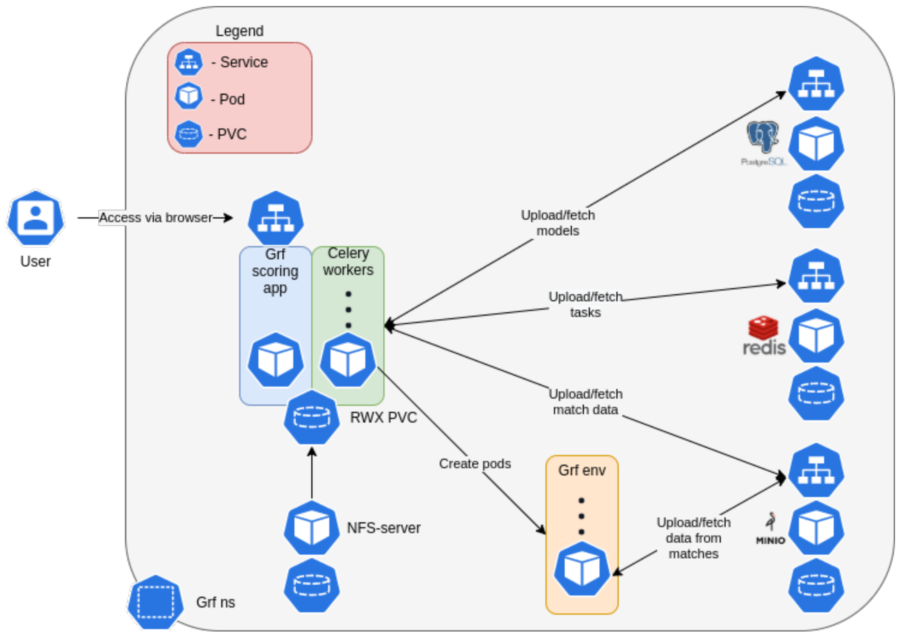


Fig. 7.7. Overview of applications component inside a cluster

The user should run the application on a cluster or using minikube - one node Kubernetes cluster running locally on the computer. Along with the application code, its components' deployment configs are delivered.

```
k8s/ - directory with all configuration files for Kubernetes deployment
├── grf-app/ - components for Django application
│   ├── celery-worker-deployment.yaml - Celery Deployment
│   ├── grf-app-deployment.yaml - Django application deployment
│   ├── grf-app-pvc.yaml - PersistentVolumeClaim for Django application
│   ├── grf-app-service.yaml - definition of NodePort Service for application
│   └── grf-app-migration-job.yaml - job performing necessary migrations for
│       database
├── postgres/ - components of PostgreSQL database
│   ├── postgres-secret.yaml - secret containing user credentials for database
│   ├── postgres-service.yaml - ClusterIP Service definition
│   ├── postgres-statefulset.yaml - definition of StatefulSet for database
│   └── postgres-pvc.yaml - PersistentVolumeClaim for Postgres database
├── rbac/ - definition of necessary RoleBasedAccessControl mechanisms
│   ├── celery-worker-role.yaml - definition of Role - rules on API server for
│   │   Celery
│   ├── celery-worker-role-sa.yaml - ServiceAccount definition for Celery
│   └── celery-worker-rolebinding.yaml - RoleBinding of Role and ServiceAccount
└── redis/ - components of Redis database
    ├── redis-deployment.yaml - deployment of Redis database
    ├── redis-service.yaml - ClusterIP Service for Redis
    └── redis-pvc.yaml - PersistentVolumeClaim for Redis database
```

MinIO and NFS-server are deployed using Helm charts. GRF environment with leaderboard module and application are accessible as images on Dockerhub - public image registry - respectively as GDNRF/grf and GDNRF/grf-scoring-app.

The application was deployed and tested on local machines using minikube as well as on Google Kubernetes Engine cluster with three e2-highcpu-4 machines. Minimal resources requirements: 4 CPU and 4GB of memory.

8. EXPERIMENTS (DOMINIK GRZEGORZEK)

In this chapter, the authors showed the training course of the prepared models. They also carried out an evaluation and compared the results of achieved models. Every experiment was performed using DGX Station[21].

8.1. Regression analysis of statistics representation

This section focuses on a comparison of the scikit-learn regression models which learned on the handcrafted statistics collected during matches with a proxy agent.

The baseline model is the dummy regressor which always predicts the mean of the training data. It achieved a 0.0 R^2 score on the train data which is the expected value taking into consideration the definition of the R^2 score(A).

Random forests due to the use of ensembles may have problems with learning when the dataset is small which in this case is quite possible. Models achieved very good results on the training set, with good hyperparameters is able to learn the whole data but achieves very poor results on the validation set which manifests itself by high MAE (mean absolute error)(A).

The linear regression model is too simple to achieve a satisfying score on the training data. What is more, validation R^2 score and mean absolute error suggest that validation samples are from beyond the distribution the model learned, resulting in evaluation metrics being out of expected bounds.

Finally, the SVM model achieved very good results on the training data and did fine on the validation set. With a simple empirical hyperparameters tuning model achieved less tam 10% MAE on the validation set meaning its predictions were on average 10 percentage points from the actual probability.

Results achieved by all of the regression models are presented in Table 8.1.

Table 8.1. Regression analysis of statistics representation results

	Training	Validation	
Metric	R^2 score	R^2 score	MAE
Dummy	0.0	~ 0.0	0.33
Random forest	0.93	0.08	0.28
Linear regression	0.82	$-1.87 * 10^{22}$	$4.47 * 10^{10}$
SVR	0.98	0.87	0.10

8.2. Training of end-to-end models

The training was carried out on a Tesla V100 graphic card (16GB of local memory), and its specification is described in Section 5.2. Reported were only those trainings that had given the final result.

8.2.1. CNN

The model based on CNN showed convergence in the very first few epochs, which is visible in Figure 8.1. After that, a deep learning model starts to overfit the training examples. It may be caused by the specific type of the dataset that the neural network is trained on. The matches from which training data had been created were combined into pairs which lead to the model "seeing" the same match more than once during each epoch. The best validation loss achieved by this type of architecture during the training is 0.045.

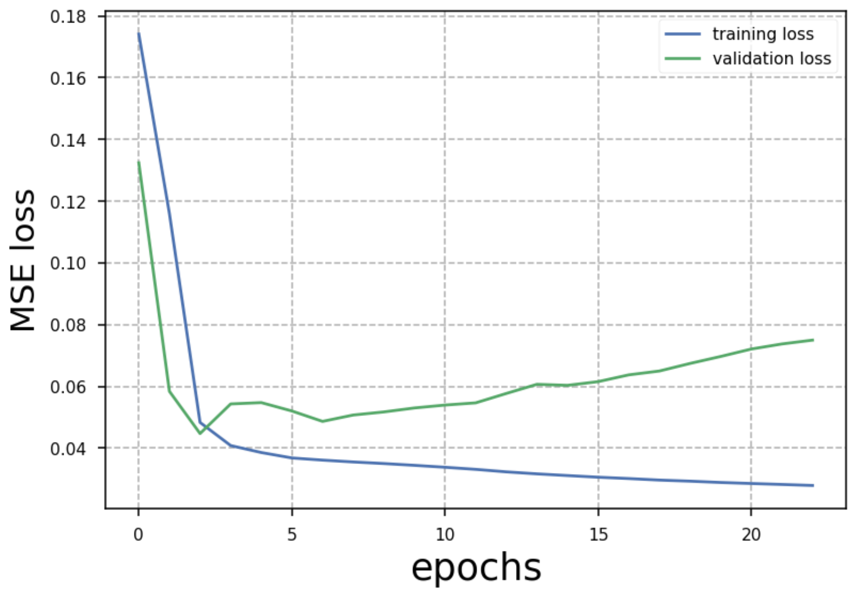


Fig. 8.1. Mean square error loss during CNN training

8.2.2. Transformer

The pretraining of an encoder lasted over 6 hours, during this time the model was able to see the entire dataset once. As can be seen in Figure 8.2 an encoder was very quickly over-trained losing the ability to generalize. The lowest validation loss the model achieved already in the 5th epoch. After that point with the decrease in mean of training loss, the validation loss was increasing. It could mean that a model was able to improve itself in recognizing the agent's match and its specific behavior, but not in extracting the general features that could represent a larger group of agents.

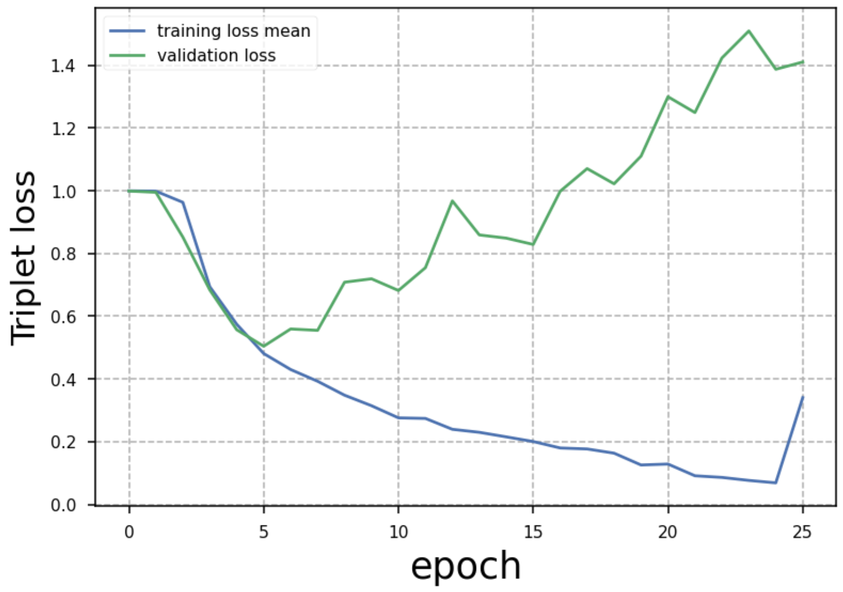
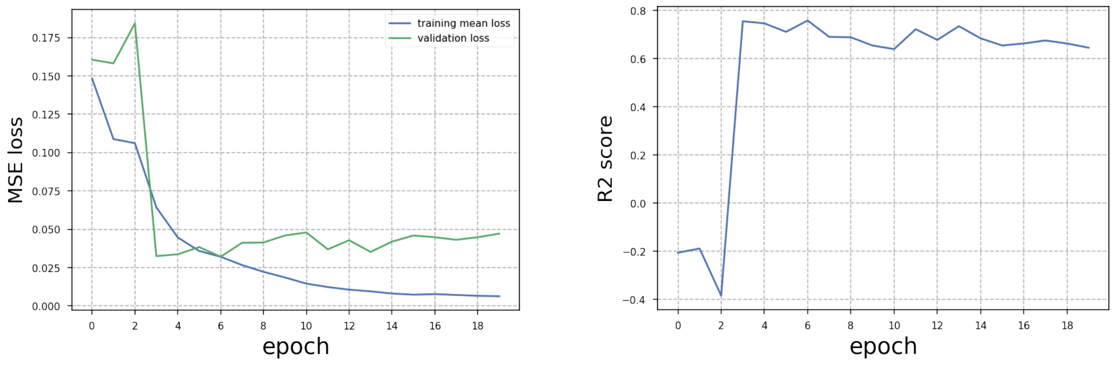


Fig. 8.2. Triplet loss during Transformer pre-training

The model which did the best with validation data and achieved 0.504 loss was taken for further training. It was certainly not an encoder that at this stage was able to create easily comparable agent embeddings, but it unquestionably provided a good foundation for further training. When transferring an encoder to setup with a regressor, the key thing was to freeze the model for some time at the beginning of the training. So as the training started, the encoder part was trained with zero learning rate until validation loss increased. During this training, an additional metric was also collected - an R^2 score (A). As a loss function mean squared error was used. The learning rate was reduced every 3000 by 10% during the training as shown in Figure 8.4.



(a) MSE loss during training

(b) R^2 score during training

Fig. 8.3. Metrics during Transformer training

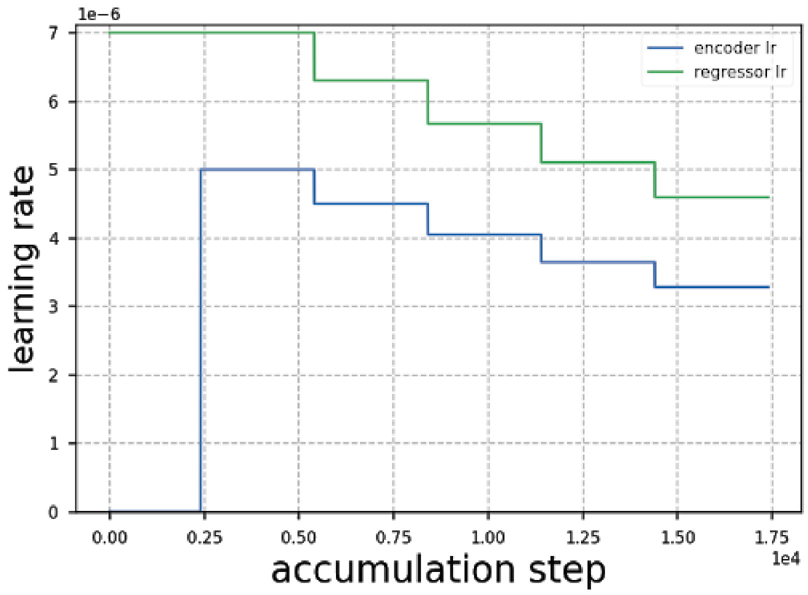


Fig. 8.4. Learning rate during Transformer training

The training was aborted after about ten hours because the validation loss was not improving anymore. Transformer encoder started to learn after 2400 accumulation steps, which corresponds to the 3 epochs. And this is also visible in Figure 8.4. After the third validation the first part of the model was unfrozen, because the model's loss then was greatest which means that the regressor started to overtrain itself. As presented in Figure 8.3 the greatest R^2 score was achieved in the fifth epoch and was 0.75. At the same time the validation loss was equal to 0.032. The model saved in that epoch creates our solution and was assessed in later sections.

8.3. End-to-end model evaluation

Due to the stochasticity of the environment and agent itself, a single match is not a sufficient benchmark for comparison when using the end-to-end models, which was additionally verified in Section 8.3.1. For this reason, evaluation requires more matches, which can be used through pooling. That could be applied before and after the model and if it is about the Transformer also between the encoder and regressor. Taking the average from match representation as could be foreseen only worsened the results because it was in fact blurring all meaningful data with random noise. But pooling an output or agent's embedding improved results in comparison to those obtained during the training where no pooling was used. That is presented in Table 8.2, which was created by model evaluation on the whole validation dataset. For 100 matches, prediction was made once, and for 5 matches results were collected by making 20 predictions on disjoint subsets of players' games and averaged.

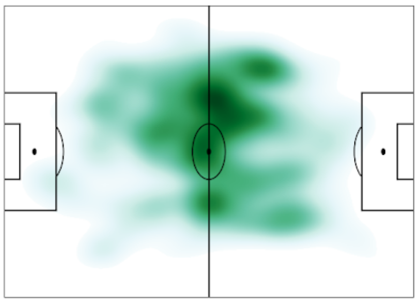
Table 8.2. End-to-end models evaluation results

	Pooling applied on	Number of matches	Loss			R^2 score			MEA		
Pooling function			Avg	Median	Max	Avg	Median	Max	Avg	Median	Max
CCN	Output	5	0.023	0.026	-	0.83	0.8	-	0.11	0.12	-
		100	0.018	0.015	-	0.87	0.90	-	0.11	0.09	-
Trans.	Output	5	0.03	0.03	-	0.79	0.76	-	0.12	0.13	-
		100	0.019	0.019	-	0.86	0.86	-	0.1	0.1	-
Trans.	Embedding encoder output	5	0.034	0.029	0.02	0.85	0.75	0.86	0.11	0.13	0.1
		100	0.019	0.02	0.02	0.86	0.85	0.85	0.1	0.11	0.1

In prediction based on multiple matches, every match was parried only once. Taking the product of compared players matches would give n times more samples, but as checked, it did not improve results, only proportionally extended the execution time. Compared were different types of pooling - a mean, median, and additionally in embedding - max function, as it has no sense applied on output probability. Surprisingly the greatest score was achieved by taking median values of CNN network predictions. The absolute error of this model was on average 0.09. That creates the best solution in terms of prediction quality on the validation dataset.

8.3.1. Analysis on out of distribution samples

To ensure the reason why adding pooling improved the result was the fact that even matches between the same pair of agents can vary greatly, an auxiliary trial has been carried out. An example analysis on a match, that in every pair in which it existed, gave meaningfully higher mean square error confirmed the supposition. It turned out that it was a match in which the player named ‘eli_whole_last’ unexpectedly tied with the proxy player, while in all other situations it was winning. What is more, the game was played on half of the compared agent’s pitch for a long time, which is illustrated in the heatmap comparison in Figure 8.5 visualizations. The positions of the goalkeeper were ignored.


(a) Usual match of 'eli_whole_last'

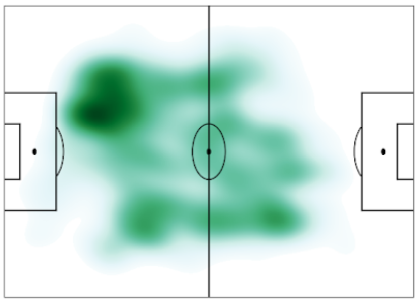
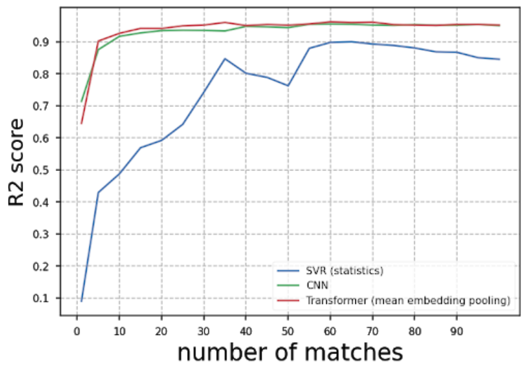

(b) Match where 'eli_whole_last' tied

Fig. 8.5. Visualization of eli_whole_last's positions during match

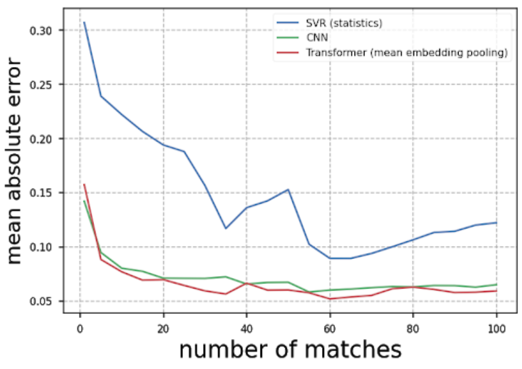
The discovered case confirms the need for using more games in comparison to compensate for the negative influence of the outliers - matches that were played differently than usual.

8.4. Result comparison

For the purpose of comparison of developed methods, an additional test dataset was created by running a leaderboard (6.2) with 5 agents, where each pair played a hundred games. The data obtained this way was used for comparison of the quality of prediction depending on the number of matches taken into account. The experiment was carried out on three best-in-class trained models: SVR taking as input statistics, Transformer which pools encoder outputs, and CNN, which takes median from calculated probabilities. The aim of the study was to verify what is the lowest number of matches to play to get the optimal solution for each method and examine the models once again on another dataset. Results are presented in Figure 8.6.



(a) Comparison of R^2 score for selected models



(b) Comparison of MAE for selected models

Fig. 8.6. Comparison of score and loss for test dataset

Analyzing both the score and the mean absolute error you can see an advantage of the end-to-end models in the correctness of predictions, which on the validation dataset was not that clear (8.3). In this trial both end-to-end models, with the optimal number of matches, gave an impressive maximal score of 0.96 and on average they were wrong about 0.06. In contrast to

MAE achieved on the validation dataset (8.3), at that time it was Transformer model, not CNN, that provided the lowest error. That made it impossible to say which architecture actually delivers better predictions.

What is more, basing on the experiment it was undemanding to conclude that end-to-end models needed definitely fewer matches to achieve the result close to optimum. Only about 15 matches were enough to get the score about the maximal value for both, while SVR needed more than twice matches more. Further examination was performed to check the time of execution of all three algorithms. Only the time of comparison was measured, the times of playing matches with the proxy player were not included. The performance test for each model was performed in the execution environment.

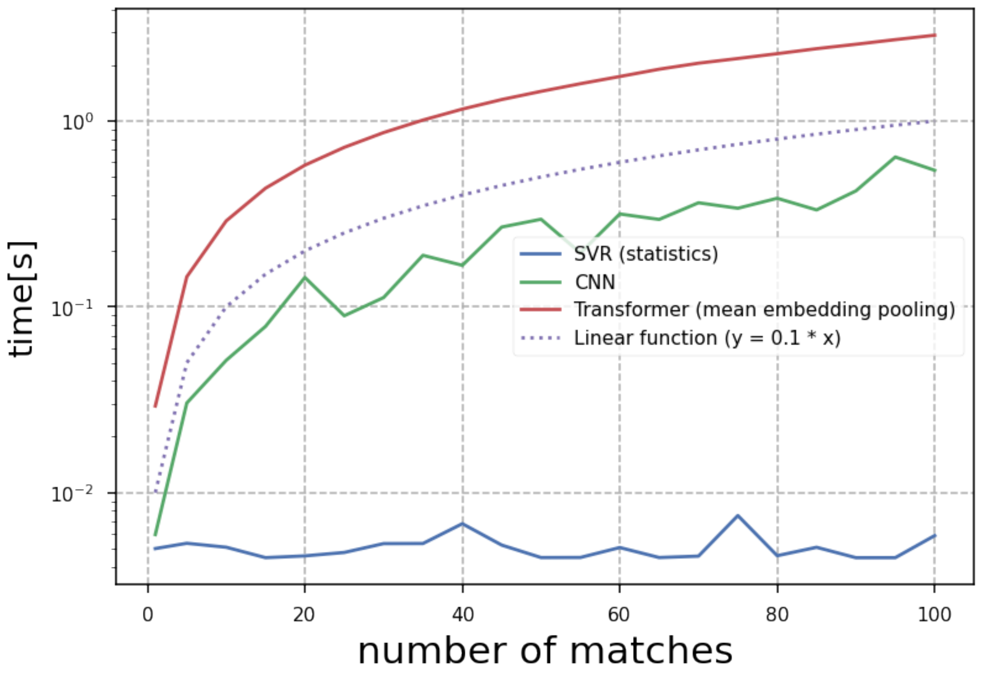


Fig. 8.7. Comparison of predictions times for selected models

As shown on the Figure 8.8, times of execution for both end-to-end linearly increases proportionally to the number of matches taken into account in the prediction. Transformer needed the longest time to return the probability since the initial phase of the experiment. Also can be spotted, that the time of comparing statistics does not depend on the number of matches. This property results from the fact that the model is fed by the value of the average statistics per match.

8.4.1. Out of distribution agent impact study

Delivered comparative algorithms should be able to judge the probability independently whether the game is played by artificial intelligence, repetitive algorithm, or even the human because it is not testing any reinforcement learning specific features. The target of this experiment was to check how the solutions would deal with an agent with unusual style of playing. For that reason, the test dataset used in the method comparison described in 8.1 was extended by playing matches between present agents and a player named bot_gfootball. It was the only agent of all used in the entire work that was not a reinforcement learning agent because it was the default bot in which the GRF is equipped. The results of the study were presented in Figure 8.8.

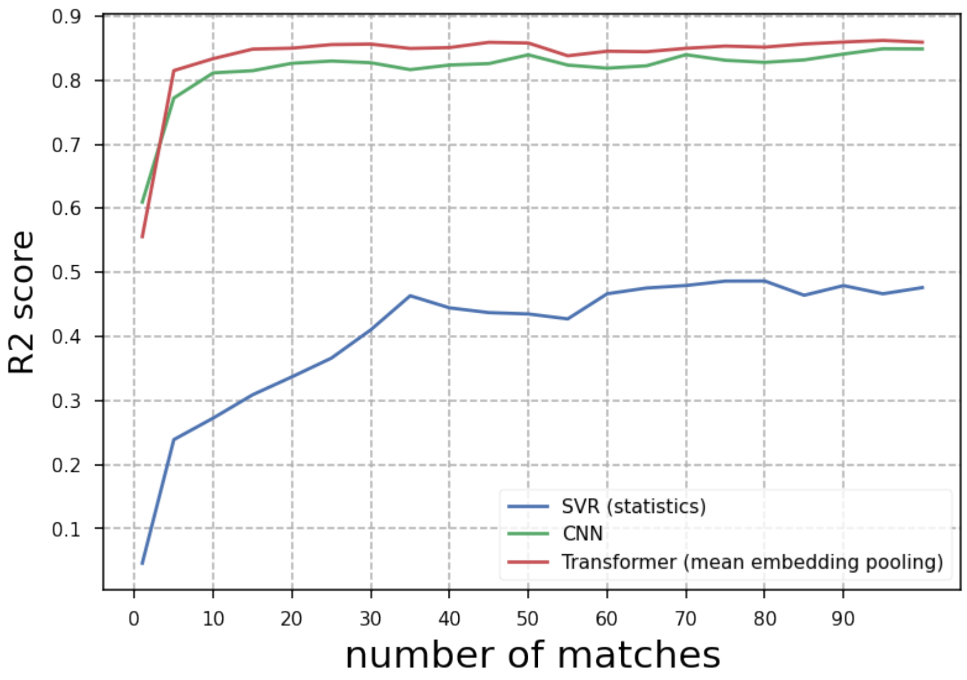


Fig. 8.8. Comparison of score for test dataset with ’bot_gfootbal’

Adding an out of distribution agent to the dataset alarmingly worsened achieved results, especially if it is about the statistics method. Regression on hand-crafted metrics gave really unsatisfying predictions. The score did not reach a 0.5 score, even though in the dataset were still samples without a bot. End-to-end solutions also had difficulties in predicting true likelihoods for gfootball, but the achieved score decrease was not as drastic as in the hand-crafted model result.

The experiment shows that every developed method has a problem comparing agents which differ in some way in the style of playing from those agents the model was trained on. Exampled GRF bot’s statically implemented tactics differed in such a way that it likely exchange passes but rarely wins. In fact, a gfootbal_bot had the highest average number of passes exchanged per game of any agent in the set, which could be confusing for the model. A likely solution to this problem is to extend the training dataset by adding atypical agents.

9. SUMMARY (SZYMON ŻEBROWSKI)

This chapter summarizes the whole thesis and outlines the possible future directions of development for the proposed solution.

9.1. Research results

The target was to propose a scoring system for two reinforcement learning agents. Presented were implementations of two separate methods - hand-crafted metrics analysis (4), and more sophisticated end-to-end regression (5). In the approach taken, players were compared on the basis of the games against selected proxy players. The task of predicting the results of the match based only on the matches with proxy player turned out to be a complex problem. Both reinforcement learning agents and environments show a large variance causing even the same player's matches to vary significantly. Despite that proposed solutions were able to achieve promising results, but had problems classifying agents with an unusual style of playing. Every proposed method requires a linearly growing number of matches in terms of the size of the set of agents being compared, which makes it usable in RL self-play problems. Additionally, analogical end-to-end solutions potentially could be used in another competitive reinforcement learning environment, because they do not use the specific features of a football game. This and the fact that they achieved better results over hand-crafted models will make them have greater potential in the future. When it comes to finer, end-to-end solutions, two separated architectures were created, one based on a Transformer encoder network (5.1.2), and the second on a CNN (5.1.1). Experiments carried on validation and test datasets did not show unequivocally which is better in terms of prediction, but CNN provides comparison results in a shorter time. For dataset preparation currently existing leaderboard system was adopted and improved (6.2) also an infrastructure for collecting statistics has been created as part of the GRF environment (6.3.1). Finally, for the model evaluation and results presentation, a web application was created. Thanks to its architecture it is easily scalable and transferable. The application allows a user to predict winning likelihood for pair uploaded agents, to calculate the actual score by playing a given number of matches, and presents the results with a confidence interval in a non-confusing way.

9.2. Directions of development

Paths of further development should focus on the improvement of proposed methods, first of all by expanding the data size by adding more agents with a unique style of playing. It would also be worth trying to train models only on matches that are problematic, i.e. those whose course was different than usual. What is more, a large area of growth is the hyperparameter optimization problem, in which in-depth investigation has not been carried out in this work. The next step could be to check if the method used allows to improve the training results through the appropriate selection of opponents in reinforcement learning self-play.

Acknowledgement

This work has been partially supported by Statutory Funds of Electronics, Telecommunications and Informatics Faculty, Gdansk University of Technology.

REFERENCES

1. Jay Alammar. *The Illustrated Transformer*. URL: http://jalammar.github.io/illustrated-transformer/.
2. The Kubernetes Authors. *Kubernetes Documentation*. 2020. URL: https://kubernetes.io/docs/home/.
3. Y. Bengio i Yann Lecun. “Convolutional Networks for Images, Speech, and Time-Series”. W: (list. 1997).
4. Greg Brockman i in. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.
5. *Celery*. URL: https://docs.celeryproject.org/en/stable/.
6. Djork-Arné Clevert, Thomas Unterthiner i Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2016. arXiv: 1511.07289 [cs.LG].
7. *Docker*. URL: https://www.docker.com/.
8. Harris Drucker i in. “Support Vector Regression Machines”. W: (1996). URL: https://proceedings.neurips.cc/paper/1996/file/d38901788c533e8286cb6400b40b386d-Paper.pdf.
9. Rohith Gandhi. *Support Vector Machine — Introduction to Machine Learning Algorithms*. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.
10. Glosser.ca. *Artificial neural network with layer coloring*. URL: https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg.
11. *Helm*. URL: https://helm.sh/.
12. Markus Hofmann i Elaine Kirwan. “Predicting Premiership Football Match Results (Machine vs. Men – Can predictive models outperform human experts?)” W: kw. 2013.
13. Gareth James i in. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. Rozd. 3. ISBN: 1461471370.
14. Diederik P. Kingma i Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].
15. Karol Kurach i in. *Google Research Football: A Novel Reinforcement Learning Environment*. 2020. arXiv: 1907.11180 [cs.LG].
16. Yann LeCun, Yoshua Bengio i Geoffrey Hinton. *Deep learning*. 2015. URL: https://www.nature.com/articles/nature14539.
17. Minh-Thang Luong, Hieu Pham i Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. W: *CoRR* abs/1508.04025 (2015). arXiv: 1508.04025. URL: http://arxiv.org/abs/1508.04025.
18. Andrew L. Maas. “Rectifier Nonlinearities Improve Neural Network Acoustic Models”. W: 2013.
19. Philip Mayfield. *Binomial confidence intervals*. https://sigmazone.com/binomial-confidence-intervals/. 2018.
20. *MinIO*. URL: https://min.io/.
21. *Nvidia DGX-1 Station*. URL: https://www.nvidia.com/en-us/data-center/dgx-1.
22. OpenAI. *OpenAI Five*. https://blog.openai.com/openai-five/. 2018.
23. *PostgreSQL*. URL: https://www.postgresql.org/.
24. J. Ross Quinlan. “Induction of decision trees”. W: (1986). URL: https://link.springer.com/content/pdf/10.1007/BF00116251.pdf.
25. *Raft algorithm*. URL: https://github.com/etcd-io/etcd/tree/master/raft.
26. *Redis*. URL: https://redis.io/.
27. Herbert Robbins i Sutton Monro. “A Stochastic Approximation Method”. W: *Ann. Math. Statist.* 22.3 (wrz. 1951), s. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586.

28. Ihab S. Mohamed. “Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques”. Prac. dokt. Wrz. 2017. DOI: `10.13140/RG.2.2.30795.69926`.

29. Robert E. Schapire. “The strength of weak learnability”. W: *Machine Learning* 5.2 (1990), s. 197–227. ISSN: 1573-0565. DOI: `10.1007/BF00116037`. URL: `https://doi.org/10.1007/BF00116037`.

30. Florian Schroff, Dmitry Kalenichenko i James Philbin. “FaceNet: A unified embedding for face recognition and clustering”. W: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015). DOI: `10.1109/cvpr.2015.7298682`. URL: `http://dx.doi.org/10.1109/CVPR.2015.7298682`.

31. Bastiaan Schuiling. *Gameplay Football*. URL: `https://github.com/BazkieBumpercar/GameplayFootball`.

32. Nitish Srivastava i in. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. W: *Journal of Machine Learning Research* 15.56 (2014), s. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

33. Richard S. Sutton i Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

34. ZPP GFootball Team. *The Football Project*. 2020. URL: `https://sites.google.com/view/rl-football/zpp`.

35. x team.com. *Kubernetes components*. URL: `https://res.cloudinary.com/dukp6c7f7/image/upload/f_auto,fl_lossy,q_auto/s3-ghost/2016/06/o7leok.png`.

36. Tin Kam Ho. “Random decision forests”. W: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. T. 1. 1995, 278–282 vol.1. DOI: `10.1109/ICDAR.1995.598994`.

37. Corinna Cortes Vladimir Vapnik. “Support-vector networks”. W: (1995). URL: `https://link.springer.com/article/10.1007/BF00994018`.

38. Ashish Vaswani i in. *Attention Is All You Need*. 2017. arXiv: `1706.03762 [cs.CL]`.

39. Oriol Vinyals i in. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. W: *Nature* (2019), s. 1–5.

40. Wikipedia. *Binomial proportion confidence interval — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Binomial%20proportion%20confidence%20interval&oldid=985324095`. [Online; accessed 24-November-2020]. 2020.

41. Wikipedia. *Coefficient of determination — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Coefficient%20of%20determination&oldid=988642476`. [Online; accessed 23-November-2020]. 2020.

42. Wikipedia. *First-move advantage in chess — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=First-move%20advantage%20in%20chess&oldid=988096712`. [Online; accessed 02-December-2020]. 2020.

43. Wikipedia. *Least squares — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Least%20squares&oldid=987688589`. [Online; accessed 25-November-2020]. 2020.

44. Wikipedia. *Mean absolute error — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Mean%20absolute%20error&oldid=988251721`. [Online; accessed 23-November-2020]. 2020.

45. Wikipedia. *Mean squared error — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Mean%20squared%20error&oldid=989293917`. [Online; accessed 23-November-2020]. 2020.

46. Wikipedia. *Regression analysis — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Regression%20analysis&oldid=990194460`. [Online; accessed 25-November-2020]. 2020.

LIST OF FIGURES

LIST OF TABLES

Appendix A: GLOSSARY TERMS

Triplet Loss

Such a loss function takes three parameters: so-called anchor which is a baseline sample, a positive and negative input. The cost function decreases as the positive sample approaches anchor and a negative is moving away. Triplet loss function usually is expressed by a Euclidean distance function:

$$L(a, p, n) = maxd(a_i\ , p_i\ ) - d(a_i\ , n_i\ ) + \alpha, 0$$

Where:

$$d(x_i\ , y_i\ )$$

is the distance between samples in n-euclidean space and α is an admissible margin distance between positive and negative pairs.

R^2 score

Coefficient of determination [41] calculated as:

$$R^2 = 1 - \frac{\sum_i e_i^2}{\sum_i (y_i - \bar{y_i})^2}$$

Where

e_i is residual - difference between observer and predicted value

y_i is target value and $\bar{y_i}$ is target's values mean

MSE

Mean squared error [45] is cost function defined as:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y_i})^2$$

Where Y are target values and $\hat{Y}$ are model prediction.

MAE

Mean absoulute error [44].

$$MAE = \frac{\sum_{i=1}^{n} \left| Y_i - \hat{Y_i} \right|}{n}$$

Where Y are target values and $\hat{Y}$ are model prediction.

LeakyReLU

Activation function [18] defined as:

$$LeakyReLU(x) = \begin{cases} ax & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Where a is constant, calculated based on cross-validation process (usually 0.01)

ELU

Activation function [6] defined as:

$$ELU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha.(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

Where a is picked constant. A common value is between 0.1 and 0.3.

Sinusoidal positional encoding

Function generating vectors containing information about the position. [38] Usually defined as:

$$\vec{p_t}^{(i)} = \begin{cases} sin(\omega_k.t) & \text{if } i = 2k \\ cos(\omega_k.t) & \text{if } i = 2k+1 \end{cases}$$

Where:

$$\omega = \frac{1}{10000^{2k/d}}$$

t - desired position in input sentence

d - encoding dimension.

Dropout Regularization technique used in deep learning [32]

SGD

Fundamental optimization algorithm used for optimizing neural networks [27]

Adam

Advanced optimization algortithm [14]

Learning rate

Tuning parameter determining step size at each iteration. For Stochastic Gradient Descent:

$$W = W \ - \alpha \nabla L$$

Where

W - neural network paremeter e.g. weights

α - learning rate

∇L - Gradient Descent for cost function