

TDK-dolgozat

Patrik Péter Szmidá
2024.



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Artificial Intelligence

Evaluating Player Actions in Professional Counter Strike using Temporal Heterogeneous Graph Neural Networks

Students' Scientific Conference (TDK)

Author:

Patrik Péter Szmida
IV. semester

Advisor:

Dr. László Toka,
associate professor

2024.

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
1.1 Esports data	3
1.2 Related work	4
1.3 Initial state	5
1.4 Research paper objective	6
2 Data preparation	7
2.1 Initial state	7
2.2 Graph dataset creation workflow	8
2.2.1 Tabular dataset creation	9
2.2.2 Missing data	10
2.2.3 Normalization	11
2.2.4 Heterogeneous graph dataset creation	11
2.3 Exploatory Data Analysis	13
2.3.1 Correlations	13
2.3.2 Output variable proportionality	15
2.3.3 Time-series trends	16
3 Model training	18
3.1 Introduction to Graph Neural Networks	18
3.2 Model architecture	20
3.2.1 Heterogeneous convolutions	21
3.3 Hyperparameter optimization and HGNN results	22
3.3.1 Neural scaling laws in hyperparameter tuning	22
3.3.2 Hyperparameter tuning	23
3.3.3 Training Results	25

3.4	Temporal learning	25
3.4.1	Temporal dataset creation	25
3.4.2	Extending the model architecture	27
3.4.3	Mini-batch learning on dynamic graphs	28
3.4.4	Results	30
3.5	Comparing model predictions	31
4	Evaluating player actions	33
4.1	Explainable AI	33
4.2	The reason behind probability fluctuations	34
4.3	Interpreting probability changes	35
4.3.1	Events dataset creation	35
4.3.2	Explanation model and Shapley values	35
4.3.3	Explaining temporal patterns	36
4.3.4	Example explanations	36
4.4	Evaluating player actions	38
5	Summary	40
	Bibliography	41
	Appendix	42
A.1	Feature descriptions	42
A.2	SHAP Explanations	45

Kivonat

Az utóbbi években az elektronikus sportok (esportok) gyorsan népszerűvé váltak, kibővítve a sportipar eddigi színterét. A Counter Strike 2, egy first-person shooter csapatjáték, az egyik legnépszerűbb esport cím 2024-ben, nagyközönség előtt rendezett, magas díjazású bajnokságokkal. Népszerűsége ellenére azonban eddig kevés AI-alapú megoldást vizsgáltak egyéni és csapatteljesítmény javítására, és gyakorlati alkalmazásban még nem vált elterjedtté.

Ez a dolgozat egy olyan keretrendszert mutat be, amely képes a játékosok akcióinak értékelésére gráf neurális hálózatok (GNN-ek) alkalmazásával, a csapat győzelmi valószínűségeinek vizsgálatán keresztül. Ehhez egy megbízható és konzisztens adatátalakítási folyamat szükséges, amely képes heterogén gráf adathalmazokat létrehozni a nyilvánosan elérhető profi mérkőzések visszajátszási fájljaiból, a kívánt időbeli részletességgel (maximum 64 pillanatkép másodpercenként). Ezek a heterogén gráfok magukban foglalják a pályák egyedi elrendezését valamint a játékosokat, különböző csomópont típusokat és kapcsolatokat használva. Ezen gráfok időbeli összefűzésével létrejönnek a diszkrét idejű dinamikus gráfok, melyek időbeli információkat is tartalmaznak. Az így létrejövő adathalmazokon egy kezdetleges, még nem idő alapú (azaz temporális) heterogén GNN tanul azzal a feladattal, hogy megbecsülje, hogy a támadó vagy a védekező csapat nyeri-e az adott kört a mérkőzésen a kör egy adott játékkállapotát nézve. A modell több mint 700.000 gráfon tanulva 75%-ot meghaladó pontosságot ér el. Azért, hogy a játék dinamikus természete is beépüljön a tanulási folyamatba, temporális heterogén GNN-ek is használatba kerültek. A betanított modellek így képesek előre jelezni mindkét csapat számára a győzelem valószínűségét az adott körben. Ezt a valószínűséget, és az ebben történő ingadozásokat vizsgálva kiemelhetők a kör fontos pillanatai, illetve a lezajlott események alapján visszakövethető, hogy melyik játékos(ok) okozták azt. Ezzel lehetségessé válik a játékosok akcióinak kiértékelése egyénileg illetve csapatszinten is.

Ez a keretrendszer forradalmasítja az egyéni játékos teljesítmény elemzését a Counter Strike esport iparágban, részletes, képkockáról képkockára történő értékelést nyújtva a játékosok teljesítményéről, megnyitva az utat az innovatív döntéshozatali edzéseknek, valamint átfogóbb stratégiai tervezéseknek, ezzel versenyelőnyt biztosítva a szervezetek számára.

Abstract

In recent years, electronic sports (esports) has gained rapid popularity, extending the existing landscape of the sports industry. Counter Strike 2, a first-person shooter team game, stands as one of the most prominent esports titles in 2024, with tournaments organized in front of large audiences where professional teams are competing for substantial prize pools. Despite its popularity, only a few AI-driven solutions have been explored in the past regarding individual and team performance enhancement, and it has not yet gained much popularity in practical use.

This thesis proposes a framework capable of evaluating player actions using graph neural networks (GNNs) through examining team winning probabilities. This involves a reliable and consistent data transforming process capable of creating heterogeneous graph datasets by parsing publicly available professional match replay files at a desired frame-rate (maximum 64 snapshots per second). These heterogeneous graphs incorporate the unique layout of the maps with the common contact areas highlighted, as well as the players, using different node types and connections. This process is then extended to be able to concatenate these graphs in a temporal manner, making the creation of discrete-time dynamic graphs possible. Leveraging the datasets, a heterogeneous GNN is trained as a baseline model, using over 700.000 non-temporal snapshots, reaching over 75% accuracy. The task of the model is to predict whether the attacker or the defender team would win the examined round of a game given a game state from that round. In order to incorporate the dynamic nature of the game in the learning process, temporal heterogeneous GNNs are then deployed. With models able to accurately predict round-winning chances for the teams from snapshots, fluctuations in these probabilities are then examined within single rounds, as these highlight the most important events in-game. This approach allows teams to quantify player-wise action impacts, enabling the evaluation of player actions individually.

This framework revolutionizes individual player performance analysis in the Counter Strike esports industry by providing a detailed, frame-by-frame assessment of player impact, opening the possibility for innovative decision-making trainings as well as more profound ways of strategic planning, providing organizations with a competitive advantage.

Chapter 1

Introduction

In light of the rapid advancements in 21st-century technology, the collection and storing of data across various domains has witnessed an exponential growth. Consequently, we find ourselves in a period characterized by the rise of artificial intelligence (AI) and machine learning, heralding a substantial transformation that continues to progress. Presently, artificial intelligence stands as one of the foremost technologies, if not the preeminent one, characterized by a wealth of publications, innovations, and the perpetual enhancement of state-of-the-art solutions.

The AI revolution is explainable by the substantial increase in the amount of stored data and the desire to comprehend and learn from it. Machine learning algorithms operate by analyzing vast datasets to discover underlying insights, aiming to utilize this knowledge for future predictions, optimizations, and other purposes. These mathematical models often surpass human cognitive limitations, which renders AI a vital tool for mimicking cognitive functions.

Several categories arise when attempting to categorize machine learning models, with some of the most common ones being unsupervised and supervised learning models. The primary purpose of unsupervised AI is to identify hidden patterns in the examined dataset without possessing prior knowledge about what to search for. This process can unveil crucial, previously unknown insights about the examined problem. The foundation upon which this paper is based involves unsupervised clustering methods, which will be explained in detail in the background section. The other, perhaps the most extensive and popular category of AI models, is supervised learning. These algorithms operate by understanding the fundamental connections and relationships between the descriptive (or predictive) features and the target variable in a dataset where both are available. They subsequently employ this acquired knowledge to predict the target label on previously unseen data. One of the most rapidly advancing areas of supervised learning is neural networks (NNs) in 2024. These models have the ability to recognize and comprehend incredibly intricate relationships at the expense of simple explainability. They utilize these insights to predict solutions across various domains. Neural networks, too, can be divided into categories based on their functionality. One of these is known as graph neural networks (GNNs). These models are capable of learning insights from graph-structured data. Graphs are data structures that are built from nodes, each acting as an entity, and connections between them, represented by edges. These structures provide a means to precisely represent relationships and structures between objects; a quality that makes graphs a preferable choice over tabular data representation in numerous applications, such as social network

modeling, among others. GNNs are increasingly gaining popularity, and in this paper, I leverage their power as the primary tool to address the research problem.

Sports always have leading viewer numbers when it comes to entertainment, with professionals captioning audiences with their incredible skills, performances, and teamwork. The top-level environment makes it essential for its competitors, both players and teams, to continuously improve their abilities and strategies. In 2024, the most efficient tool to help achieve this is sports analytics. It is a field that combines sports with data science and machine learning with the goal of maximizing athletic performance and assisting professionals in optimal decision-making while on the court. Sports analytics operates by gathering, analyzing, and understanding data collected from past games. This knowledge can help the teams prepare new strategies against their opponents or improve the real-time decision-making and performance of their athletes. This innovative discipline can often surpass traditional training methods, allowing coaches, athletes, and teams to enhance their abilities and gain a competitive advantage on the field.

While professional sports stand as one of the leading sectors in the entertainment industry, a new variant of contests on a top level has emerged. In the 21st century, competitive video gaming has become a global phenomenon as a result of modern technology, leading to the emergence of a new branch of professional sports known as electronic sports (esports) [5].

Esports has rapidly gained popularity in recent years. The term refers to competitive video gaming at a professional level in several kinds of games, such as first-person shooters (FPS), multiplayer online battle arenas (MOBA), real-time strategy (RTS), and sports simulations. The professional events from the different esports games entertain millions of fans worldwide. Akin to traditional sports, esports features professionals whose full-time job is to study these games and improve their own skills and teamwork abilities, qualities necessary to be able to perform at the highest level. Top events feature the best teams and individuals worldwide who compete for substantial prize pools and titles. Esports competitions not only fill huge stadiums with passionate fans but also draw millions of online viewers, creating a special atmosphere akin to traditional sports events. One of the most popular esports titles that certainly attracts fans worldwide is Counter-Strike.



Figure 1.1: A scene from the grand final of the Intel Extreme Masters Katowice 2023 Championship.

Counter-Strike 2, abbreviated as CS2 (or CS), holds a prominent position in the global esports landscape, ranking in the top three most popular esports game worldwide in 2024 [1]. One of the main reasons why CS is highly popular is because of its easy-to-understand

yet highly entertaining and exciting gameplay, coupled with top-tier tournaments organized both in stadiums and through online streams. Figure 1.1 captures a moment from a tournament’s grand final. Developed by Valve Corporation, it is a team-based first-person shooter that demands skill, strategic thinking, and teamwork from its competitors. Two opposing professional teams go up against each other in this esports. Both teams choose their roles at the start, aligning themselves as either Terrorists (Ts) or Counter-Terrorists (CTs). With seven distinct maps, each offering a unique structure and tactical setting, the game provides a dynamic platform for teams to master. Each of these maps features two small areas known as bombsites, which play a crucial role in the gameplay. Figure 1.2 shows the layout of some of the most popular maps.

A match takes place on one of the seven maps and consists of rounds lasting approximately two minutes. At the beginning of each round, players spawn in either the CT or T starting area, depending on their team. By default, they are equipped with a basic pistol, but if they survived the previous round, they retain any weapons and gadgets they had saved. Before each round begins, players utilize the initial 15 seconds to acquire better weaponry and tactical gear if they can afford it, while also discussing strategies for the upcoming encounter. During this preparatory phase, a designated member of the Terrorist team obtains a bomb, which can be freely exchanged among teammates. As the timer starts, both teams move towards the bombsites. The objective for the attackers is to secure control of a site, plant the bomb, and detonate it, while the defending CT players strive to prevent this or defuse the explosive in case of a successful activation. In the event of a bomb plant, a countdown begins, granting the defending team a critical 40-second window to neutralize the bomb threat. Victory in each round hinges on achieving the specified objectives or completely eliminating the opposing team before time runs out. After each round, both teams’ members are rewarded with in-game currency, which they can utilize to upgrade their weaponry and equipment. However, the losing team earns substantially less, potentially placing them at a disadvantage in future rounds due to equipment discrepancies. After the first 15 rounds, the first half of the game ends. Each game is split into two halves with a maximum span of 15 rounds. The two teams switch roles in halftime (CTs become Ts and vice versa), and the match continues. The first team to reach 16 rounds wins.

1.1 Esports data

Esport analytics is also fundamental to individual and team development in the professional scene, but it has a major advantage compared to traditional sports due to being played in a virtual environment. Many esports titles, including Counter-Strike, offer players the opportunity to rewatch any match if they have the replay file (also called a demo file). In CS, a dedicated company called HLTV specializes in collecting and publicly sharing professional tournament-related information, such as news, player statistics, ratings, and replay files of professional matches. Through these demo files, one can rewatch the matches in the game, but they can also be processed to extract data into CSV files at an astonishing rate of 128 snapshots per second, each containing every available piece of information related to the game at that moment, including positions, weapons, utilities, and more. This is made possible by a Python library called awpy, which was published in the article described below.

1.2 Related work

The primary goal of this project is to develop a machine learning-driven player action evaluation system, drawing inspiration from Peter Xenopoulos’ work titled *Valuing Player Actions in Counter Strike: Global Offensive* [11]. In the article, he proposed an innovative method to assess player decisions by analyzing winning probabilities. His approach involved an XGBoost (eXtreme Gradient Boosting, XGB) model that took tabular data that contained all critical aspects of a game snapshot as input—such as player positions, weapons, remaining health of players, time left, number of players alive, and more. With this XGB he effectively predicted the likelihood of either the CTs or the Ts winning the examined round by looking at a snapshot taken from it. Extending this idea, he leveraged this model to predict the winning team throughout whole rounds (a prediction for every second of a round) using the game state representations. Looking at these predictions, changes in the winning chances can be examined as the round progresses. The fluctuation in team winning probabilities over time stemmed from players’ decisions and actions, coupled with game events like kills occurring across different map locations. Xenopoulos highlighted the correlation between changes in winning probabilities and player actions, suggesting a method to evaluate player decisions by reverse-engineering these fluctuations. Essentially, a player’s decision was deemed valuable if it positively impacted their team’s winning probability, while a negative impact signaled a poor tactical choice.

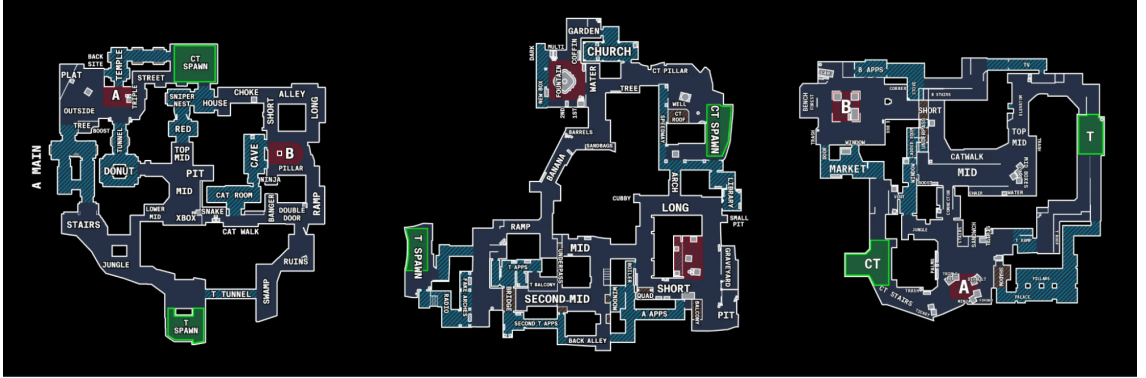


Figure 1.2: Layout of maps Ancient, Inferno, and Mirage, respectively.

Source: <https://csgoconsole.com/callouts> (November 2, 2024)

While Xenopoulos’ approach introduced a groundbreaking methodology for assessing decision-making in Counter-Strike, it does have limitations. One prominent drawback lies in storing game snapshots as tabular data. Since the different maps have unique structures, as shown in Figure 1.2, representing game states in a tabular format cannot capture the intricate relationships between various parts of the seven unique maps in Counter-Strike. Xenopoulos’ approach stored player coordinates in the game-state vectors, complemented by one-hot encoding for position names, but this method fails to capture the spatial relationships between these areas. Moreover, the model’s reliance solely on the current game state overlooks the importance of previous states. Considering the dynamic nature of the game, incorporating historical game states into predictions would provide a more comprehensive understanding of gameplay dynamics and potentially enhance the accuracy of the model’s predictions.

1.3 Initial state

The initial state of the project involved a contact spot analysis on the map Inferno, a prominent map in professional Counter-Strike, aiming to identify "weak" and "strong" engagement zones. There are positions where players are frequently eliminated or succeed in eliminating opponents. Unsupervised clustering algorithms, including DBSCAN, OPTICS, and HDBSCAN, highlighted critical engagement areas. Ensemble techniques further refined these contact spots by averaging clustering similarity matrices across models and weighting them based on a custom scoring system using the Hopkins statistic. This Bayesian model averaging approach provided an enhanced representation of these zones. Figure 1.3 illustrates the resulting clusters in a region on Inferno.

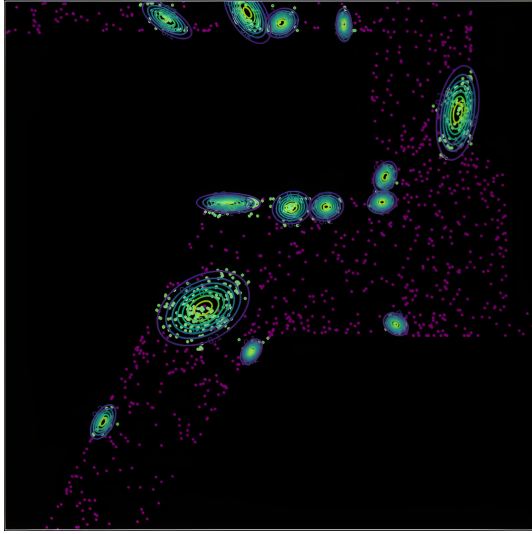


Figure 1.3: Gaussian distributions of clusters on an area of the map Inferno.

The project also initiated an evaluation of player actions and decision-making in professional Counter-Strike matches, focusing on leveraging GNNs to overcome spatial information limitations in game snapshots. To support the extensive data requirements, a custom scraper collected match replays and player statistics from HLTV, covering tournaments from early 2022 through Q2 2023. A base skeleton graph was constructed to represent the map Inferno, involving a thorough analysis of player movement patterns and strategic node placement. Cluster analysis results were integrated by adding cluster centers as nodes, highlighting critical engagement areas. The resulting graph structure, shown in Figure 1.4(a), provides a detailed model of Inferno.

The initial state also included a dataset of momentary game-state representations, constructed by embedding player-specific data into the map graph structure. This approach involved using distinct node types for players, transforming the analysis from ordinary to heterogeneous graphs. In each game snapshot, player nodes were positioned based on real-time locations and connected to the nearest map-type nodes. These nodes integrated both state-descriptive features, such as positions and directional velocities, along with statistical metrics from the current game and overall performance data. An example graph is illustrated in Figure 1.4(b).

Trained graph neural network models were also involved, aimed at predicting the round-winning team. This objective was approached as a binary graph-classification problem, where each game state served as input to predict whether the defending team would secure

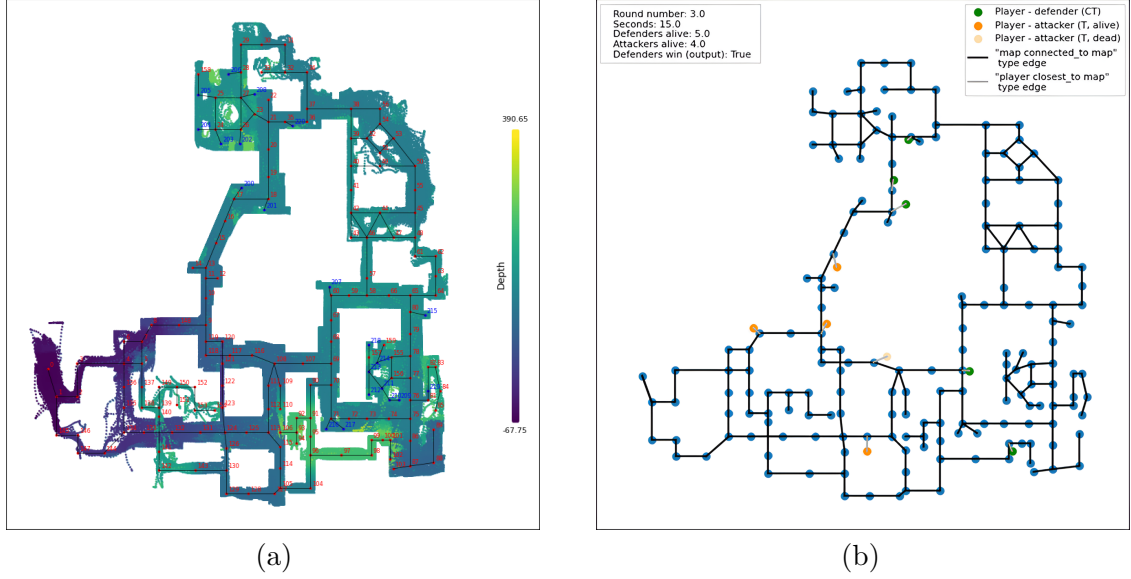


Figure 1.4: (a) Graph of the map Inferno, with the background color outlining the shape of the map and blue nodes on the graph highlighting important contact areas. (b) An example graph that represents a moment in a round from a professional match.

the round. Initial experiments with various network architectures and convolution types resulted in a peak accuracy of slightly over 65%, which, while promising given the problem’s complexity, highlighted several areas for improvement. These areas included the absence of hyperparameter optimization, as well as minor inaccuracies in data representation due to the dataset creation process. Additionally, data scaling techniques, such as normalization or standardization, were not applied. There was also potential to refine the map graph.

1.4 Research paper objective

This paper aims to address the imperfections found in previous works and the shortcomings of the referenced article. In the related work, two main drawbacks were identified: the lack of accurate spatial representation and the neglect of the time-series nature of the data. This paper proposes solutions for both shortcomings. The first drawback is addressed by employing heterogeneous graphs for the snapshots, accurately capturing spatial relationships. The paper proposes a consistent and reliable data transformation process capable of converting professional CS match replays into game-state-representing heterogeneous graphs. This process is extended with data scaling functionalities, such as normalization. To address the second issue, (Heterogeneous) Temporal Graph Networks (TGNs) are utilized in the learning process to account for the time-series nature of the data.

The prediction task remains the same: a binary graph classification to foretell whether the defending CT side would win the round. The neural network training involves hyperparameter optimization to enhance the predictive performance of previous GNNs. Another essential aim of the paper is to examine the probability predictions of the models and use it to evaluate the player’s actions in the game.

Chapter 2

Data preparation

This chapter addresses the data preparation topics of the paper, serving as the foundation for graph neural network training. This involved developing a consistent and reliable data transformation workflow that encapsulates information from game snapshots in graph format, thereby creating a dataset for the graph neural networks to learn from. This workflow also includes data scaling functionalities, enabling normalization. This workflow is implemented in Python and relies on the Pandas and Numpy packages. Additionally, this section contains an Exploratory Data Analysis to examine various relationships between variables in the dataset. Python’s Matplotlib library served as my main tool for plot generation and visualization in this part. The overall dataset quality is also assessed through different quality dimensions such as data accuracy, availability, completeness, consistency and others.

2.1 Initial state

As a result of the efforts in previous semesters, the data gathering did not start from scratch. Utilizing the scraper I developed, I was able to collect all top-tier tournament match replay files for matches held from Q3 2023 until recent days. This amounts to approximately 1,000 match replay files in total. However, since both my previous works and this paper focus on matches played on the map Inferno, the relevant match count is around 120. Using the awpy library, I parsed all Inferno match files at a rate of four snapshots per second. I concluded that four snapshots per second is sufficient because there are no relevant, game-changing events occurring within milliseconds; thus, this rate is adequate to capture all important game events and there is no need for 128 game states per second. The parsing process with this package involves taking a demo file as input and generating nine CSV files as output, each containing information about various aspects of the actual game. A brief introduction to these files is provided below.

- **bomb:** This file contains bomb-related information about the parsed match, such as the start and finish times of the bomb plant.
- **damages:** This CSV contains all information related to dealt damages in a match, including time stamps, used weapons, and more.
- **rounds:** These files contain general information about the rounds, such as the score, the winner, etc.

- **grenades:** The dataset holds all grenade-related information about the parsed match.
- **infernogs:** A detailed dataset for the incendiary grenades.
- **smokes:** A detailed dataset for the smoke grenades.
- **kills:** Contains detailed data about every elimination that occurred in the game, including the time stamp, the weapon used, and additional relevant information.
- **ticks:** This file contains all player-related information, such as status features (coordinates, speeds, and others), captured at the specified parse rate.
- **weapon_fires:** Information regarding every weapon fire in the game is included, encompassing details such as coordinates, the weapon utilized, and additional data.

In addition to the parsed match demo files, which encompass all match-related information, the map model was also crucial for the creation of the graph dataset. I utilized a refined version of the Inferno graph model in Figure 1.4(a) for this purpose. The map graph comprised 199 nodes, each with unique properties, and 231 undirected edges that interlinked them to form the map structure. The node features included the X, Y and Z coordinates of the node, along with boolean flags. These bits indicated whether it is a common contact spot (based on the cluster analysis conducted in previous semesters), or whether the node is burning or smoked at the time of the snapshot. This structure was stored in two CSV files: one containing the nodes and their properties, and the other containing the edges.

The available datasets provided sufficient information to initiate the graph dataset creation process. However, I aimed to incorporate universal player-describing metrics as well. The rationale behind this was to furnish future GNN models with comprehensive statistical insights regarding players, as significant skill differences could exist among them regarding their roles. For instance, in-game leaders, akin to team leaders in traditional sports like football, possess exceptional tactical thinking and team-coordinating abilities, but they often lack top-tier shooting skills. Conversely, there are star players in this esports blessed with remarkable shooting prowess. To enable GNNs to distinguish between different player types, I employed the player statistical scraper function within the developed scraper to gather performance features for all players who participated since Q3 2023. For each player, I adopted a strategy of assigning their statistics from the previous year as uniform descriptors. For instance, for a game played in 2023, players would be attributed their 2022 statistics, and so forth. Leveraging the scraper, I amassed universal player statistics for 2022 and 2023 into distinct datasets, where each row encapsulated one player’s yearly average metrics.

With all the necessary datasets in hand, I proceeded to develop a data transformation workflow capable of reliably generating graph game state representations from the information dispersed across various data files.

2.2 Graph dataset creation workflow

In this section of the paper, I propose a data transformation workflow that can consistently and reliably create accurate graph game state representations of the examined matches. Leveraging insights gained from previous semesters, I could accurately outline and plan

the desired data structure to be produced as the output of the transformation process. The workflow takes a single match replay files, the Inferno graph model, and the player statistical information datasets as input and returns a graph dataset containing the graph representation of all snapshots of the match.

2.2.1 Tabular dataset creation

The development of the transformation process began by dividing it into two distinct workflow sections. The first section involved only the parsed game replay files and the overall player statistics dataset. The objective of this phase was to prepare a tabular dataset where each row represented a game snapshot, containing all essential information such as player states, statistics, and overall game state features. Essentially, each row of this table served as a tabular version of the yet-to-be-created graph, without incorporating any details about the Inferno graph model. The steps involved in this phase of the data transformation are outlined below:

1. **Preparing the datasets:** The first step involves leveraging the *awpy* library to parse the replay of the match and get the dataframes listed previously, along with the yearly player statistics dataset. Specifying the replay file paths for the transformation process is the only necessary step for this phase.
2. **Calculating in-game statistics for players:** Utilizing the five replay datasets, I measured in-game performance properties about the players, such as the number of kills, deaths and assists in the match. In addition to yearly performance metrics, statistics from the actual matchup are also vital for the prediction task. For instance, if the model is examining snapshots from the 10th round of a match between teams *A* and *B*, and team *A*'s star player is underperforming, this will be reflected in his in-game measures. The model can then incorporate this information as a potentially critical factor in its prediction.
3. **Handling weapon information:** Within the *ticks* dataset, columns record the weapon currently held by each player in every snapshot, as well as their inventory. This information is encoded among the player features as boolean flags, one representing whether the player has a specific weapon in his inventory and another whether it is the current active weapon. Additionally, ammunition-related informations are also stored to encapsulate further details (for example if the player is low on ammunition).
4. **Player data:** The next step involved segmenting the *ticks* CSV contents based on individual players. Each row in this dataset encapsulated information about one of the ten players at a specific moment in the match. By dividing the dataset into ten smaller datasets, each corresponding to a single player, handling and analyzing player data became manageable. Following this separation, I extended each player's dataset with the yearly performance metrics obtained from the scraped data, enriching the available information.
5. **Bomb-relevant information:** This part of the workflow focuses on collecting all bomb-related events from the game and preparing them for the integration into the output dataset. By systematically gathering and organizing these events, this step ensures that crucial bomb-related data is readily available for inclusion in the final dataset.

6. **Creating the tabular snapshot dataset:** This phase involved combining the previously created player dataframes and extending them with general match-related information such as score, time remaining, bomb-related data, and other relevant details. This final step of the subprocess resulted in the creation of the output tabular snapshot dataset. Each row of this table represented a moment of the processed game, encompassing all essential information about both the players and the overall game state. This comprehensive dataset served as a detailed and structured foundation for the subsequent graph creation process.

This subprocess accurately integrates data dispersed across several files into a coherent match snapshot tabular dataset, where each row corresponds to a moment in the given match. Given that matches are parsed at a rate of four snapshots per second, and matches in this esports typically last around 30 to 40 minutes, each match’s tabular dataset comprises roughly 5,000 to 7,000 rows, with an overall feature count exceeding 1,600. Although the paper concentrates on matches played on Inferno, the process is generalized and capable of transforming data related to all maps.

2.2.2 Missing data

In the initial phase of the process, a challenge emerged when attempting to merge the previous year’s player performance dataset with the parsed match snapshot files. Some lesser-known players, seldom seen in top-tier games, lacked available data on HLTV. Although this only affected around five percent of the players, it raised an important issue: how to handle this missing data.

To tackle this challenge, I devised a simple yet refined imputation strategy, going beyond basic mean or mode imputation techniques. This involved a thorough examination of the yearly player feature dataset, particularly focusing on the distributions of the statistical variables. Drawing from the insights gained, I crafted a fictitious player profile with statistics reflecting an average player in all respects. However, directly imputing these measures for all missing players risked homogenizing the data and potentially obscuring important distinctions between players or introducing unwanted patterns. To overcome this issue, I leveraged the fictive player to create a dataset consisting of approximately 1000 similarly fictitious profiles, each with slightly varied statistics. Each feature X of this new dataset followed a normal distribution

$$X \sim \mathcal{N}(\mu, \sigma^2),$$

where μ represents the original fictitious individual’s value for the column and σ^2 is the feature interval’s 2.5%. For instance, the average *rating 2.0* statistic for the first fictive player was 1.0, and the values in this dataset ranged from approximately 0.94 to 1.06. This created slight difference among the players while representing each with ordinary statistics.

Implementing this approach in the workflow, when encountering a player with missing data, one profile from this dataset would automatically be assigned to him. This ensured that each player had unique statistics while maintaining consistency across values. Given that these features encompassed uniform player-related statistics from the previous year, it was crucial for the same player to have consistent statistics across matches. This approach effectively addressed missing data imputation in a manner that upheld consistency throughout the dataset.

2.2.3 Normalization

Normalization serves as a vital preprocessing step, harmonizing features with diverse value intervals into a unified range between zero and one. This ensures consistency and comparability within the dataset, potentially enhancing the performance and convergence of machine learning models. The proposed data transformation workflow incorporates an optional normalization subprocess. It expects the created tabular snapshot dataset and the map graph model as inputs. Reliable normalization of the matches requires satisfying the following three criteria:

1. **Player invariant:** Columns representing specific player information must be scaled uniformly, irrespective of the player they correspond to.
2. **Dataset invariant:** Columns containing identical information should be scaled uniformly, irrespective of the dataset in which they are stored.
3. **Match invariant:** Columns containing identical information should be scaled uniformly, regardless of the match in which the data was recorded.

Initially, a solution was considered regarding the first condition. It involved reverting the dataframe structure to mirror that of the original *ticks* dataset, allocating ten rows for each game snapshot to encapsulate information about the ten different players. However, this approach proved impractical, as it would substantially increase conversion time. Furthermore, scaling this method for larger datasets, such as normalizing 50 matches collectively, would lead to vast dataset sizes, resulting in slow performance and memory constraints.

To overcome these concerns, I devised a more efficient strategy by building a dictionary to store the highest and lowest values for each column for a match. The extraction of these values is a fast operation using the Pandas library. This method facilitated the synchronization of normalizing across different players by considering all ten players' values when selecting the scaling interval. Moreover, this approach also provides scalability, as the dictionary could be efficiently constructed across multiple matches, ensuring the third condition is also satisfied. The dataset invariance problem stated in the second condition arose when normalizing coordinates. Positional features X, Y, and Z were not only separated among the 10 players but were also present in the Inferno map's node dataset. A solution to this was normalizing the nodes of the Inferno graph first and then applying the same scaler to transform all player coordinates. This approach may result in player coordinate values occasionally falling below zero or exceeding one, as the nodes on the edges of the map are not aligned precisely with the playable boundaries. However, this does not compromise the consistency or informational integrity of the scaled values, thus it satisfies the second condition.

These solutions for single match normalization enable scalability to multiple matches by utilizing dictionaries to unify scaling. Additionally, the map structure-based coordinate normalization ensures consistency in normalizing positional features across all datasets.

2.2.4 Heterogeneous graph dataset creation

After the normalization, the data was ready to be converted into a graphs. In this final phase of the transformation process, the Inferno graph model served as the foundational structure. Essentially, a graph is created for every row (snapshot) in the tabular dataset, incorporating players into the graph as well as overall game features. To achieve this with

precision, I opted for the use of heterogeneous graphs. Heterogeneous graphs, also known as knowledge graphs, accommodate multiple types of nodes and links, each corresponding to distinct information and connections between them. This data structure holds the potential to store and represent complex information relationships; a flexibility essential for this use case. Players, as entities, possess a significantly larger number of features (around 160) describing their state, compared to the map entities, which have only nine. Heterogeneous graphs adeptly handle this disparity by recognizing the two node types and maintaining the uniqueness of each. This allows for a detailed representation of the game states, capturing both the complexity of player attributes and the spatial dynamics of the map. Thus, heterogeneous graph representations were the ideal choice to use.

Another crucial factor in the graph creation process was the choice of library, ensuring compatibility with future neural network models. In previous semesters, I leveraged the PyTorch and PyTorch Geometric (PyG) packages. PyTorch is a popular neural network library designed for deep learning tasks, while PyG is an extension that facilitates learning on graphs. I decided to continue using these packages for the paper. I utilized the HeteroData object from the PyG library to construct and store the game state representation graphs.

Initially, this second part of the transformation process creates an empty HeteroData object and adds the nodes of the Inferno map to it, labeling them as *map* type nodes. The nine properties of each node are stored in the node’s feature set tensor. To represent the connections between the vertices, the first edge type, *map closest_to map*, is initialized, linking *map* type nodes with one another. At this stage, the graph of the Inferno map is successfully stored in HeteroData format. The next step in the process is to incorporate the ten players’ data from the previously created tabular dataset. The function responsible for this adds ten *player* type nodes to the graph at the appropriate positions based on the player coordinates and stores the player attributes in the node tensors. To seamlessly integrate these entities into the graph, they are connected to the map structure by calculating the closest *map* entity for each player node and creating a *player closest_to map* directed edge, which serves as the second link type in the knowledge graph. Lastly, self-loop links for the players were added, labeled as *player is player*. The role of this will be examined in detail in a later section. The process also enhances the graphs with global features that are not tied to specific nodes or edges; hence, they are stored at a global graph level. Some of these attributes include the remaining time, the count of alive players per team, the game score, and other overall game metrics, as well as the output variable indicating whether the defenders win the round. These graph-level attributes enrich the holistic representation of each game snapshot.

To ensure numerical accuracy, the data types for storing all relevant information required careful planning. Player and map feature tensors were stored as float values, with particular attention given to bit precision, as the required level varied among features. After testing various bit lengths, the *float32* format proved to be the most reliable, accurately capturing the necessary details in the feature tensors. Although the *float64* format, which offers greater decimal precision, was also tested, it was deemed unnecessary for this task and suboptimal in terms of memory usage. Conversely, the *float16* format, which is accurate only to three decimal places, lacked sufficient precision for certain features, such as coordinates. For other graph features, the data type varied based on the information to be represented. Boolean flags, for example, required no more than a single bit, making PyTorch’s *bool* type the most efficient choice. Additional features, such as the number of players alive or consecutive rounds lost, were stored as *int8* values, as these

features encompassed small natural numbers. In each selection, the primary objective was to represent the data as accurately as possible while maximizing memory efficiency.

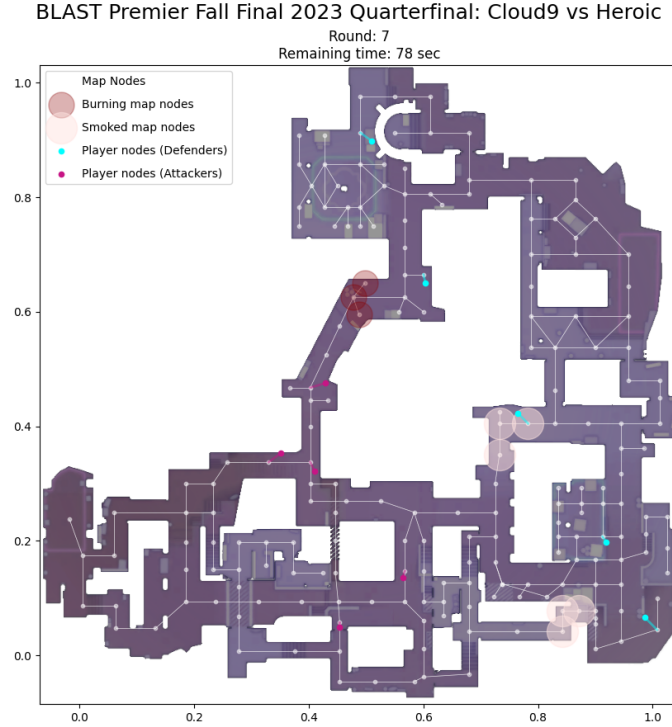


Figure 2.1: An example graph from the created dataset. The game state is from a quarterfinal match between teams Cloud9 and Heroic.

The finalization of the heterogeneous graph creation subprocess marked the completion of the data conversion procedure. Figure 2.2 summarizes the workflow in a flow chart. Employing this converter, every scraped game since Q3 2023 was converted into heterogeneous graph datasets one by one, ensuring the normalization process is synchronized across all games. With the four snapshots per second conversion rate, this resulted in approximately 700,000 heterogeneous graphs. Figure 2.1 illustrates an example graph from the dataset.

2.3 Exploratory Data Analysis

Exploratory Data Analysis (EDA) is the process of summarizing and visualizing the key characteristics of a dataset to uncover patterns, anomalies, and test hypotheses. In this section, the tabular game state dataset is used for a detailed analysis to understand its structure, distributions, and potential insights it may offer. With over 1600 columns in the dataset, this EDA section will focus on highlighting the most important phenomena among them. The complete feature list, along with descriptions, is provided in A.1.

2.3.1 Correlations

Examining correlations is crucial to understand the linear relationships between different variables. In this section, the focus is solely on the correlations between overall game-describing features. Given the complexity of the entire dataset, which includes over 1600

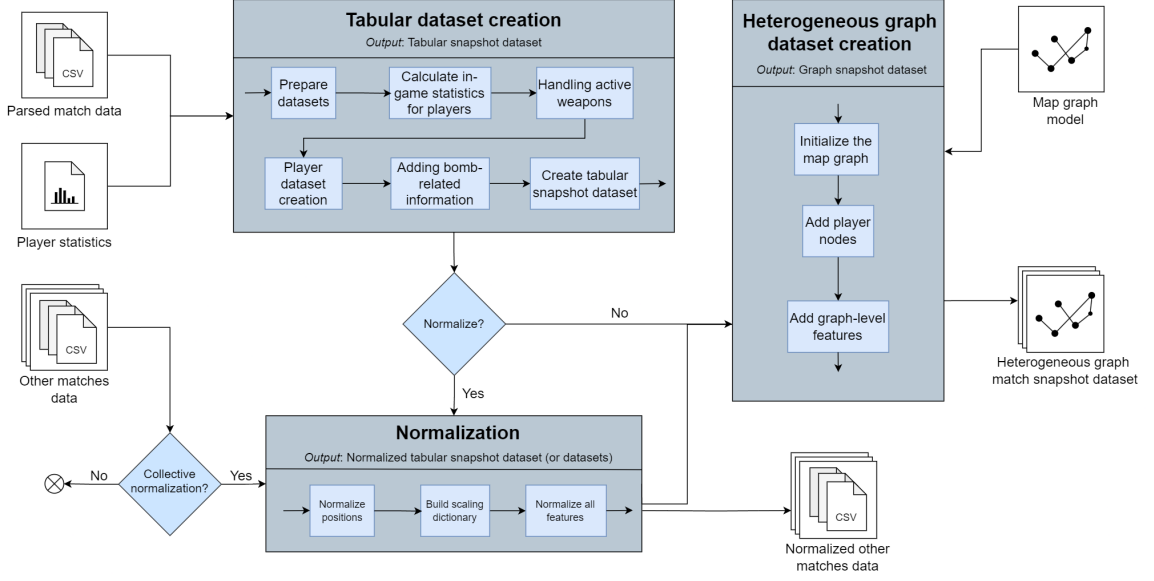


Figure 2.2: Flow chart of the graph data transformation process.

columns with numerous player-specific features, narrowing the scope to general attributes allows for a more manageable and insightful analysis.

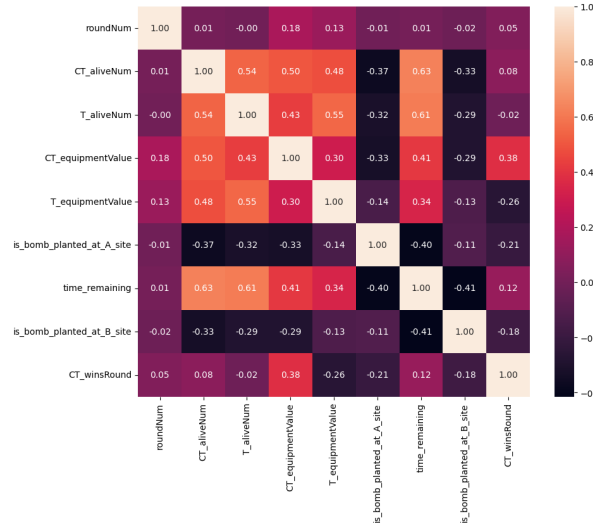


Figure 2.3: Correlation matrix of the game-describing features.

The correlation matrix on Figure 2.3 examines the linear relationships between overall game state-describing features. Interestingly, the heatmap reveals that the output variable (*CT_winsRound*) does not exhibit significant correlation with any other feature, underscoring the complexity of the prediction task. The highest correlations are with the number of players alive on each team, equipment values, and remaining time, which aligns with intuitive expectations as these are critical factors in determining the round-winner. However, the generally low correlations indicate that predicting the winner of the round involves many additional factors beyond these key variables.

2.3.2 Output variable proportionality

Examining the output variable is essential to understand its characteristics and behavior within the dataset. In the presence of bias, it may be necessary to implement equalizer strategies and transformations to ensure a balanced and fair analysis. The correlation plot highlighted the most important relationships of the *CT_winsRound* output variable, providing further motive for a detailed examination of these relationships, as they can reveal deeper insights into the factors influencing round outcomes.

To discover potential bias within the dataset, the examination of the distribution of the output variable is mandatory. Given that it represents either the attackers or the defenders winning the round, the target feature conforms to a Bernoulli distribution. Denoting this as X , the proportion of Counter-Terrorists winning the round is represented by $Bernoulli(X) = p$, the likelihood of Terrorists emerging victorious is represented by $(1 - p)$. The analysis of the round win proportions highlighted that 46.1% of the graphs resemble a CT win, while in the other 53.9% the Ts were the winners. It is important to point out that these proportions serve as estimations of the actual values, as the lengths of the rounds introduce a degree of bias. For example, consider a dataset comprising only two rounds: the first round lasts 2 minutes and is won by the attackers, while the second round lasts one minute and is won by the defenders. In this scenario, the output variable ratio derived from the dataset would reflect 33.3% for the CTs and 66.6% for the Ts. However, the true ratio of rounds won is 1 to 1. According to both HLTV statistics and an analysis with corrected round lengths, the actual proportions of round victories revolve around 51% to 49% for the CTs and Ts, respectively. This balanced distribution suggests that techniques such as Synthetic Minority Over-sampling Technique (SMOTE) for bias handling are not mandatory for the dataset. Moreover, this equitable proportion further underscores the challenge inherent in predicting the outcome of each round. Based on this insight, an output variable proportionality analysis is conducted based on the number of players alive as well as the remaining time feature. The results are visualized in Figure 2.4.

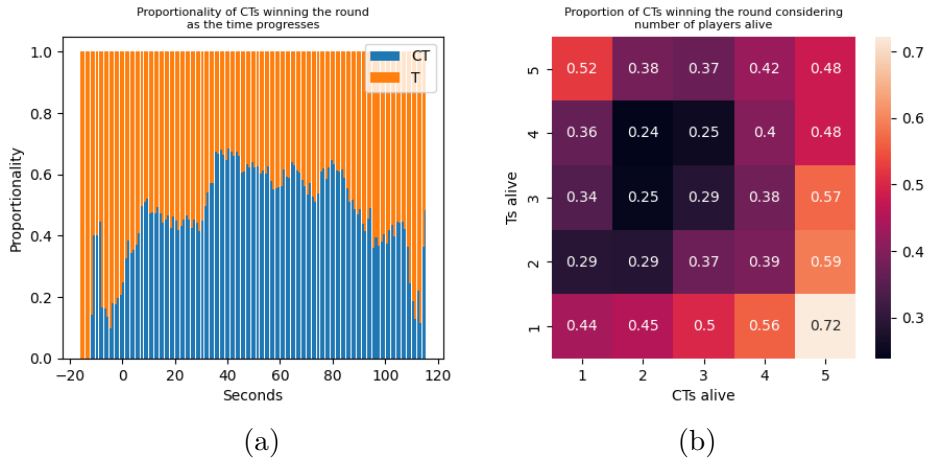


Figure 2.4: (a) Proportion of CTs winning the round examined across all seconds. (b) Proportion of CTs winning the round depending on the number of players alive on each team.

Figure 2.4(a) illustrates the proportions of round victories for the Ts and CTs based on the round timestamp. Interestingly, it reveals a bimodal distribution in which the Ts tend

to secure victories in rounds of extreme brevity or lengthy duration, while the CTs excel in rounds of intermediate length. This trend can be attributed to the dynamics of the game: the CTs thrive in rounds of moderate duration because the time factor becomes essential for the Ts as they have to plant the bomb. Thus, if the CTs can hold their ground while the Ts are trying to enter a bomb site, they have high chances of winning. Conversely, if the attackers successfully get to the bomb sites, they can plant the bomb. This initiates a 40-second countdown, as indicated by the negative time values on the histogram, putting the defenders under pressure. As a result, in the after-plant scenarios, the favored side is typically the attackers.

The matrix on Figure 2.4(b) illustrates the round winner proportion based on how many players are alive in each team. It is evident that the number of players alive on each team significantly influences the prediction of round winners. The data reveals that even when one team has only one more player alive than the other, there is a noticeable shift in the proportion of CTs winning the round. However, this metric alone is not always sufficient to predict the round-winning team, as the matrix demonstrates instances where winning the round is still achievable despite a player number disadvantage, contingent upon various factors such as team composition and strategic positioning, further underlining the complexity of the prediction task.

2.3.3 Time-series trends

Given that each game round spans approximately two minutes, and the objective of this paper is to predict the round winner using game state snapshots captured every second, it allows for the visualization of trends within individual rounds.

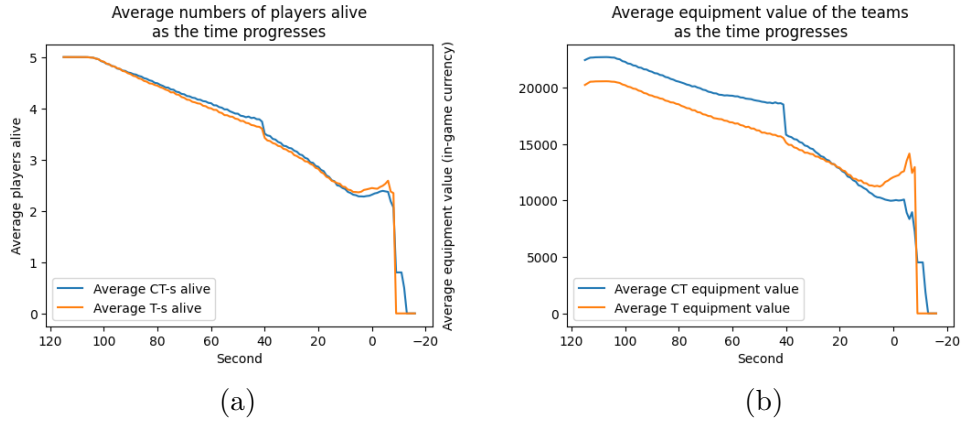


Figure 2.5: (a) Average number of players alive on each side as the time progresses. (b) Average total equipment value of the players on each team as the time progresses.

From Figure 2.5(a), it is evident that, on average, the number of Counter-Terrorists alive exceeds that of Terrorists, with a minor peak for the attacking side occurring around zero seconds. This observation can be attributed to the fundamental dynamics of the game. Counter-Strike matches are divided into rounds, each of which is won by either the CTs or the Ts. Following each round, both teams receive in-game currency, which they can utilize in subsequent rounds to acquire better weapons and equipment. However, the winning team earns significantly more money than the losing team. Additionally, it is worth noting that on the defending side, both weapons and utilities are slightly more expensive compared to the attacking T side, hence the higher average equipment value

for the defenders shown on Figure 2.5(b). Consequently, when the defenders recognize that they are at a significant disadvantage, they often opt to concede the round to the opponent and focus on preserving their weapons and gear for future rounds. This strategy explains the higher average number of players alive for the defenders.

Chapter 3

Model training

3.1 Introduction to Graph Neural Networks

Deep learning, a subfield of machine learning, is a leading research field in 2024 and is widely used in practical applications as well. It focuses on leveraging artificial neural networks to automatically discover patterns in large datasets. Neural networks, the foundation of deep learning, are computational models inspired by the structure of the human brain. These networks consist of interconnected layers of perceptrons, often referred to as neurons in the literature. Each of the neurons receives and processes information before passing it to subsequent layers, progressively abstracting the input data. This enables the model to learn complex patterns and make accurate predictions.

Various types of neural networks are specialized to excel in different tasks. Convolutional Neural Networks (CNNs), for example, are widely used for image analysis due to their ability to recognize spatial hierarchies in data. Meanwhile, Large Language Models (LLMs) are designed for natural language processing, capable of understanding and generating text. Another unique model type within deep learning is the Graph Neural Network (GNN).

A GNN can be viewed as a generalization of CNNs, where images represent a specific case of structured graph data. Unlike pictures with pixels uniformly organized into a grid, GNNs work with data represented by nodes and links connecting them. In these graphs, nodes represent individual entities, and edges capture the relationships or connections between them. The nodes often have a feature set vector in order to store information about the entity they represent. Different edge variations are also widely used in different problem domains, such as directed, weighted or signed edges.

Graph neural networks are designed to discover patterns in these graphs. They fundamentally operate through matrix multiplication, leveraging these operations to learn the information conveyed by edges. Therefore, input graph to these models can be described by the following matrices:

$$X = \begin{pmatrix} x_{11} & \cdots & x_{1m} \\ x_{21} & \cdots & x_{2m} \\ x_{31} & \cdots & x_{3m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{pmatrix}, \quad X \in \mathbb{R}^{n \times m} \quad A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad A \in \mathbb{R}^{n \times n},$$

where X is the feature matrix of the nodes, A is the adjacency matrix, n represents the number of nodes and m denotes the number of features in a node.

Graph neural networks operate by convolving information across nodes through the edges, facilitating a flow of information that enables each node to consider its neighbors' features. This iterative process allows the model to capture both local and global patterns within the graph structure. After the convolutional layers, the aggregated information can be passed through a Multi-Layer Perceptron (MLP) to further process the learned features and yield the final output. Through this sequence of operations, GNNs effectively learn to model the underlying relationships in graph-structured data. These neural networks are highly versatile and can be applied to a variety of practical use cases, including

- **Node Classification/Regression:** Predicting or estimating a label or value for each node in the graph, such as classifying users in social networks by interests or predicting molecular properties of atoms in molecular graphs.
- **Link Prediction:** Identifying potential connections between nodes, often used in social networks to suggest friends or in knowledge graphs to infer missing relationships.
- **Edge Classification:** Predicting the type or strength of relationships represented by edges, which can be useful in applications such as knowledge graphs, where each edge represents a type of relationship.
- **Graph Classification:** Assigning a label or category to entire graphs, useful in applications like molecular activity prediction in drug discovery, where each molecule is treated as a graph with specific properties.
- **Community Detection:** Discovering groups or communities within a network, helpful in fields like social network analysis, where communities may represent clusters of users with similar interests.
- **Graph Sampling:** Efficiently sampling subgraphs from larger graphs to reduce computational complexity during training and inference, often used in large-scale graphs to speed up learning while retaining key information.

Graph Neural Networks have evolved into various specialized variants, each tailored to handle unique graph structures and solve specific tasks. Heterogeneous GNNs (HGNNs) are designed to learn on heterogeneous graphs, where a distinctive feature is that multiple types of nodes and edges are present. This introduces the complexity of having multiple node feature X and adjacency A matrices in the input, one for each node and link type, respectively [10]. Another interesting quality of heterogeneous graphs compared to regular ones, is that the adjacency matrices are not necessarily square, due to the varying number of nodes per node type. The added layers of complexity in HGNNs enable convolutions to be applied individually to each edge type, allowing for distinct convolution operations tailored to each link type within the graph. It even allows for variation in the number of convolutional layers used for each edge type, adapting the model more precisely to the structure of the heterogeneous graph.

Another notable variant is Temporal Graph Networks (TGNs). TGNs are an advanced class of GNNs designed to model and learn from dynamic graph structures where nodes, edges, and their features change over time. These models capture the evolution of relationships and node states, which is essential in applications like social network analysis

or recommendation systems. TGNs take dynamic graphs as input to the network. These evolving graphs can be represented in two main formats:

- **Discrete-Time Dynamic Graphs (DTDGs)** are sequences of graph snapshots taken at regular intervals. In each snapshot, nodes and edges capture the state of the network at a particular point in time. This format is particularly well-suited to applications where data arrives in periodic intervals, such as daily trading networks or hourly traffic data.
- **Continuous-Time Dynamic Graphs (CTDGs)**, on the other hand, track interactions as they occur, using continuous timestamps for each event (e.g., a new edge being formed or a node attribute changing). CTDGs model the network evolution as an uninterrupted sequence of events, enabling more granular modeling of dynamic relationships. This approach is suitable for highly dynamic environments, such as social media interactions, where events happen sporadically rather than at fixed times.

TGNs leverage techniques from recurrent neural networks (RNNs), particularly through GRU (Gated Recurrent Unit) or LSTM (Long Short-Term Memory) layers, to capture temporal dependencies [10]. These layers allow the TGN to learn how node features and connections change over time, encoding this sequential information to make accurate predictions. In a TGN, a temporal convolution layer, such as a GRU or LSTM, is typically applied after the spatial convolution to capture time-dependent patterns within each node’s feature history. This layering enables TGNs to learn both spatial and temporal patterns effectively. Moreover, by combining spatial and temporal learning, TGNs can be applied to various time-sensitive tasks, such as predicting future links in a network, identifying the next likely interaction, or understanding trends across evolving systems. Their design bridges the capabilities of recurrent neural networks for sequence modeling and graph neural networks for relational data, making TGNs a powerful tool for dynamic graph analysis.

3.2 Model architecture

With the data preparation phase completed, attention turned to neural network training. The objective is to predict whether the CTs or Ts will win a round based on the current game state, represented by a heterogeneous graph. This is formulated as a binary graph classification task, for which I opted to deploy HGNNs. To construct these deep learning models I utilized Pytorch Geometric, as it provides a wide variety of tools for processing and analyzing graph-structured data.

Setting out to build a HGNN, the initial convolution strategy revolved around spreading information from the player nodes first and convolving them across the map. The core idea is to identify the players as the source of the information since they are the key drivers of events occurring on the map. In this context, the map nodes act as a surface that allows information from the player nodes to propagate, effectively illustrating the dynamic interactions and impacts generated by player actions.

With the initial architectural strategy for convolutions set, development has proceeded to the implementation phase.

3.2.1 Heterogeneous convolutions

Incorporating heterogeneous graphs into the framework has introduced substantial complexity in the range of potential convolution layer variations and orderings. To address this, the convolutions are defined at a higher level of abstraction. A *composite* or *heterogeneous convolution* layer is formally defined as an ordered sequence of convolution operations over specified edge types within the graph. This enables unified handling of convolutions within the heterogeneous graph.

Specifically, the initial strategy consisted of two main steps: the first propagation is through the *player connected_to map* edges. Player nodes update their neighboring map-type nodes with this operation, incorporating their own feature data into the latter node. Subsequently, map nodes propagate information among themselves using the *map connected_to map* links between them. However, it was noted in the data preparation section that an additional self-edge for the player nodes, labeled as *player is player*, was incorporated into the graphs. This addition is significant because although the player nodes serve as the primary sources of information, they do not undergo updates according to the initial convolution strategy. Furthermore, this node type contains over 160 features, many of which contribute little to the overall representation of the player. For example, the player’s inventory is represented using boolean flags indicating whether the player possesses a specific item (over 40 features). However, most of these feature values are zero, as there is a limit to the number of inventory items that can be stored based on the item types, yet all is important to represent the player accurately. Additionally, there are general player performance-related features that also exhibit correlations with one another. Consequently, dimension reduction techniques were initially considered to represent the player with a more efficient selection of features. Nevertheless, in an effort to optimize the deep learning process, self-loop edges were added to the player nodes instead. This modification allows the GNN models to perform dimension reduction on the player nodes themselves before any other convolution occurs. This approach ensures that the updated representations of the player nodes retain the most critical features relevant to the learning process. This self-convolution of the player nodes serves as an optional convolution strategy, enriching the hyperparameters of the neural network to optimize in the future.

With this extension, a heterogeneous convolutional layer is defined as outlined in the following sequence. At each stage, the edge type through which the convolution is applied is indicated in brackets.

1. **Player dimension reduction** (*player is player*): a $Conv_1$ convolution creates a p' lower dimensional representation of the player vector p .
2. **Player updates map** (*player closest_to map*): the information of the player’s p' vector gets propagated to the neighboring map node’s vector m through convolution $Conv_2$. The updated map vector is denoted by m' .
3. **Spread information throughout the map** (*map connected_to map*): the map nodes update their m' representation by incorporating the neighbors’ information using some convolution $Conv_3$. Note that m'' represents a map node’s feature vector after convolution $Conv_3$, and $|m'| = |m''|$.

In these heterogeneous convolutions, each operation is executed once in the defined order. The output vector size treated as a tunable parameter to be optimized for each convolution at each layer.

In the initial network, graph SAGE [2] operators were utilized in all $Conv_n$ convolution scenarios. Equation 3.1 describes the mentioned operator.

$$\mathbf{x}'_i = \sigma(\mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \text{Agg}(\mathbf{x}_j : j \in \mathcal{N}(i))) \quad (3.1)$$

The SAGE convolution updates the node representations in two steps. First, it retains information about the original node’s feature vector x_i by multiplying it with a learnable W_1 weight matrix. In the second step, it collects the node’s neighbors’ vectors, aggregates them with some aggregation function $\text{Agg}(\cdot)$, and incorporates this vector into the original node by multiplying it with another learnable W_2 matrix. After the updates, a $\sigma(\cdot)$ activation function is applied on the output vector.

Following the heterogeneous convolutional layers, additional convolutions along the *map connected* to *map* links further spread the information of the players throughout the map. The number of these extra convolutions along with the output feature dimensionality are hyperparameters to optimize. The updated graph, now containing node embeddings, is then flattened into a single vector. The first segment of this held the embedded features of the player nodes, followed by the map nodes, with the final portion comprising the overall graph-level features. This graph vector is then passed through a series of dense layers, where both the number and size of the layers remained tunable, along with the choice of activation function for each. The final layer consisted of a single perceptron with a sigmoid activation, as the task was binary classification. With this architecture, the model was fully prepared for deployment.

3.3 Hyperparameter optimization and HGNN results

3.3.1 Neural scaling laws in hyperparameter tuning

Neural scaling laws describe how the performance of neural networks improves predictably as certain model parameters such as dataset size (D), model capacity or parameter count (N), and available compute (C) are increased. The concept gained traction when it was observed that, for certain deep learning models, performance metrics such as loss or error rate follow a power-law relationship with respect to D , N and C . These scaling laws have proven useful in practice, as for example, Kaplan et al. [3] demonstrated for LLMs that model depth has only a minimal impact on performance compared to the number of parameters, which is the more influential factor. This essentially narrows the hyperparameter search space and facilitates model training [6]. Working with graphs, however, presents a distinct environment compared to images or natural language processing. In the realm of GNNs, scaling laws are not as easily definable as in other domains and is still an active research topic. Nevertheless, investigating these laws could facilitate a more precise hyperparameter search, potentially enhancing model performance by identifying optimal configurations.

Liu et al. [6] stated that in the case of graph neural networks, the depth of the model plays a crucial role in the training process, distinguishing it from LLMs. This arises due to phenomena such as over-smoothing and over-squashing. Over-smoothing occurs when the features of nodes become indistinguishable as information is aggregated across many layers, leading to a loss of valuable distinctiveness in the graph. On the other hand, over-squashing refers to the compression of information from distant nodes as it is propagated through multiple layers, causing the model to struggle to capture important

relationships and dependencies. This insight emphasizes the importance of examining the optimal number of convolutional layers before tuning other parameters.

To explore this topic, an initial HGNN model was constructed. Around 40% of the data was sampled randomly and used for this experiment. Subsequent testing revealed that models with around 10^5 learnable parameters demonstrated sufficient capacity to achieve reasonable performance in the classification task. Thus, the search was conducted while ensuring that the number of learnable parameters remained around 10^5 , focusing solely on varying the number of convolutional layers of the map nodes and adjusting the learning rate. The reduction in player dimensions, followed by convolving player information to the map nodes, was fixed at two layers with output vector sizes 15 and 5, respectively, as these parameters do not significantly influence the over-smoothing problem. The output vector size of the convolutions was also fixed in order to solely focus on the layer numbers. The Binary Cross Entropy (BCE) function measured the loss, and the AdamW was utilized as the optimizer. AdamW [7] is a numerically stable variant of the Adam optimizer designed to maintain lower weight values. Each model’s available compute time C was fixed to 10 epochs. Figure 3.1 concludes the output of the testing.

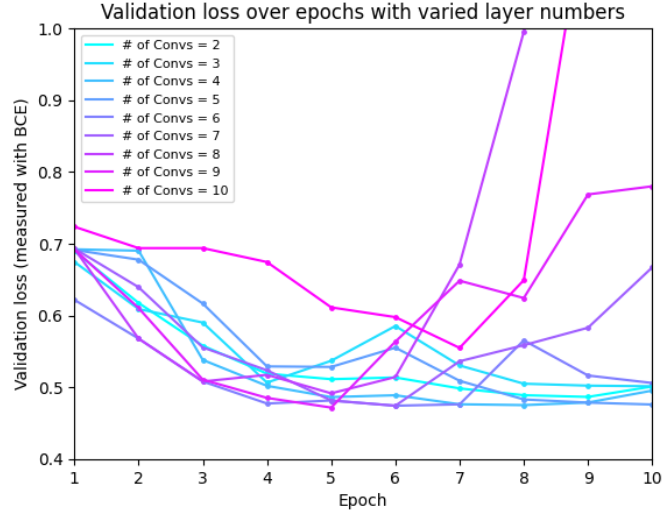


Figure 3.1: Validation losses of HGNNs with fixed hyperparameters and similar parameter number N , across varying numbers of convolutional layers.

The results indicate that the optimal number of convolutions along the map type edges are between 2 and 6. Above 7 convolutions, the loss tends to increase highly after some epochs and the network fails to capture the important patterns. This indicates that over-smoothing becomes a problem above 7 convolutions.

3.3.2 Hyperparameter tuning

With the range of convolutional layers defined, attention then turned to tuning the network’s remaining parameters. Table 3.1 summarizes the hyperparameter search space explored, excluding those specific to the dense layers after the convolutions.

In the table, it is also highlighted that the list of used convolutional operators was extended to include Graph Convolutional Network (GCN) and Graph Attention (GAT) operators.

Parameter	Intervals / Values
Learning Rate (γ)	$\gamma \in \mathbb{R}^+, \quad 10^{-8} \leq \gamma \leq 10^{-1}$
L2 Regularization (λ)	$\lambda \in \mathbb{R}^+, \quad 10^{-8} \leq \lambda \leq 10^{-1}$
# of Heterogeneous conv. layers (L_H)	$L_H \in \mathbb{N}, \quad 2 \leq L_H \leq 7$
# of Map node conv. layers (L_m)	$L_m \in \mathbb{N}, \quad 0 \leq L_m \leq 4$
Embedded player vector size at layer l ($ p'_l $)	$ p'_l \in \{5, 10, 15, \dots, 70\}$
Embedded map vector size at layer l ($ m'_l $)	$ m'_l \in \mathbb{N}, \quad 3 \leq m'_l \leq 20$
Attention heads at layer l (if GAT is used) (H_l)	$H_l \in \mathbb{N}, \quad 1 \leq H_l \leq 5$
Convolution operators	GraphSAGE, GCN, GAT
Activation functions after convolutions	ReLU, Leaky ReLU, Parametric ReLU

Table 3.1: Parameter intervals and values for hyperparameter tuning.

The GCN update rule for a node i is given by Equation 3.2 [4].

$$\mathbf{x}'_i = \sigma \left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{\Theta} \mathbf{x}_j \right). \quad (3.2)$$

In this formula, $\mathbf{x}'_i$ represents the updated feature vector of node i following the convolution operation. The symbol σ denotes an activation function, applied element-wise after the aggregation. The summation $\sum_{j \in \mathcal{N}(i) \cup \{i\}}$ aggregates information from the neighborhood of node i (denoted by $\mathcal{N}(i)$), including the node itself. Here, $e_{j,i}$ represents the weight of the edge between nodes j and i , allowing for differentiation in edge importance; in cases where the graph is unweighted, $e_{j,i}$ can be set to 1. The learnable weight matrix $\mathbf{\Theta}$ is applied to the feature vectors, transforming them before aggregation. The terms $\hat{d}_j$ and $\hat{d}_i$ denote the degrees of nodes j and i , respectively, inclusive of self-loops. The normalization factor $\frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}}$ is essential to ensure that the aggregation respects each node's degree, thus mitigating issues of over-smoothing. This formulation enables GCNs to generate node embeddings by aggregating information from neighboring nodes in a manner that accounts for the connectivity of each node through weighted and normalized aggregation.

Equation 3.3 defines the GAT update rule for a node i .

$$\mathbf{x}'_i = \sigma \left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j} \mathbf{\Theta} \mathbf{x}_j \right). \quad (3.3)$$

In this expression, $\mathbf{x}'_i$ denotes the updated feature vector of node i after the attention mechanism is applied. The function σ , often implemented as LeakyReLU in GAT architectures, acts as an activation function applied following the aggregation. The term $\alpha_{i,j}$ represents the attention coefficient between nodes i and j , determining the relative importance of node j 's features in updating node i [9]. The learnable weight matrix $\mathbf{\Theta}$ transforms the

feature vectors, with each transformed feature then weighted by the respective attention coefficient $\alpha_{i,j}$. The attention mechanism thus allows GATs to dynamically assign different weights to each neighboring node, emphasizing nodes that contribute more relevant information for the specific task, thereby enhancing the flexibility and expressiveness of the network.

With the hyperparameter ranges defined, the hyperparameter search was conducted using the Optuna Python library. This package offers a suite of algorithms designed to optimize the search process, effectively managing the learning workflow to maximize the efficiency and accuracy of hyperparameter tuning. By leveraging Optuna’s capabilities, it was possible to systematically explore the parameter space, ultimately facilitating a more targeted and resource-efficient approach to model optimization.

3.3.3 Training Results

The dataset used for training consisted of 700,000 graphs generated from processed matches scraped since Q2 of 2023, organized into min-batches. The dataset was split into training and validation sets in a 75-25 ratio; however, this split was conducted at the match level rather than at the snapshot level. This approach was necessary because if the 700,000 graphs were shuffled and then randomly split, both the training and validation sets would contain snapshots from the same rounds within the same matches. Such overlap would increase the risk of reduced generalization ability. To mitigate this risk, the training set comprised snapshots from the first 80 matches out of the 120 scraped ones, while the remaining matches were reserved for validation. This method ensures that the model is validated on rounds of matches it has not encountered during training, supporting a more robust evaluation of its performance.

The training process was conducted in multiple phases, with each iteration further narrowing the parameter ranges to identify the most optimal solution. As the parameters became more refined, the models demonstrated increasingly successful performance. Generally, it can be stated that the better-performing models were able to achieve an accuracy above 70% after around only 2-3 epochs, with the best models reaching the milestone of 75%. Table 3.2 summarizes the hyperparameters for the most successful training run, while Table 3.3 presents the achieved results assessed through different metrics.

The best model achieved an accuracy exceeding 75%, a noteworthy performance given the complexity of the task. It also outperformed Peter Xenopoulos’s model across all metrics, including Log Loss, Brier Score, and AUC, as reported in their study [11]. Besides the best configuration, there were several additional models that reached the milestone of 75%, the remaining successful models performed just below that. Although these results are remarkable by all measures, the model still has limitations; it learns patterns solely from static snapshots rather than temporal sequences. To address this, the focus has shifted towards incorporating temporal patterns into the learning process.

3.4 Temporal learning

3.4.1 Temporal dataset creation

The initial step in learning temporal patterns is to construct a temporal dataset, composed of dynamic graphs. These graphs vary over time, and as noted above, two primary types have emerged in practice: discrete-time dynamic graphs (DTDG) and continuous-time

Parameter	Value
Learning Rate (γ)	5.341×10^{-2}
L2 Regularization (λ)	5.496×10^{-2}
# of epochs	4
Batch Size	200
# of Heterogeneous conv. layers (L_H)	2
# of Map node conv. layers (L_m)	3
Embedded player vector size at layer l ($ p'_l $)	50, 20
Embedded map vector size at layer l ($ m'_l $)	20, 15, 10, 10, 5
Convolution operators (every convolution)	GAT
Attention heads at layer l (H_l)	1
Activation functions (every activation)	Leaky ReLU

Table 3.2: Hyperparameters of the most successful model.

Metric	Value
Log Loss	0.4661
Accuracy	75.28%
Precision	71.59%
Recall	80.61%
F1 score	75.83%
Brier score	0.1564
AUC	85.3%

Table 3.3: Achieved results on the validation set.

dynamic graphs (CTDG). Since the original dataset consists of consecutive snapshots taken at regular intervals, storing dynamic graphs in the DTDG format is a natural choice. This decision is further supported by the fact that numerous elements can change between snapshots (e.g., player positions, active weapons, health points, etc.), which would require a significant number of events in a CTDG format. Additionally, these events would occur at fixed time intervals, effectively nullifying the benefit of using the CTDG format.

A DTDG is obtained by concatenating n consecutive static snapshot graphs in temporal order, each representing a distinct time step. The initial dataset was designed with a fixed interval length for dynamic graphs, leaving the option to include variable-length dynamic graphs as a future development. The choice of interval length required careful consideration: excessively long intervals may fail to emphasize critical events, such as gunfights, while overly short intervals may lack meaningful temporal patterns and information. As a result, an interval length of 5 seconds was selected for the initial dataset, which, given the game-state’s representation at 0.25-second intervals, corresponds to a sequence of 20 consecutive static graphs. Thus, each dynamic graph comprises 20 temporally ordered static graphs.

As round durations vary based on events, in most cases, the number of snapshots per round is not divisible by 20. To address this issue, a strategy was implemented wherein the dynamic graphs are constructed from the end of each round. As a result, this method may exclude up to the first 4.75 seconds (or 19 snapshots) of a round. This omission is considered acceptable, as the initial 10-20 seconds of each round typically involve initial player positioning and lineup setup. However, to further enhance data diversity, the temporal dataset creator processes each round twice: once from the end and once starting 2.5 seconds before the end. This strategy enriches the temporal dataset by capturing the same events in two intervals, allowing the model to learn patterns that appear at different stages within the time windows. Consequently, this solution ensures that only a negligible portion

(up to 2.25 seconds) of each round’s beginning is excluded from the temporal dataset. By applying this strategy to concatenate data according to time intervals, the initial dataset was created with 5-second interval dynamic graphs, amounting to approximately 90,000 DTDGs in total.

3.4.2 Extending the model architecture

With the creation of the temporal dataset, focus shifted towards expanding the current HGNN architecture. The objective is to develop it to a Temporal Heterogeneous Graph Neural Network (THGNN) that expects dynamic graphs as input, enabling it to learn temporal patterns within the data. The initial concept was to introduce temporal learning between the convolutional layers and the dense layers, thus allowing the spatial patterns learned by the convolutions to be further analyzed temporally. To achieve temporal pattern learning, a Graph Convolutional Gated Recurrent Unit (GRU) operator was employed.

In traditional recurrent neural networks, the GRU is a simplified variant of the Long Short-Term Memory (LSTM), designed to manage sequential data by leveraging a reset gate and an update gate. These gates control the flow of information through time by selectively updating hidden states, allowing the network to retain relevant information and discard irrelevant features. Given an input $\mathbf{x}_t$ at time step t and a hidden state $\mathbf{h}_{t-1}$ from the previous time step, the GRU updates the current hidden state $\mathbf{h}_t$ as follows:

$$\begin{aligned} \mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1}) \\ \mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1}) \\ \tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h (\mathbf{r}_t \odot \mathbf{h}_{t-1})) \\ \mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \end{aligned}$$

where $\mathbf{z}_t$ is the update gate that determines how much of the previous state $\mathbf{h}_{t-1}$ should carry over, $\mathbf{r}_t$ is the reset gate that controls the influence of the previous state on the current candidate hidden state $\tilde{\mathbf{h}}_t$, and $\odot$ denotes element-wise multiplication. This structure enables the GRU to capture temporal dependencies in sequential data by learning which features to keep or reset over time.

In the graph-based GRU variant, these operations are extended to account for the spatial structure of the graph. Here, node representations not only evolve through time but also consider the influence of neighboring nodes at each time step. By incorporating this information at each step, the graph GRU enables a more robust representation of node feature evolution, capturing both the temporal progression of individual node features and the influence of spatially connected nodes over time. This formulation allows the THGNN to effectively model the temporal patterns in node feature vectors across consecutive snapshots while also considering the influence of neighboring nodes.

Given that the input graphs are heterogeneous with two distinct node types, the network was extended with two separate GRU convolutional layers. One is dedicated to capturing the temporal patterns of map-type nodes, while the other focused on player nodes. With this addition, the neural network is now equipped to learn temporal patterns effectively. After the HGNN’s convolution operations are applied across all 20 snapshots of a DTDG, the graphs with the embedded nodes pass through the GRU convolutional layers in temporal order, allowing the model to learn the temporal dependencies within the data. After

the node feature vectors are updated temporally, the same flattening operation creates the embedded vector graph, which is then passed through dense layers to make the final prediction.

3.4.3 Mini-batch learning on dynamic graphs

In neural network training, a commonly used method to enhance efficiency is to process data in mini-batches rather than individually, as with Stochastic Gradient Descent (SGD). This mini-batching approach leverages the parallel computation capabilities of modern GPUs, facilitating faster training by processing data in batches. In the field of graphs, the construction of batches uses a unique technique. Suppose we have n graphs, each defined by an adjacency matrix A_i and a feature tensor X_i , where $i \in \{1, n\}$. The standard approach to mini-batching graphs, as shown in Equation 3.4, involves concatenating the X_i feature tensors vertically and creating a larger, block-diagonal adjacency matrix from the individual A_i matrices. The result is a single large graph comprising n disconnected components. In this configuration, the new matrices X and A are constructed to ensure that separate components (the original graphs) cannot influence each other, only themselves. Thus, this approach enables mini-batch learning on graphs, a technique natively supported in PyTorch Geometric. The heterogeneous nature of the graphs in this specific application does not present any further challenges either as the library offers solutions for it. However, the introduction of dynamic graphs added complexity to the batching process.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix} \quad (3.4)$$

Essentially, a DTDG can be viewed as a mini-batch of sequential static graphs. Due to this, dynamic graphs were used in the training as batches of sequential non-temporal snapshot graphs. This approach allows spatial convolutions to be performed first in a batch manner, followed by temporal convolutions applied in sequential order to the static graphs. However, in this setup, each mini-batch corresponds to a single DTDG during training, effectively implementing the SGD method. This setup caused the runtime to be slow. For instance, training on 700 dynamic graphs with SGD took 14 minutes per epoch on the THGNN, whereas in the HGNN, training on approximately 500,000 static graphs with mini-batching took 10 minutes per epoch (the extra time is partly due to temporal convolution overhead, but largely due to the inefficiency of the SGD approach).

To address this issue, mini-batch learning needed to be implemented for dynamic graphs. In this context, a mini-batch would contain multiple DTDGs, effectively creating a mini-batch that consists of temporal mini-batches. To achieve this, a custom data loader class was derived from PyTorch, which stores and passes the graphs in single array format (for example, a batch of size 40 is represented as a list of 40 elements, each of which is a list of 20 HeteroData objects). The formatting of the matrices, as well as the use of the original PyTorch Geometric data loaders, were implemented in the network’s forward method. Throughout this process, the goal was to optimize training with the minimal number of operations, which required the appropriate use of matrices, thus the need to code it in the forward pass.

Let B denote a mini-batch provided by the custom data loader class. B consists of n dynamic graphs $\mathcal{G}_i$, where $i \in \{1, \dots, n\}$. Each $\mathcal{G}_i$ graph consists of a sequence of

static graphs $\mathcal{G}_1^i, \mathcal{G}_2^i, \dots, \mathcal{G}_t^i$. The letter in the upper index identifies the dynamic graph containing the examined static graph. For each static graph $\mathcal{G}_k^i$ (where $k \in \{1, \dots, t\}$), the adjacency matrices A_k^i and feature tensors X_k^i are stored. The first step when B is input into the THGNN is to apply spatial convolutions on each graph. To achieve this, the original PyTorch data loader is utilized. Since these spatial convolutions are executed independently for all $\mathcal{G}_k^i$, a batch of size $n \times t$ is created to store all static graphs. This approach is consistent with the batch learning method used in the HGNN. Equation 3.5 shows the resulting A and X matrices (for simplicity, only one of each are illustrated, as there are more A and X matrices in heterogeneous graphs).

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1^1 & & & & & \\ & \mathbf{A}_2^1 & & & & \\ & & \ddots & & & \\ & & & \mathbf{A}_t^1 & & \\ & & & & \mathbf{A}_1^2 & \\ & & & & & \ddots \\ & & & & & & \mathbf{A}_t^2 \\ & & & & & & & \ddots \\ & & & & & & & & \mathbf{A}_t^n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1^1 \\ \mathbf{X}_2^1 \\ \vdots \\ \mathbf{X}_t^1 \\ \mathbf{X}_1^2 \\ \vdots \\ \mathbf{X}_t^2 \\ \vdots \\ \mathbf{X}_t^n \end{bmatrix} \quad (3.5)$$

Following the spatial layers, the GRU operators handle the temporal convolutions. To learn temporal patterns, the static graphs comprising each dynamic graph must be fed to the GRU sequentially, which is not feasible with the current $n \times t$ batch structure. Therefore, after the spatial convolutions, the network splits the batch, restoring the original $n \times t$ arrangement of static graphs, or, equivalently, n dynamic graphs. At this state, it was considered to iteratively process each dynamic graph $\mathcal{G}_i$, feeding each of its static graphs $\mathcal{G}_k^i$ to the GRU layer in temporal order. However, this approach would implement the SGD algorithm and would neglect the fact that all dynamic graphs have equal length, allowing their constituent static graphs to be organized into batches based on their time step.

To enhance learning efficiency, time-batches, denoted by T , were created. A time-batch T_k collects all $\mathcal{G}$ static graphs with the same time stamp k from all n dynamic graphs and creates a batch for them. Essentially, k mini-batches are created, one for each time stamp. As illustrated in Equation 3.6, each T_k batch groups the graph matrices based on identical time stamps, allowing them to be processed collectively. Passing these through the GRU layer sequentially in temporal order maximizes computational efficiency during training.

$$\mathbf{A}_{T_k} = \begin{bmatrix} \mathbf{A}_k^1 & & & \\ & \mathbf{A}_k^2 & & \\ & & \ddots & \\ & & & \mathbf{A}_k^n \end{bmatrix} \quad \mathbf{X}_{T_k} = \begin{bmatrix} \mathbf{X}_k^1 \\ \mathbf{X}_k^2 \\ \vdots \\ \mathbf{X}_k^n \end{bmatrix} \quad (3.6)$$

Once the GRU convolutions are complete, the original dynamic graph structures are restored and the network proceeds similarly to the HGNN. The graphs are flattened out into vectors and then passed through the dense layers in an $n \times t$ batch. This implementation of mini-batch learning for dynamic graphs achieved a considerable performance improvement, reducing runtime from 14 minutes with SGD on 700 dynamic graphs to 50 minutes for 60,000 dynamic graphs processed in batches of 200. With the learning time improved

by several orders of magnitude, hyperparameter optimization within the network could proceed effectively.

3.4.4 Results

To find the most effective THGNN model, I applied the same hyperparameter search procedure as with the HGNN, although the list of parameters to be tested expanded with the introduction of temporal learning. For simplicity, I defined the output vector sizes of the GRU layers following the convolutional layers to match the input vector sizes, effectively making these layers perform a temporal update on the node features. However, the GRU layers have a parameter K that required tuning. This K represents the Chebyshev filter size, which determines the size of the K -hop neighborhood considered when updating each node in the GRU. Table 3.5 summarizes the new parameters and their respective ranges.

Parameter	Value
Learning Rate (γ)	4.772×10^{-3}
L2 Regularization (λ)	2.772×10^{-6}
# of epochs	3
Batch Size	200
# of Heterogeneous conv. layers (L_H)	2
# of Map node conv. layers (L_m)	2
Embedded player vector size at layer l ($ p'_l $)	30, 8
Embedded map vector size at layer l ($ m'_l $)	30, 20, 10, 5
Player GRU Chebyshev filter size (K_p)	3
Map GRU Chebyshev filter size (K_m)	3
Convolution operators (every convolution)	GAT
Attention heads at layer l (H_l)	1
Activation functions (every activation)	Leaky ReLU

Table 3.4: Hyperparameters of the most successful model.

Parameter	Intervals / Values
Player GRU Chebyshev filter size (K_p)	$K_p \in \mathbb{N}, \quad 1 \leq K_p \leq 5$
Map GRU Chebyshev filter size (K_m)	$K_m \in \mathbb{N}, \quad 1 \leq K_m \leq 5$

Table 3.5: Additional hyperparameters introduced with temporal learning.

Metric	Value
Log Loss	0.4587
Accuracy	76.09%
Precision	74.86%
Recall	74.41%
F1 score	74.64%
Brier score	0.1561
AUC	85.5%

Table 3.6: Achieved results on the validation set.

Although the inclusion of temporal patterns increased training time, it improved model performance. Table 3.4 presents the hyperparameters of the most successful THGNN training, while Table 3.6 displays the achieved results. The ability to learn temporal patterns led to the highest model performance, surpassing 76% accuracy. Similar to the HGNN, the most accurate models achieved approximately 75% accuracy; however, the best THGNN model was the only one to reach the milestone of 76%. With temporal learning successfully integrated into the prediction task, a comparison between the HGNN and THGNN was conducted to assess the effect of temporal learning on the predicted outcomes.

3.5 Comparing model predictions

To examine the impact of learning temporal patterns, comparing the two models is an essential step. The most straightforward approach for this comparison is to evaluate their performance by comparing the predicted probabilities for the same rounds. Figure 3.2 presents the progression of winning probabilities predicted by each model in a selected round. The figure clearly shows that both models suggest similar values, indicating that their predictions are generally consistent. Additionally, it is apparent that the model trained on temporal patterns exhibits considerably more fluctuations than the standard model. This is intuitive, as the temporal model is more sensitive to in-game events and context, capturing finer temporal details.

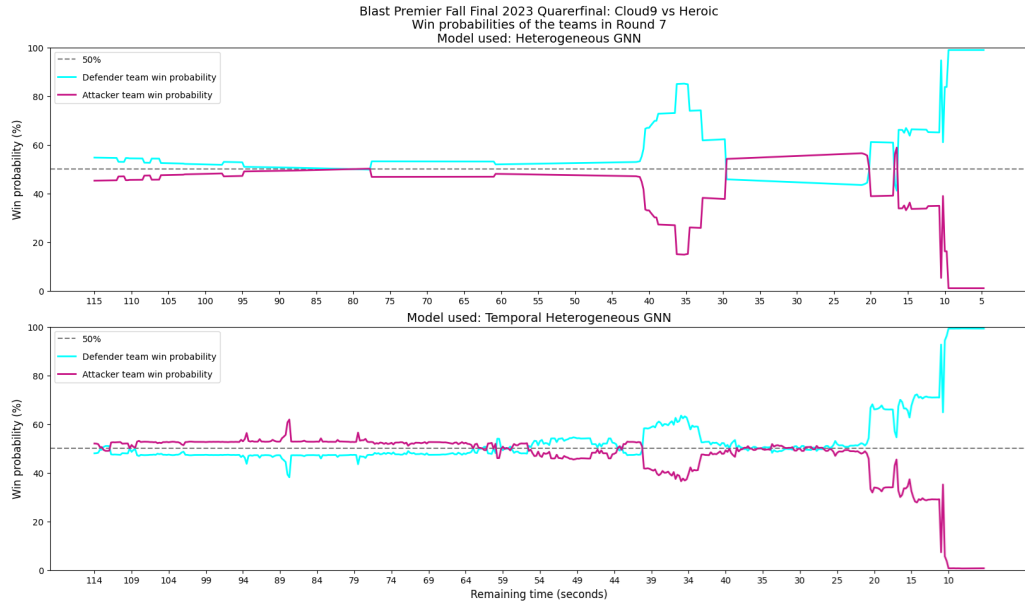


Figure 3.2: Predicted team winning chances compared between HGNN and THGNN for the same selected round.

The figure reveals further interesting patterns. For instance, there are noticeable spikes in the probabilities predicted by the temporal model at specific times (e.g., around 94 and 85 seconds remaining). Upon comparing these spikes with the recorded and live-broadcasted match footage, their exact cause could not be determined, necessitating further investigation, which will be conducted in the sections on model explanation and player evaluation. However, certain other differences are already explainable based on the match recording. For example, the HGNN model shows a temporary decrease in the attackers' win probability around the 35-second mark. The footage reveals that this occurs as the attackers, after reaching the bomb site, transfer the bomb from Player A to Player B, who is in a better position to plant it. During this transfer, the bomb is momentarily neither held by a player nor placed on the ground, leading the HGNN model to interpret this as a loss of possession (since it has no information about the bomb's trajectory while airborne). Consequently, the model predicts a decrease in win probability. However, as soon as Player B picks up the bomb, this disadvantage is eliminated, and the predicted win chance returns to its previous level. In contrast, the temporal model shows no such drop in probability, as it understands that the bomb will soon reach another player, allowing it to provide more realistic probabilities.

There are also more significant jumps in the win probabilities predicted by the models, typically linked to elimination events (e.g., at 39, 21, 16, and 11 seconds). This is expected, as gaining or losing players has one of the most substantial impacts on the round's outcome and thus on the win probabilities. These observations suggest that both models have learned realistic patterns related to game flow and, at first glance, provide valuable predictions when examining multiple rounds and comparing them with match footage. In Chapter 4, the fluctuations of the winning chances and the reasoning behind the models are explored in greater detail to link changes to specific player actions, thereby enabling player evaluation.

Chapter 4

Evaluating player actions

Another important topic addressed by this paper is the evaluation of player actions using the trained GNNs. This involves developing approaches to understand and interpret the model predictions. Such insights provide the opportunity to link changes in the predicted probabilities to the actions of the players. This capability enables the assessment of their actions by examining the impact those actions have on their team’s winning chances.

4.1 Explainable AI

Explainable Artificial Intelligence (XAI) has emerged as one of the most crucial aspects in today’s AI landscape. Its importance lies in enhancing trustworthiness, ensuring traceability, and distinguishing between explainability and interpretability. Trustworthiness is vital as users are more likely to adopt AI systems that they can understand and trust, while traceability allows stakeholders to track the decision-making process of the model, thereby enabling accountability. Furthermore, the distinction between explainability and interpretability is significant; while explainability refers to the clarity with which a model’s predictions can be understood, interpretability involves how easily the underlying model can be grasped by humans.

In the realm of deep learning models, this area raises particularly intriguing questions, as these models often learn complex patterns at the expense of interpretability, making it challenging to reverse-engineer specific predictions and understand the reasoning behind the model’s outputs. Traditional neural networks and CNNs have various tools available for this purpose, such as LIME (Local Interpretable Model-agnostic Explanations), which approximates the model locally with an interpretable model to explain individual predictions by examining the effects of perturbations in the input features. Another widely used technique is SHAP (SHapley Additive exPlanations), which provides a unified measure of feature importance based on cooperative game theory principles, ensuring that the contribution of each feature to the prediction is fairly assessed. However, the explainability of graph neural networks remains an actively researched area, presenting numerous additional challenges compared to standard models due to the intricate relationships and structures within graph data. For this reason, the explanatory part of the model predictions in this paper is conducted using tabular data formats.

4.2 The reason behind probability fluctuations

The player evaluation approach presented in this paper is based on the intuitive notion that the fluctuations in win probabilities estimated by the models are linked to player actions and, indirectly, to their decisions. If this assertion holds true, it follows that if no significant actions occur on the field, the estimated probability should remain stable. This hypothesis can be easily verified with the successfully trained models by creating static rounds and predicting winning chances for them. A static round is an artificially constructed round where no actions take place, all players remain in their starting positions throughout the round. In such rounds, according to the game’s rules, the defending team will always win, as the attackers cannot complete their objectives before time expires. To test this hypothesis, a module was developed that takes an arbitrary round, captures its initial timestamp, nullifies any values related to speeds or movements, and creates a version of the round where all players are in their starting positions, with no actions occurring on the field. The output variable is always set to 1 in these scenarios. Figure 4.1 displays the estimated probabilities for a given static round, with the estimates derived from the most successful THGNN model.

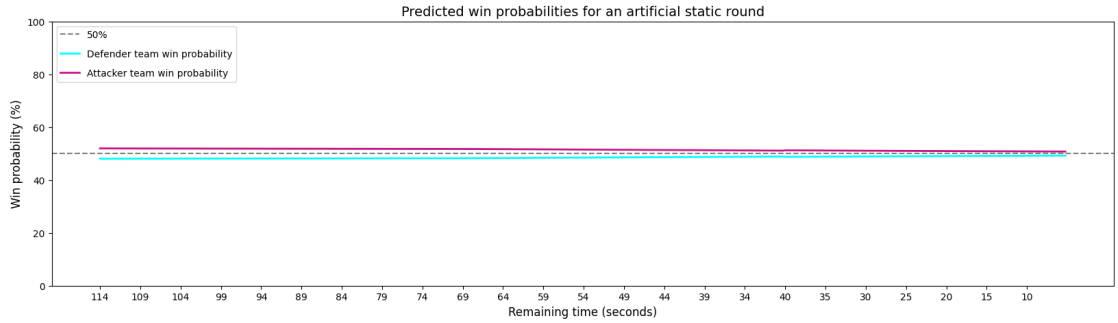


Figure 4.1: Predicted win probabilities in a static round with matching equipment values for the teams.

Several interesting phenomena can be observed from Figure 4.1. Firstly, the proposed hypothesis holds true: when no actions occur on the field by the players, the winning probabilities do not change significantly, fluctuations are not present. The minimal variation is likely attributed to the time factor: the less time remaining, the better the chances of the defenders winning the round. Another noteworthy observation is that the win probability consistently hovers around 50%. This is understandable at the beginning of the round, as both teams are fully equipped with equal chances of winning, which the model recognizes. However, as the round progresses and the time factor becomes more and more crucial for the attackers, the win probability does not rise significantly in favor of the Counter-Terrorists. The explanation for this is that the model did not encounter any data during training where no actions occurred in a round, preventing it from learning this aspect of the game dynamics. This raises an intriguing question: how would supplementing the training data with static rounds during training impact the model’s performance. This subject is recognized as an area for future research.

Through the conducted examination, it has been established that fluctuations in win probabilities are always tied to specific events and player actions. This insight shifts the focus to the next phase: deciphering the underlying factors that drive these changes.

4.3 Interpreting probability changes

In order to investigate the actions linked to the observed fluctuations in probabilities, a slightly different method from conventional model explainability is necessary. This analysis aims not to uncover the rationale behind the model’s probability predictions but to understand and interpret how the resulting changes correlate with the actions performed by the players. Such an approach provides a clearer understanding of the relationship between player behavior and the dynamics reflected in the model’s outputs. This aligns intuitively with the objective of the framework, as professional analysts similarly look for valuable and failed player decisions in practice.

4.3.1 Events dataset creation

The interpretation of win chance changes in a selected round began with the preparation of an events dataset. Since it was established that every fluctuation is linked to specific player actions, the initial approach was to create a tabular dataset for the round containing the state-changes between snapshots. A process was developed that takes all heterogeneous graphs associated with the examined round as input and transforms them back into tabular format. The output is a vector that represents the graph. This vector includes only the state-describing features related to the players and map nodes. For instance, the player’s coordinates, speed, direction of view, active weapon, health, etc., as well as features of the map node indicating whether it is currently burning or smoked, are included, while the players’ yearly statistics are not, as they do not describe the current state of the player. Additionally, boolean flags were incorporated into the graph vector to indicate which nodes the players are linked to (with a separate boolean feature for each node, where 1 indicates a connection to that node and 0 indicates otherwise). Alongside this information, the vector also included the estimated probability for the snapshot, which served as the output variable. Processing all heterogeneous graphs this way yielded a tabular state-describing dataset of the examined round.

To obtain the event dataset, which reflects changes between snapshots, the differences between the rows of this table must be calculated. Therefore, if the original dataset consists of s snapshots, the event dataset will consist of $s - 1$ snapshots, where each row will contain the changes between two sequential snapshots of the round. Following this modification, the output variable column will no longer represent the probabilities estimated by the GNN but rather their changes between the snapshots. The resulting dataset facilitates the examination of the relationship between player state changes and fluctuations in win probabilities.

4.3.2 Explanation model and Shapley values

When interpreting AI models, it is evident that the best explainer of a model is the model itself. However, Lundberg et al. [2017] stated that with deep neural networks, it is inevitable to utilize a simpler model to interpret the predictions for one or several selected inputs. This simpler model is commonly referred to as the explanation model, which aims to approximate the behavior of the original model around the chosen samples. This approach is also utilized in this paper to understand the fluctuations in win probabilities estimated by the GNN.

The objective is to train an explanation model for the event dataset of the examined round that is able to accurately predict the output variable, and to use Shapley values to

measure the contribution of each state change to this change in win probability. While this approach aligns with the one described in the mentioned article, it differs slightly as the explanation model is not intended to replicate the original model’s behavior exactly but rather to accurately mimic how it shifts the predicted values.

The explanation model needed to be chosen such that it could learn from a dataset with a large number of features (over 1000) and be explainable through Shapley values. The decision was made to use eXtreme Gradient Boost (XGBoost) regression. Although it is a boosting model that cannot be classified as a true white-box model, the high number of features necessitated moving beyond simple linear models.

To explain a round, an XGBoost model is trained and intentionally overfitted to replicate the behavior of the original GNN as closely as possible, thereby approximating the local patterns of the round’s snapshot samples with the highest accuracy. This model then allows for the examination of each state-change’s contribution to the probability changes for each snapshot. This process utilizes Shapley values. Shapley values, derived from cooperative game theory, quantify the contribution of each feature to the prediction, ensuring a fair distribution of the overall prediction among the contributing features. To implement the analysis, the SHAP Python package was utilized.

4.3.3 Explaining temporal patterns

To draw conclusions from the temporal patterns learned by the THGNN, it was necessary to make enhancements to the current explaining architecture. The previously described solution provides explanations for events occurring between sequential snapshots. However, it is evident that if we calculate differences not for sequential snapshots but for snapshots with more delay between them, this architecture will explain changes that occurred over a longer duration.

To incorporate the temporal patterns into the explanation process, six event datasets were created. One dataset contains changes compared to the previous snapshot, while the others measure changes against states from 1, 2, 3, 4, and 5 seconds prior. The choice of these five additional datasets stems from the fact that each dynamic graph in the dataset consists of 20 snapshots, corresponding to a total duration of 5 seconds. By training XGBoost models on all six generated datasets and explaining the predictions using Shapley values, the effects of previously occurring events can be averaged into the explanations for the same snapshots. This allows for a more comprehensive understanding of the temporal dynamics influencing model predictions.

4.3.4 Example explanations

With the explanation architecture prepared for deployment, the changes in model predictions were examined across several rounds. In this paper, the focus will be on the predictions visualized for the round shown in Figure 3.2. The interpretations provided align well with intuition; for example, when eliminations occur, the Shapley values accurately indicate that the most significant features responsible for the probability changes are Player *A*’s death or Player *B*’s increased eliminations. Such instances can be observed in the round at the 40 or 21-second marks. A.2 contains the shap values visualized for the latter example. Furthermore, the model also effectively identifies bomb-related events. As seen in Figure 3.2, around the 34-second mark, the bomb starts getting planted, and three seconds later the plant turned out successful, leading to two significant probabil-

ity increases for the attacking side during this period. The Shapley values also correctly reflect the importance of these events.

Additionally, the explanation model successfully highlights changes related to grenade usage. For instance, there is a slight increase in the defenders’ winning chances around the 77-second mark, which persists for a noticeable time. Upon examining this change, the model highlighted that it was due to a defender’s grenade usage. Reviewing the match replay footage, it was observed that the highlighted player threw an incendiary grenade just seconds before the increase, as well as a flashbang, effectively pushing the attackers back and regaining crucial map control. This causal relationship is validated by the explanations, which indicated that the most critical features included two attackers getting blinded and a map node setting on fire.

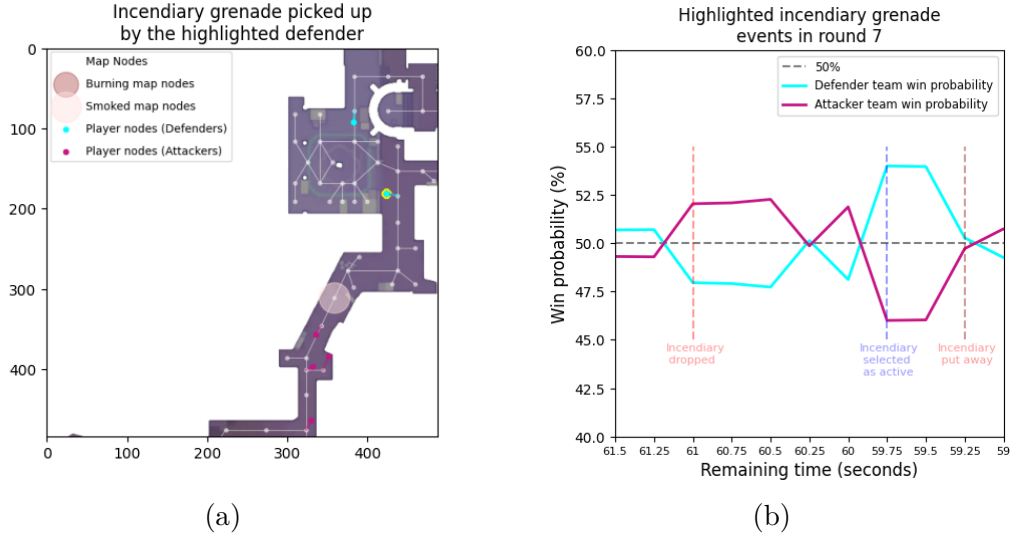


Figure 4.2: (a) The situation at the moment the incendiary grenade was picked up by the highlighted player. (b) The winning chances of the teams at the time of the examined incendiary explanations.

One of the most intriguing events of this round regarding explanations occurs around the one-minute mark. At this point, the Counter-Terrorists first experience a decrease, then a noticeable increase in their winning chances for two seconds. However, after comparing this to the video replay of the match, the reason for these changes not evident. Analyzing it with the explanation model, it highlights the significant events associated with the changes. At the time of the decrease (at the 61 seconds mark), the second defender highlighted on Figure 4.2 (the defender without yellow highlighting) drops an incendiary grenade to the ground. Intuitively, this action causes a decrease as the other defender (highlighted in yellow) has to step back to retrieve it. However, once the grenade is in his inventory, he selects it as his active weapon, resulting in a +5% increase in the team’s winning probability. Although this boost only lasts until it stays his active weapon (around the 59-second mark), it is important to note that there are four attackers grouping in the lower part of the image, a position referred to as banana. Should they decide to push forward to the bombsite, a well-timed incendiary grenade from the defenders could potentially halt and delay the push, providing the CTs with enough time to prepare. Interestingly, the push does not occur, yet both the model and the explanation method positively valued the incendiary grenade being held as the active weapon, highlighting the potential impact of it. Visualizations in A.1 highlight the of the SHAP values corresponding to these events and their significance. Upon analyzing other rounds from different matches, several similar,

seemingly inconsequential events were highlighted by the GNN and explanation model, often exceeding expectations.

While the explanations provided are mostly consistent and reliable, they are not entirely free from errors. For example, in the visualized round, the spike in winning chances at around 88 seconds does not correspond to any significant event in the game. The reason the THGNN might have recognized this as important could simply be attributed to data hunger. GNN models often experience data hunger, especially in this instance, particularly given the limited data size of 90,000 dynamic graphs.

4.4 Evaluating player actions

With the reliable explanation process in place, evaluating player actions has become achievable in multiple ways. One of the most intuitive methods is to identify significant events within a round according to the model and assess which players caused these events. Using the explanatory component, these contributions can also be quantified, allowing for a clear tracking of which players' actions lead to significant changes in outcomes.

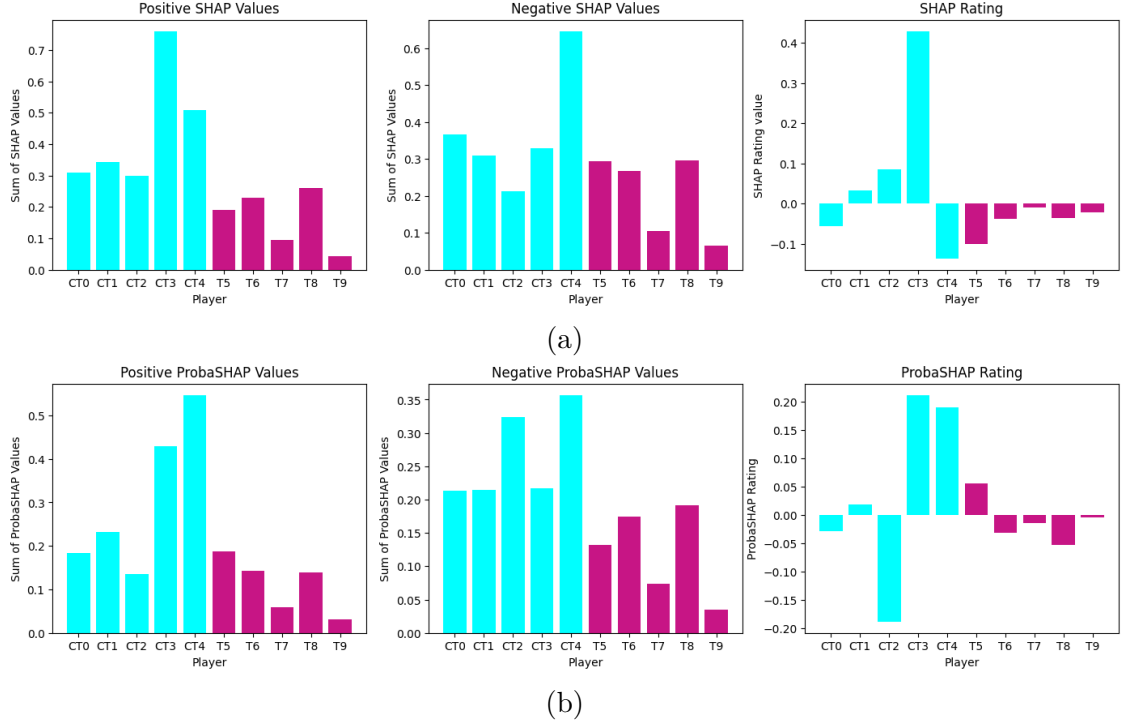


Figure 4.3: (a) SHAP Ratings of the players in the examined round. Defenders are plotted with blue, attackers with purple. (b) The refined ProbaSHAP Rating of the players in the same round.

Moreover, this framework can also be employed to generate new statistics. In this esports, there is a statistic known as impact aimed to measure the extent to which a player can influence the opposing team. This includes metrics such as opening kills, damage inflicted, and double or even triple kills. The proposed model offers a novel approach to measuring this impact. The framework allows for an analysis of each player's state changes frame by frame, assessing their effects on win probability using Shapley values. These quantities can be aggregated per second, providing insight into the magnitude and direction of each player's influence on win probability expressed through Shapley values. If this aggregation

is performed not just per second but across the entire round, a new impact-like rating, a SHAP Rating can be derived, evaluating the ratio of a player's positive to negative actions within the analyzed round. Figure 4.3(a) displays the SHAP ratings for players in the previously analyzed round.

The SHAP Rating can be further refined by incorporating the probability changes into the calculations, normalizing the variations in Shapley values. For instance, if a player's action has a high Shapley value for a minor probability change, it may not be as significant as a smaller contribution to a larger change. Therefore, normalizing Shapley values among players within the analyzed snapshot and multiplying them by the win probability change results in a new, more sophisticated statistic labeled as the ProbaSHAP Rating. Figure 4.3(b) illustrates these ratings for the previously analyzed round.

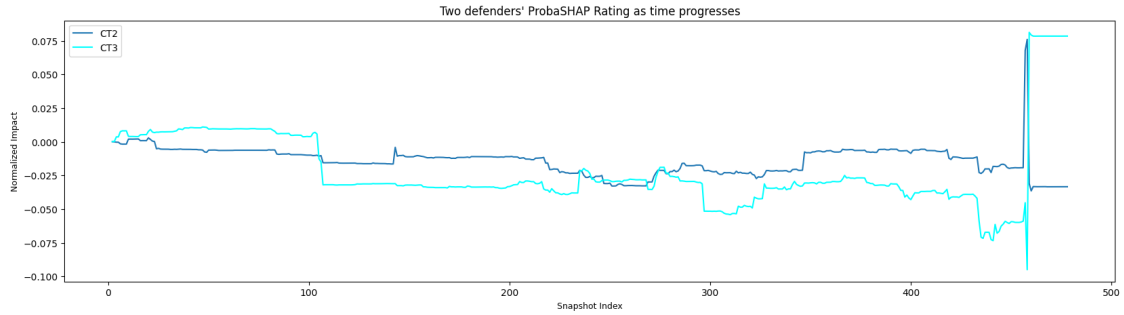


Figure 4.4: ProbaSHAP Rating evolution of two defenders visualized as time progresses.

Another interesting analytical opportunity provided by the framework is the examination of players' ratings evolve over time. Figure Z visualizes the ProbaSHAP Ratings of two defenders throughout the round. The negative values for both defenders in the figure arise from the fact that they were positioned on the opposite side of the map at the A bomb site during the first half of the round, while the attackers successfully secured control of the B bomb site. The increase in both of their ratings begins when they start approaching the B bomb site to retake it with their teammates. Notably, towards the end of the round, the defender labeled as CT3 (the player's fictive name is *sjuush*) experiences a significant decrease, then an increase in his rating. This is due to successfully eliminating the last remaining attacker, thereby securing the area for his teammates to defuse the bomb. The decrease prior to that is due to almost getting eliminated by the attacker just before winning the fight. In contrast, his teammate, *device*, denoted as CT2, exhibits a spike in his ProbaSHAP rating. This observation occurs because the Counter-Terrorists are in a 3v1 situation, nearly retaking the bomb site, and at this moment, *device* begins the bomb defusal process, resulting in the increase. However, shortly after, his rating declines as he stops defusing to assist in the fight against the last remaining enemy. This negative evaluation of his rating arises from the fact that he had two teammates available to cover him during the defusal and there was limited time remaining to accomplish this task. Nonetheless, after eliminating the last attacker, they were able to defuse the bomb, although this does not impact the ratings, since at that point, the round victory is secured.

Chapter 5

Summary

Counter-Strike is an increasingly popular esports, yet there is a notable lack of practical AI solutions for evaluating player or team performance. This gap in the field motivated the present research. The primary objective was to develop a novel system for assessing player actions using Graph Neural Networks (GNNs), capable of predicting team win probabilities based on spatial and temporal patterns in gameplay. By linking these probabilities to in-game events, the goal was to trace fluctuations in winning chances back to specific player actions, thereby providing a meaningful evaluation of individual and team performance.

The proposed approach employs advanced HGNN and THGNN architectures on match replayed data transformed into heterogeneous graphs. Achieving over 76% accuracy in predicting round-winning teams on the map Inferno, the models serve as the state-of-the-art solutions for this task. Additionally, a matrix optimization technique is proposed that can be applied generally to learn from heterogeneous dynamic graphs in a mini-batch manner. This approach improves compute time and efficiency by orders of magnitude. Beyond predictive accuracy, the framework provides interpretability through proposed SHAP and ProbaSHAP player ratings, offering insights into how specific actions impact win probabilities. This methodology enables in-depth player performance analysis, with a focus on temporal changes, thus presenting a comprehensive tool for understanding strategic dynamics in professional Counter-Strike.

This paper acknowledges several limitations that impact the overall findings and conclusions. Firstly, the dataset used is relatively small in comparison to the complexity of the problem and the GNN architecture, indicating a potential for improved results with a more extensive dataset. Additionally, there are occasional instances of inaccurate probability estimates produced by the models, which can affect the reliability of the predictions.

Bibliography

- [1] Esports Charts. Popular esports games in 2024 by viewership. <https://escharts.com/top-games>, 2024, Accessed on May 28, 2024.
- [2] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.
- [3] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL <https://arxiv.org/abs/2001.08361>.
- [4] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017. URL <https://arxiv.org/abs/1609.02907>.
- [5] Marc Leroux-Parra. Esports part 1: What are esports? <https://hir.harvard.edu/esports-part-1-what-are-esports/>, 2020, Accessed on May 28, 2024.
- [6] Jingzhe Liu, Haitao Mao, Zhikai Chen, Tong Zhao, Neil Shah, and Jiliang Tang. Neural scaling laws on graphs, 2024. URL <https://arxiv.org/abs/2402.02054>.
- [7] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.
- [8] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions, 2017. URL <https://arxiv.org/abs/1705.07874>.
- [9] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018. URL <https://arxiv.org/abs/1710.10903>.
- [10] Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer Nature, 2022.
- [11] Peter Xenopoulos, Harish Doraiswamy, and Claudio Silva. Valuing player actions in counter-strike: Global offensive. In *Proceedings of the IEEE International Conference on Big Data*, 2020.

Appendix

A.1 Feature descriptions

In the dataset, player columns adhere to a structured naming convention: **player_ID_FEATURE**. Here, *ID* denotes the player’s identification number $[0 - 9]$, while *FEATURE* describes the statistic contained in the column. For instance, *player_0_x* holds the X coordinate of the first player, and *player_1_x* contains the X coordinate of the second player, and so forth. In the subsequent lists, the *FEATURE* elements of the naming convention are described

Player-related features:

- x, y, z: Coordinates of the player.
- hp: Healt remaining. $[100-0]$
- armor: Armor remaining. $[100-0]$
- flashGrenades: Number of Flash Grenades of the player. $[0-2]$
- smokeGrenades: Number of Smoke Grenades of the player. $[0-1]$
- heGrenades: Number of High Explosive Grenades of the player. $[0-1]$
- totalUtility: A player’s total number of grenades.
- isAlive: Whether the player is alive in the actual secon of the round.
- isReloading: Whether the player is reloading his weapon.
- isBlinded: Whether the player is blinded by a flash grenade.
- isDucking: Whether the player is crouching.
- isDefusing: Whether the player is defusing the bomb (only important for the defenders).
- isPlanting: Whether the player is planting the bomb (only important for the attackers).
- isUnknown: Whether the player’s position is unknown to the enemy team.
- isScoped: Whether the player is scoped (when using a long range sniper rifle).
- equipment_value: How expensive are the weapons and utility at the player are.

- `equipmentValueRoundStart`: How expensive are the weapons and utility at the player were when the round begun.
- `has_helmet`: Whether the player has a helmet (it is an extension to the armor: more expensive but gives additional protection to players)
- `has_defuse`: Whether the player has a defuse kit (only the CTs can have it, it is a tool for speedig up the defusing of the bomb, if it was planted)
- `hasBomb`: Whether the player has the bomb (only relevant for attackers).
- `stat_kills`: Number of successful eliminations done by the player in the examined match.
- `stat_HSK`: Number of headshot-kills.
- `stat_openKills`: Number of opening kills (first kill in the round).
- `stat_tradeKills`: Number of trade kills (a trade is when a player A gets eliminated by X, but then A's teammate, C, eliminates X shortly after).
- `stat_deaths`: Number of deaths of the player in the examined match.
- `stat_openDeaths`: Number of opening deaths (when the player gets eliminated first from the round).
- `stat_assists`: Number of assists (when the player assists in a kill by causing damage to the opponent, but the finishing strike is struck by a teammate).
- `stat_flashAssists`: Number of times the player successfully blinded an opponent, resulting in a teammate killing that opponent.
- `stat_damage`: Total damage done by the player.
- `stat_weaponDamage`: Total damage done by the player by weapons.
- `stat_nadeDamage`: Total damage done by the player by nades.
- `activeWeapon_[weapon_name]`: Boolean flags that indicate which weapon the player is currently holding in his hands (some examples for `weapon_name`: AK -47, AWP, Smoke_Grenade, etc.).
- `is_ct`: Whether the player is playing on the defending side.

Game-describing features:

- `roundNum`: Shows in which round the snapshot was taken in
- `CT_aliveNum`: Number of players alive on the CT side.
- `T_aliveNum`: Number of players alive on the T side.
- `match_id`: Match Identifier.
- `t_score`: How many rounds the attackers (Ts) won so far in the match
- `ct_score`: How many rounds the defenders (CTs) won so far in the match

- `is_team1_ct`: Whether the first team is playing on the CT side currently
- `CT_equipmentValue`: Total value of equipment on the CT side.
- `T_equipmentValue`: Total value of equipment on the T side.
- `team1_total_grenade_num`: Number of grenades of the first team. (Removed)
- `team2_total_grenade_num`: Number of grenades of the second team. (Removed)
- `is_bomb_being_planted`: Boolean flag, whether the bomb is being planted at the moment.
- `is_bomb_being_defused`: Boolean flag, whether the bomb is being defused at the moment.
- `is_bomb_defused`: Boolean flag, whether the bomb is defused.
- `is_bomb_planted_at_A_site`: Boolean flag, whether the bomb is planted at A site.
- `is_bomb_planted_at_B_site`: Boolean flag, whether the bomb is planted at B site.
- `bomb_X`: Bomb X coordinate.
- `bomb_Y`: Bomb Y coordinate.
- `bomb_Z`: Bomb Z coordinate.
- `time_remaining`: Remaining time from the round.
- `numerical_match_id`: Four character long numerical match id.
- `bomb_mx_pos[1 – 9],:` Both bombsites were divided into 9 smaller fractions (a 3x3 matrix). This indicates which area the bomb is planted in (if planted).
- `CT_winsRound`: Output variable, shows whether the defenders win the round in the end.

A.2 SHAP Explanations

This paragraph provides examples of using Shapley values to interpret the model's decision-making. The SHAP Python package's force plot illustrates each feature's contribution to changes in winning probability, with red indicating a positive (win-increasing) Shapley value and blue indicating a negative impact. Only the most influential features are shown, with the values representing changes between consecutive snapshots in the normalized data. Figure A.2.1 examines the elimination at the 21-second mark in the previously analyzed round, offering a clear view of how specific in-game actions influence win probability.

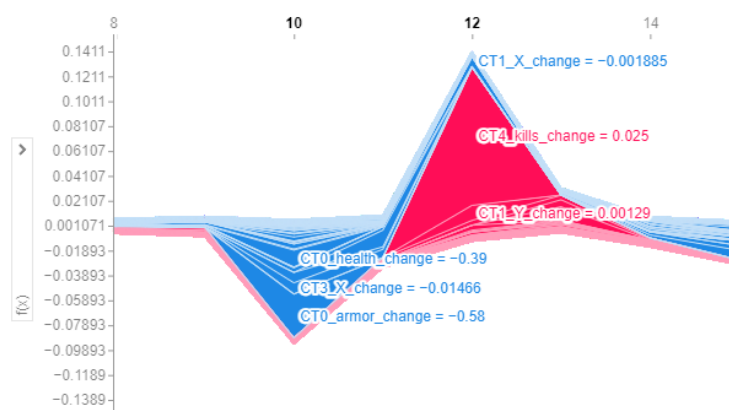


Figure A.2.1: An elimination event examined using Shapley values. The x axis represent the time stamps, while the y values show the probability change between two snapshots normalized between 0 and 1.

The figure shows that the primary factor driving the 14% boost in win probability is the increase in eliminations by the player labeled CT4. Notably, just two snapshots earlier (0.5 seconds prior), the defender labeled CT0 experienced a substantial loss in health and armor, which in turn lowered the CTs' win chances.

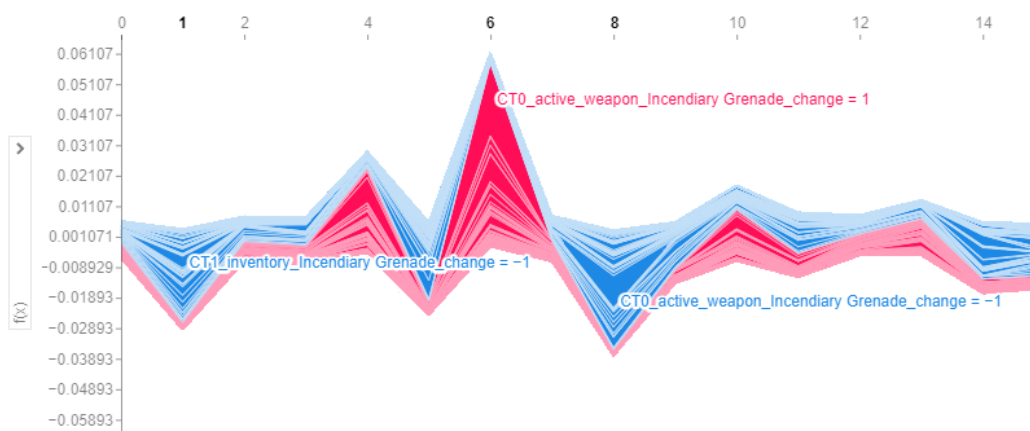


Figure A.2.2: The SHAP values of the described events with the incendiary grenade exchange in the examined round.

Another example analyzed involves the events with the incendiary grenade described in Chapter 4.3.4. Figure A.2.2 illustrates the factors behind the probability fluctuations shown in Figure 4.2(b). Initially, the main factor in the first probability decrease in the first timestamp of the plot, is the player labeled CT1 dropping the incendiary grenade (signaled by a change in the indicator flag from 1, indicating possession, to 0). Subsequently, when CT0 picks it up as an active weapon in the sixth snapshot, the win probability increases, as his position is favorable for using the grenade in the event of an attacker push. However, when CT0 later switches to another active weapon, the model responds by reducing the win probability by around 4%.