POLITECNICO DI TORINO

Master's Degree in Department of Control and Computer Engineering

Master's Degree Thesis

AI-Assisted Optimization of Cloud Gaming Experience

Supervisors
Prof. Paolo Giaccone
Prof. Andrea Bianco
Dr. German Sviridov

Candidate
Bahadir Basaran

Academic Year 2020-2021

Abstract

Cloud Gaming technology is gaining immense popularity day by day. While the idea of not paying any more for pocket-burning hardware sounds good at first, the question immediately arises if we can at least get the same gameplay quality. The extensive source-demanding nature of Cloud Gaming technology requires more serious Quality of Experience (QoE) assessments on video games to preserve player satisfaction.

Today, games are way more sophisticated than ever before: they are mostly composed of various game stages such as environment exploration, action-combat, dialogue with Non-player Characters (NPC). This thesis study follows the idea that there may be a correlation between the stages of a game being streamed to a player and the stream's bitrate, and that game stages can be classified by exploiting this correlation. In this way, e.g., Cloud Gaming service providers can allocate their resources among their players according to the instant game stages of the players, and there would be many more in-game QoE optimization possibilities.

The result of this thesis study, in which a Deep Reinforcement Learning (DRL) agent is used instead of the human factor that causes the QoE assessments to be cumbersome, indicates that it is possible to classify instant game stages with high accuracy by processing the corresponding instant bitrate values.

Contents

Chapter 1

Introduction

1.1 Problem Description and the Motivation

The world of video gaming has witnessed many cool names for innovative technologies. One such innovative technology that is getting enormous attention from day to day is Cloud Gaming.

Cloud Gaming is a technology that allows players to use any device to play the most demanding games through a broadband Internet connection, without the need to store any game data on their devices. While this technology has many advantages, such as saving players from constantly spending money on hardware and from storing many game data into their personal computers, it comes with limitations due to massive data transmission over the network. Those limitations, e.g., latency, can have a disastrous effect on gameplay, especially on games with a high level of dynamism, such as First Person Shooter (FPS) games. As a result, Quality of Experience (QoE) assessments play a crucial role in tackling such limitations.

According to Qualinet (2013) [1], QoE measures the level of enjoyment or frustration felt by a user while using a service or an application. It is the satisfaction of the user's expectations regarding the usability, functionality, and efficiency of the application or service being used, according to the user's personality and present condition.

Involving human players in video games' QoE assessments is quite inefficient, with many limitations regarding cost, variability across players, and lack of transferability across games. Therefore, this thesis study follows the idea of using pretrained Deep Reinforcement Learning (DRL) agents -instead of human involvement- to automate QoE assessments for more efficient outcomes.

Today's video games combine various game stages such as exploration, action,

dialogue in an immensely dynamic way. Particularly in competitive online games, e.g., FPS genre, players' tolerance to imbalanced network conditions can vary according to their instant game stages. As discussed in Section Limitations, particular game stages (e.g., action) are pretty fragile to imbalance, while others (e.g., dialogue with Non-player Characters) may have a high tolerance to imbalance.

This thesis study investigates if there is a correlation between the stages of a game being streamed to a player and the stream's bitrate, and proposes the idea of taking automated in-game QoE assessments one step further by creating a DRL agent capable of classifying game stages based on the instant bitrate values of the stream. In this way, besides many in-game QoE optimization possibilities, Cloud Gaming service providers can discriminate players' instant game stages and allocate high resources to those currently in an, e.g., delay-critic stage while allocating relatively fewer resources to those currently in an, e.g., delay-tolerant stage.

1.2 Outline of Design and Implementation

The first section of Chapter 3 discusses the game selection of the thesis study and the reasons behind the selection. In the following sections, the research platform used as an Application Programming Interface (API) to the selected game and the DRL agent trained on the game are explained. The training methodology and the architecture of the agent are also illustrated.

Once having discussed the infrastructure, how the agent is exploited to generate gameplay dataset, and how the generated dataset is processed for dataset analysis are explained.

Section Deep Learning Architecture and Frameworks Used in the Study first gives an insight about Neural Networks, together with the specific type of network architecture used in this thesis study, and highlights the Machine Learning framework used to train the model afterward. The model creation process is explained with detailed diagrams in this section.

The training process of the model is analyzed with all components in detail. In the end, the model is tested with data generated on different game maps and different configurations.

Chapter 2

Cloud Gaming

2.1 Cloud Gaming: Future of Gaming Industry

Cloud Gaming, sometimes called *Gaming-as-a-Service*, is a game-changing technology that allows players to access their favorite games straight from the cloud, rather than having topnotch gaming computers or consoles such as PlayStation, Xbox.

In today's technology, players are obligated to renew their hardware frequently to play the latest games, and those games often require the utmost expensive hardware. GPUs (Graphics Processing Unit) might be considered the most critical of all. The number of people interested in Crypto Mining is increasing day by day, and this causes a GPU shortage, hence transformation to the black market.

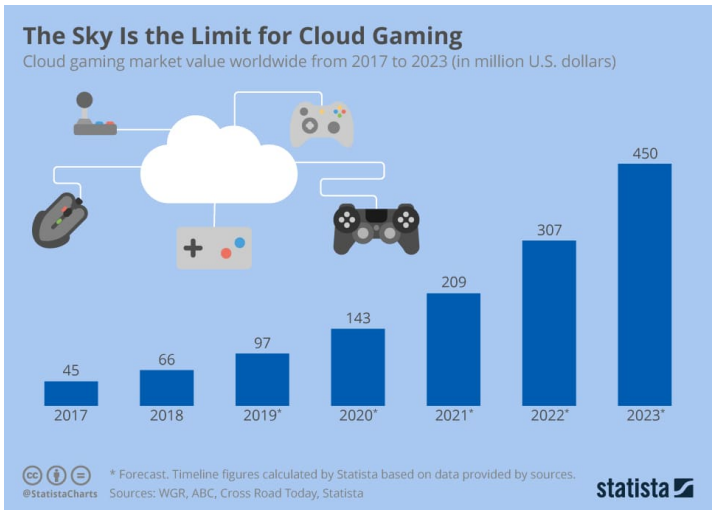


Figure 2.1: Cloud Gaming worldwide market value, reproduced from [2]

The worldwide market value from 2013 to 2017 depicted in Figure 2.1 indicates that Cloud Gaming Industry currently has a market value of over 200 million U.S. Dollars, and according to the estimations, it will be approximately half a billion U.S. Dollars in the next five years.

The difference observed in competitive -especially in First Person Shooter genre- games between players with state-of-the-art equipment and players with old technology hardware cannot be ignored. However, the advantages of the technology are not limited to getting rid of expensive hardware. Without dealing with time and space-consuming updates, instant access to games is another point since service providers have already dealt with them. Besides, since cheating is attempted on the client-side, Cloud Gaming makes cheating almost impossible because games run on the server-side.

Considering all the facts, giving players the freedom to access their games instantly with any device by eliminating the dependence on hardware will form the gaming industry's future. Cloud gaming would not only alter players' gaming experience, but it might also turn pocket-burning gaming computers and consoles antiquated.

2.1.1 Pioneer Services

Although a flawless service is yet to come because of the limitations mentioned in Section Limitations, it is evident that the significant players are sparing significant sources to dominate this field.

No matter how ironic it looks, NVIDIA GeForce Now is considered the best Cloud Gaming service at the moment. Although NVIDIA is the most prominent GPU manufacturer and hence Cloud Gaming technology is a potential threat for them, they have invested big in this technology rather than staying against it. Players can play games in high resolution across a wide range of devices such as mobile devices, computers, TVs. There is a free-to-use subscription tier, and players have to buy whatever they would like to play.

One of the most promising services is Google Stadia. Unlike GeForce Now, Stadia does not have a third-party game store, and it does not mandate any payment for streaming if the player already acquires the game. In addition, Stadia PRO gives access to 4K HDR streaming of the acquired games.

Amazon has been working on Luna, a channel-oriented platform that offers 4K game streaming, along with Twitch integration. However, Luna is still in the Beta Phase.

2.1.2 Cloud Gaming Under the Hood

In a nutshell, the most significant difference between the traditional gaming setup and the Cloud Gaming setup comes from where the information is processed, as seen in the diagrams below in detail.

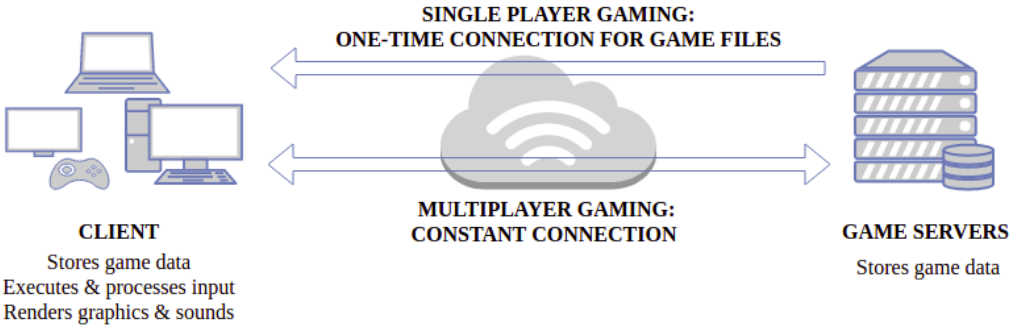


Figure 2.2: An illustration of traditional gaming setup

Figure 2.2 illustrates the fundamental setup of traditional gaming. In the traditional setup, all source-demanding processes are operated on the client-side. The client executes game commands, and the output graphics and sounds are rendered directly by the client's hardware components. Game servers are only involved for one-time game data access in single-player gaming or interaction with other players in multiplayer gaming.

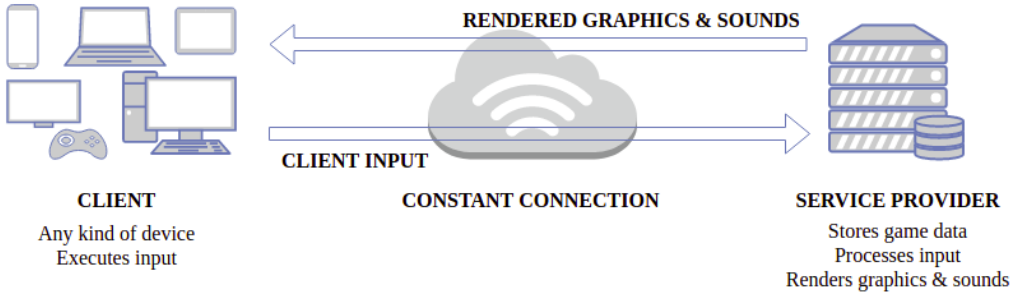


Figure 2.3: An illustration of Cloud Gaming setup

Figure 2.3 illustrates the fundamental setup of Cloud Gaming. In the Cloud Gaming setup, all source-demanding processes are operated directly on the service provider side instead of burdening the client. Besides a desktop computer, laptop, or console; the client can be any kind of device (dumb or not) with a controller: mobile phone, tablet, TV.

The client only executes game commands, and the input is transferred to the service provider's servers through the Internet connection. All the game data, required

hardware are stored directly on the service provider's servers, and the client's input is directly processed on these servers. Graphics and sound are rendered according to the client's input and transferred back.

2.1.3 Limitations

The common belief is that Cloud Gaming services are pretty sufficient to take computers or consoles' places for single-player and casual multiplayer games. In these kinds of games or game modes, an average player does not experience a significant difference between traditional setup and Cloud Gaming service. An average player can tolerate a slight latency, and even a professional player might be immune to a slight latency in single-player or casual multiplayer games. However, in competitive games such as the First Person Shooter genre, each millisecond is crucial, and many cannot tolerate latency. Likewise, while 100 milliseconds of lag cannot be noticed easily during a dialogue scene with an NPC (Non-Player Character), half of this lag might cause a catastrophe during an action-combat scene.

Cloud Gaming services have more latency than local gaming setups such as gaming PC and consoles. The reason behind this problem is quite apparent. As shown in Figure 2.3, instead of being processed directly on the client's device, each client input must traverse the network, and resulting graphics and sounds must traverse back. This loop introduces latency inevitably.

In this context, there are many significant factors to be evaluated and engineered as of now. Players' Internet connection speeds, distances to locations of Cloud Gaming services' servers are the crucial matters that come into mind first. Considering how worldwide streaming services such as Netflix are burdening to bandwidth today, the size of the problem will tremendously increase when the gaming industry invades the Internet.

2.2 Related Work

The limitations depicted in Section Limitations clearly indicate the importance of Traffic Engineering and Quality of Experience (QoE) assessments for Cloud Gaming services.

Based on the subjective QoE assessments conducted by Laghari et al. (2019) [3], the technical parameters that have an impact on Cloud Gaming QoE are specified as bitrate, frame rate, throughput, packet loss, and delay rate.

Bitrate and frame rate are the measurements of the number of bits and frames

transmitted per unit time end to end, respectively. Throughput, the number of bits transmitted over the network in a given time, is a similar term to bitrate. However, the bitrate can be assumed as a design parameter, while throughput is an observed, dynamic parameter. Higher bitrate and throughput, roughly, mean more detailed graphics, and a higher frame rate provides a smoother view for players, hence a better game experience. However, they all require more severe scaling on the network in order to preserve the QoE.

Packet loss and packet delays are important network terms referring to drop of data irrecoverably and late arrival of data between server and end-user, respectively. Both of the situations degrade the user experience dramatically. Therefore network must ensure lower loss and delay in terms of a better experience.

In the same sense, QoE assessment conducted on the famous game World of Warcraft by Slivar et al. (2015) [4] indicates that packet loss and packet delay have unignorable effects on QoE, especially packet loss with more influence than delay.

The following test scenarios are designed by Jarschel et al. (2011) [5] to assess packet loss and delay effects on QoE.

Scenario ID	Delay	Packet Loss	Direction
B	0 ms	0.0 %	both
D1	80 ms	0.0 %	both
D2	200 ms	0.0 %	both
D3	300 ms	0.0 %	both
L1	0 ms	0.3 %	both
L2	0 ms	1.0 %	both
M1	40 ms	1.5 %	both
M2	180 ms	0.3 %	both
A1	120 ms	1.0 %	client to server
A2	120 ms	1.0 %	server to client

Table 2.1: Test scenarios designed by Jarschel et al. (2011) [5]

The scenarios shown in Table 2.1 are tested by a group of casual and leisure gamers on three different kinds of games: slow-paced, medium-paced, and fast-paced. Each player is supposed to test each scenario for one minute, starting from the baseline *B*. At the end of each scenario, players are asked to evaluate his/her QoE by a scoring metric called MOS (Mean Opinion Score), from 1 to 5, with increasing quality. Figure 2.4 below indicates the obtained MOS ratings per each scenario and each type of game.

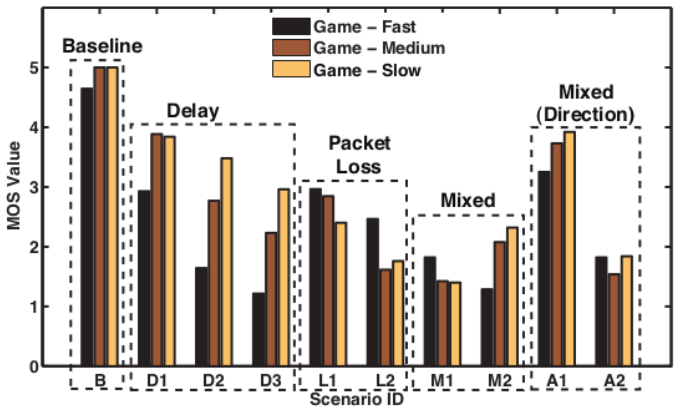


Figure 2.4: MOS ratings per each scenario and game, reproduced from [5]

Based on Figure 2.4, it is easy to deduct that the faster the game or higher the delay-loss, the lower the QoE. Another interesting deduction that supports the claim in the Section Limitations is that players experience approximately the same quality up to 80 milliseconds of delay during the slow and medium-paced games. After 80 milliseconds, the delay starts to be noticeable by players.

The assessments discussed above are subjective assessments, which means the active involvement of test groups. The last assessment described by Table 2.1 and Figure 2.4 is composed of 79 test runs, with 790 player votes. Even though a relatively small group of players carry out this experiment, it is quite time and resource-consuming. Besides, the assessment process can vary from person to person. The same person can perform very differently on different tests of the same test case or while performing very well on one type of game genre, can perform poorly on another one. These kinds of possibilities bring variability to results, damage assessment reliability, and decrease transferability across games.

An interesting idea, which also constitutes the starting point of this study, is put forward by Sviridov et al. (2020) [6]. Instead of conducting subjective tests on Cloud Gaming QoE assessments, they propose to take human players out of the assessment process by replacing them with Deep Reinforcement Learning (DRL) agents to automatize this process.

This study is built on the idea that in-game score and player satisfaction are positively correlated, so is QoE. To nurture this idea, a set of well-known Atari games are selected as a testbed, and DRL agents to be replaced with human players are trained on those games. After the training process, the agents are ready to play the games in different test scenarios. The test scenarios being applied are poor network conditions such as fixed latency, random latency, random keystroke drop due to the nature of the study, and the obtained results indicate that DRL agents

can efficiently and successfully conduct QoE assessments instead of using human players. Today's games often contain multiple game stages such as exploration, combat, dialogue, and one of the limitations of this work is that it is based on games that required uniform gameplay. As suggested in the article, better QoE assessments can be achieved if an agent can classify game stages in multistage games. This idea forms the basis of this thesis study.

Chapter 3

Design and Implementation

3.1 Game Selection

Doom (1993) [7] is a widely-known old-school First Person Shooter (FPS) game for MS-DOS that was released in 1993 by id Software. Players take on the role of *Doomguy*, a space mariner whose duty is to fight hordes of invading demons from hell.

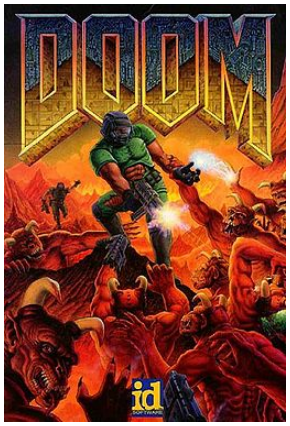


Figure 3.1: Doom Cover, reproduced from [7]

Doom is regarded as one of the most influential titles in video game history and is usually recognized as one of the best games of all time. Along with its predecessor Wolfenstein 3D (1992) [8], it helped define the FPS genre and spawned a slew of copycat games known as Doom clones. In addition, it was a pioneer in online distribution and technology such as 3D graphics, multiplayer gaming, and custom modification support through packed WAD files.

The player plays *Doomguy* through a sequence of levels set in army bases on Mars' moons and hell. In order to complete a level, the player must navigate through the area until they reach a specified exit chamber. The levels are organized into designated episodes, with the last level focused on a boss battle against a challenging opponent. While the environment is rendered in 3D, the opponents and objects are rendered as 2D sprites from various predefined angles of view, a practice called 2.5D graphics and scientifically as ray casting. Levels are frequently labyrinthine, and a full-screen automap depicts the locations explored up to that moment. The player must combat various attackers, such as devils and zombies, while maintaining ammo, health, and armor supplies. Monsters frequently emerge in huge groups, and the game has five difficulty settings that amplify the amount and damage done by opponents, with enemies re-spawning after death and moving quicker than usual on the maximum difficulty setting. The monsters' behavior is relatively straightforward: they either move toward their opponent or strike by shooting fireballs, biting, and scratching.

The most important reasons for using Doom in this thesis study are that Doom is a pretty simple game that does not require high processing power and the possibility to access and intervene its game engine.

3.2 A.I. Research Platform as an API to the Game

ViZDoom (2018) [9] is an Artificial Intelligence research platform based on the famous old-school game Doom and enables the creation of Artificial Intelligence agents that play Doom by exploiting raw visual data retrieved from the Screen Buffer. It is primarily meant for experiments in Machine Learning, especially in Deep Reinforcement Learning.

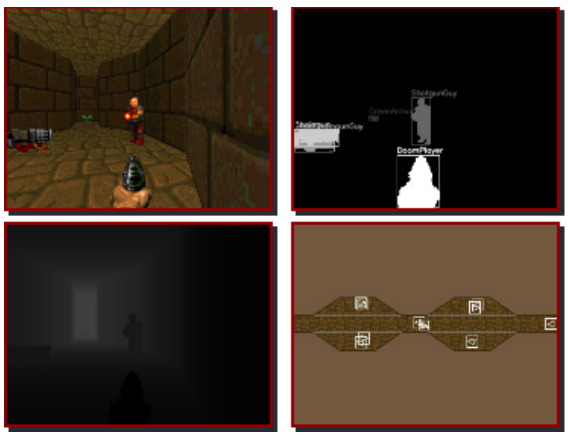


Figure 3.2: ViZDoom, reproduced from [9]

The main features of this platform are that being fast and lightweight while providing easy access to the game engine, the depth buffer, and an API for Python. Aside from the asynchronous and synchronous single-player and multiplayer modes, custom scenarios are simple to design.

3.3 Deep Reinforcement Learning Agent Trained on the Game

Arnold is a PyTorch implementation of the agent proposed in *Playing FPS Games with Deep Reinforcement Learning* (2017) [10]. Arnold was built and trained on ViZDoom and won the *ViZDoom A.I. Competition 2017* [11].

Arnold includes a collection of 17 pre-selected maps for training and evaluation, as well as five pre-trained models which can be visualized and played against, including the ones that won the *ViZDoom A.I. Competition 2017*.

3.3.1 Methodology

In partially observable environments, an agent observes only a part of the environment, which is typically insufficient to deduce the whole state. Likewise, in Doom, the agent's point of view is confined to 90 degrees centered on its position. Deep Recurrent Q-Networks are introduced by Hausknecht & Stone (2015) [12] to tackle such situations by taking an extra input provided by the network at the previous step into consideration. Arnold was built based on Deep Recurrent Q-Networks.

3.3.2 Model Architecture

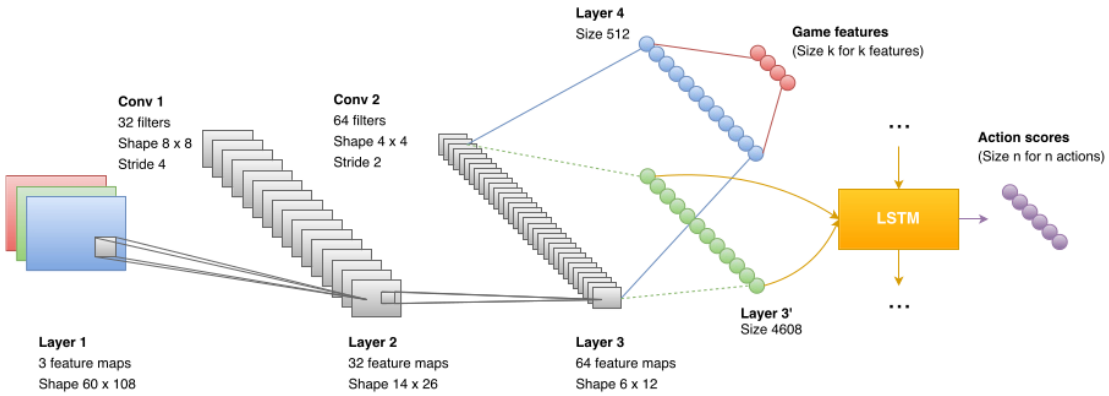


Figure 3.3: Arnold model architecture illustration, reproduced from [10]

The two convolutional layers (Conv 1 and Conv 2) at the beginning are fed with the screen buffer. Conv 1 and Conv 2 output are separated into two streams. The bottom one flattens the output (Layer 3) and transmits it to the LSTM layer. The top one transfers it to Layer 4, an additional hidden layer, and finally to a final layer representing each game element.

3.4 Gameplay Dataset Generation

Figure 3.4 below illustrates how the agent is manipulated for frame extraction and labeling.

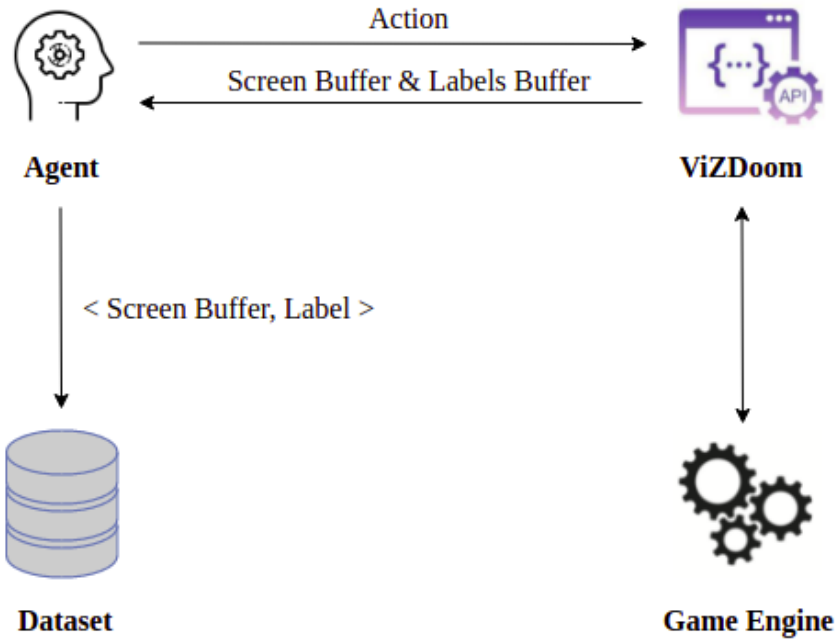


Figure 3.4: An illustration of frame extraction and labeling process

As seen clearly in the figure, the agent does not directly interact with the game engine. Instead, all interventions to the game engine should pass through ViZDoom, and ViZDoom serves here as an *Application Programming Interface (API)*.

In the agent's every interaction with the environment, ViZDoom fetches the instant vision of the agent by interacting with the game engine and sets the Screen Buffer with the vision. ViZDoom also keeps track of all objects in the agent's vision and saves the labels of existing objects into the Labels Buffer.

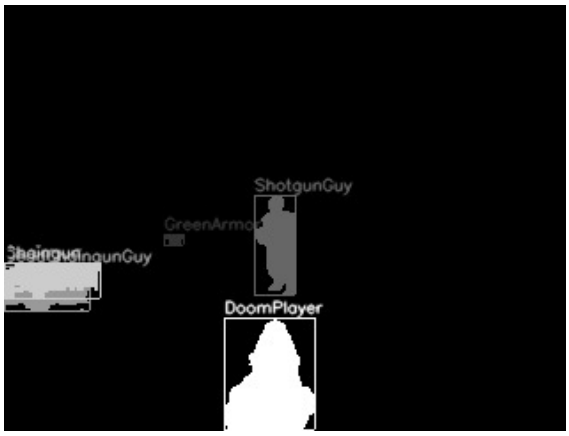


Figure 3.5: An example moment of Labels Buffer, reproduced from [13]

In this way, it is possible to detect and classify objects in the agent's vision to deduct if there is an enemy on the scene and label the scene as a combat stage or an exploration stage.

Pseudocode 1 summarizes the general mechanism behind the frame extraction and labeling.

Pseudocode 1: Frame Extraction and Labeling

```
Create dataset directory
Set the chunkSize
frames ← dict()
labels ← dict()
indexChunk ← 0
while experiment lasts do
    Make an action
    Fetch Screen Buffer through ViZDoom
    Fetch Label Buffer through ViZDoom and detect the label
    frames[indexChunk] ← ScreenBuffer
    labels[indexChunk] ← Label
    indexChunk ← indexChunk + 1
    if indexChunk = chunkSize then
        Combine frames and labels into a chunk and save
        indexChunk ← 0
        frames ← dict()
        labels ← dict()
    end if
end while
```

At the beginning of each experiment, an empty dataset directory is created.

Then, after initializing the data containers for frames and labels, the agent starts taking action. Instead of saving whole data at once, chunk-by-chunk saving is preferred for better memory utilization.

Figure 3.6 below illustrates the dataset generation process. Once having saved whole data chunks, the script Dataset Builder processes all those chunks as a sequence. It first extracts all frames and labels. Since all frames are in the form of an array, they are converted to images. FFmpeg (2006) [14] processes the sequence of images and calculates the bitrate per each timestamp. Dataset Builder receives the raw data from FFmpeg, combines frames of each timestamp with corresponding labels, and outputs the resulting dataset in CSV (comma-separated values) format.

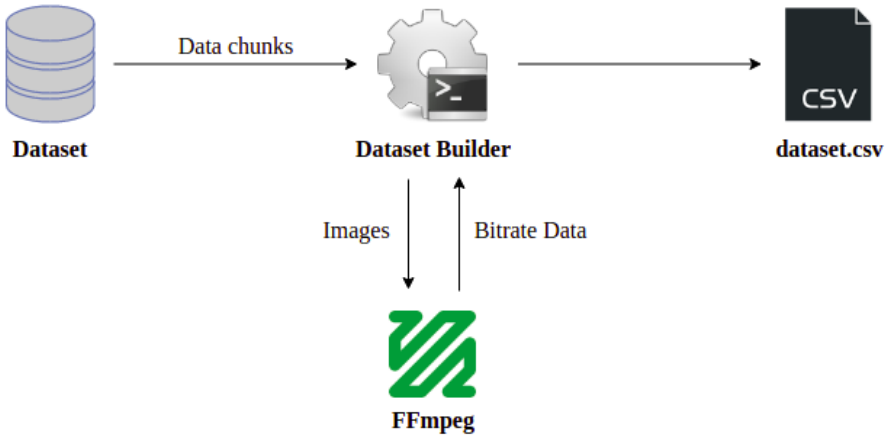


Figure 3.6: An illustration of dataset generation process

Pseudocode 2 illustrates the dataset generation process.

Pseudocode 2: Dataset Generation Process

```
labels ← list()
foreach chunk do
    foreach (frame, label) in chunk do
        image ← frame
        labels ← labels + label
    end foreach
end foreach
Run FFmpeg over images to generate dataset.csv
foreach (index, row) in dataset.csv do      ▷ row: time, frameSize, frameType
    row ← row + labels[index]               ▷ row: time, frameSize, frameType, label
end foreach
return dataset.csv
```

3.4.1 Overview of Raw Dataset

Before diving into the pre-processing step and the dataset analysis, it is necessary to discuss a raw form of a generated dataset. Table 3.1 illustrates a snippet of a raw dataset. Each row in a raw dataset is in the form of *Time, Frame Size, Frame Type, Game Stage*, respectively.

Time [s]	Frame Size [bytes]	Frame Type	Game Stage
24.800	1414	P	1
24.828	1691	P	1
24.857	1823	P	1
24.885	1807	P	1
24.914	1591	P	0
24.942	1430	P	0
24.971	5054	I	0
25.000	813	B	0
25.028	752	B	0
25.057	927	B	0
25.085	1724	P	0
25.114	989	B	0
25.142	1423	B	0
25.171	804	B	0
25.200	1893	P	0
25.228	914	B	1
25.257	934	B	1

Table 3.1: A snippet of a raw dataset

The *Time* column indicates the time in seconds where frames are captured, and the *Frame Size* column indicates the size of that particular frame in bytes.

Frame Type, also known as *Picture Type* (2020) [15], expresses the three types of frames used in video compression. Although it is not in the scope of this thesis study, it would be enlightening to summarize the characteristics of *I*, *P*, *B* frames: Among all three types, *I Frames (Intra-coded Frames)* are the least compressible frame types, yet they do not require other video frames to decode. *I Frames* can be considered as a complete image. *P Frames (Predicted Frames)*, also knows as *Delta Frames*, are more compressible than *I Frames*. They only keep the changes from the former frame to the current one. *B Frames (Bidirectional Predicted Frames)* can use both former and latter frames for maximum possible compression.

The *Game Stage* represents the instant stage that the agent is currently in, and the values *0*, *1* indicate the exploration and the combat stages, respectively.

3.4.2 Bitrate Calculation

The definition *bitrate* refers to the number of bits that can be conveyed or processed over a network in a given unit of time. In streaming, this unit is usually a second, and in this thesis study, bitrate is measured in *kilobits-per-second (kbps)*.

The frame rate of Doom is 35, which means the game runs at 35 frames-per-second. Therefore, every 35 frames are collected in this context, and their sizes are accumulated to calculate the bitrate.

$$b_t = \begin{cases} \sum_{n=t-f}^{t} s_n, & t \geq f; \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

where b_t is the bitrate at any time t, f is the frame rate, and s_t is the frame size at any time t.

Time [s]	Frame Size [bytes]	Frame Type	Bitrate [kbps]	Game Stage
24.800	1414	P	301	1
24.828	1691	P	305	1
24.857	1823	P	315	1
24.885	1807	P	321	1
24.914	1591	P	328	0
24.942	1430	P	325	0
24.971	5054	I	360	0
25.000	813	B	358	0
25.028	752	B	358	0
25.057	927	B	354	0
25.085	1724	P	363	0
25.114	989	B	364	0
25.142	1423	B	371	0
25.171	804	B	364	0
25.200	1893	P	375	0
25.228	914	B	374	1
25.257	934	B	376	1

Table 3.2: A snippet of a the raw dataset including bitrate data

3.5 Gameplay Dataset Pre-processing

The main focus of this thesis study is on bitrate and game stages. Bitrate and game stage data should be processed for consistency with real-world network scenarios.

The sole pre-processing technique discussed in this section is the Data Smoothing. Different pre-processing techniques applied to the data are discussed in the following sections before starting the training process of the Deep Learning Model.

Data Smoothing is a technique to reduce, eliminate or manipulate volatility and any noise in data. Smoothed data can highlight significant trends better and help important patterns to stand out. However, despite the apparent advantages, excessive smoothing may also cause information loss. Therefore, it is better to be aware of the trade-off and be careful about finding the sweet spot.

Smoothing is applied easily through various methods. The method used in this thesis study is the Exponential Moving Average (2004) [16]. The Exponential Moving Average, also referred to as an *Exponentially Weighted Moving Average*, is the Moving Average technique that gives more credit to the most recent data samples.

Unlike the Simple Moving Average (1920) [17], which evaluates all observed data samples with equal weight, the Exponential Moving Average treats observed data samples with variable weighting factors, which decrease exponentially over time.

The Exponential Moving Average is applied recursively to a data series Y by using Equation (3.2).

$$\begin{aligned} S_t &= \begin{cases} Y_1, & t = 1; \\ \alpha Y_t + (1-\alpha) S_{t-1}, & t > 1 \text{ and } 0 < \alpha < 1; \end{cases} \\ \alpha &= \frac{2}{w+1} \quad , \quad w \geq 1 \end{aligned} \tag{3.2}$$

where S_t is the Exponential Moving Average value at any time t, Y_t is the actual value at any time t, α is the Smoothing Factor, and w is the observation window length, time span.

In order to preserve consistency, both the bitrate and game stage data should be smoothed together. The window length w is selected 9 here (corresponds to approximately a quarter-second, since the game runs at 35 frames-per-second) to demonstrate the same dataset snippet in Table 3.2. Different values for window length are evaluated in the data analysis phase.

Pseudocode 3 illustrates the logic of data smoothing.

Pseudocode 3: Data Pre-processing: Smoothing

```
1: Set the window length w
2: α ← 2/(w + 1)
3: foreach (index, row) in dataset.csv do
4:     if index = 0 then
5:         β ← row['bitrate']                    ▷ β: last smoothed bitrate value
6:         γ ← row['stage']                      ▷ γ: last smoothed game stage value
7:     else
8:         row['bitrateSmoothed'] ← α · row['bitrate'] + (1 − α) · β
9:         row['stageSmoothed']   ← α · row['stage']   + (1 − α) · γ
10:        β ← row['bitrateSmoothed']
11:        γ ← row['stageSmoothed']
12:        if row['stageSmoothed'] > 0.5 then
13:            row['stageSmoothed'] ← 1
14:        else
15:            row['stageSmoothed'] ← 0
16:        end if
17:    end if
18: end foreach
19: return dataset.csv
```

Once smoothing having completed, the data snippet shown in Table 3.2 becomes the snippet shown in Table 3.3.

Time [s]	Frame Size [bytes]	Frame Type	Bitrate [kbps]	Game Stage
24.800	1414	P	287.794	1
24.828	1691	P	291.235	1
24.857	1823	P	295.988	1
24.885	1807	P	300.990	1
24.914	1591	P	306.392	1
24.942	1430	P	310.114	1
24.971	5054	I	320.091	1
25.000	813	B	327.672	0
25.028	752	B	333.738	0
25.057	927	B	337.790	0
25.085	1724	P	342.832	0
25.114	989	B	347.066	0
25.142	1423	B	351.852	0
25.171	804	B	354.282	0
25.200	1893	P	358.425	0
25.228	914	B	361.540	0
25.257	934	B	364.432	0

Table 3.3: A snippet of the smoothed dataset

3.6 Gameplay Dataset Analysis

Before the detailed analysis, the following bitrate-time figures generated in different map-weapon combinations indicate the bitrate changes over time according to those map-weapon combinations. Each episode below is limited to one minute for better visibility.

Each map and weapon has different characteristics, and this reflects on bitrate-time graphs. For example, while the missile and shotgun can kill opponents with a single shot, the pistol usually needs more than one shot to kill.

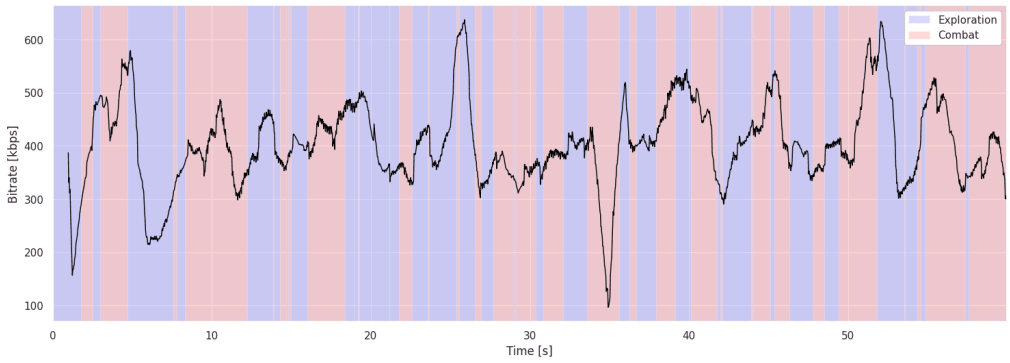


Figure 3.7: An episode recorded in Map 1 with a missile

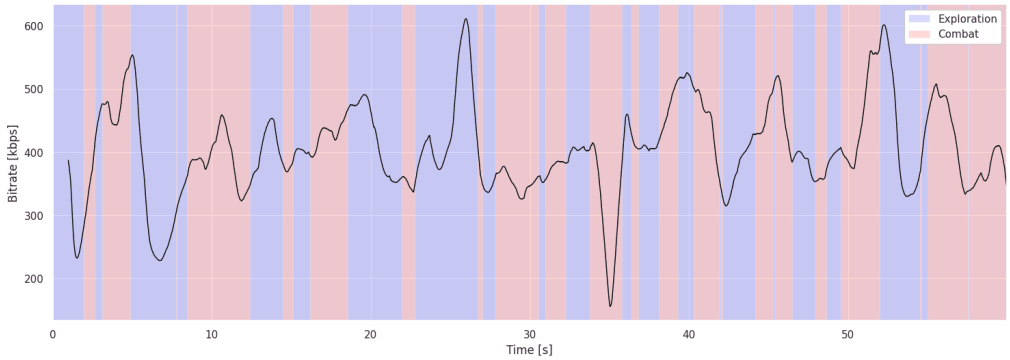


Figure 3.8: An episode recorded in Map 1 with a missile, smoothed

First of all, it is obvious to see the effect of smoothing the bitrate, considering the Figure 3.7 and Figure 3.8. The figure belonging to the smoothed data is way cleaner and noise-free. In this way, it is easier to get an insight into the trend of the data.

The background colors indicate the game stage for a particular moment. Since the smoothing operation is also applied to the game stage data for the sake of coherency, game stage values for each timestamp can also be alternated. As an example, one can see this alternation on the seconds approximately 23 - 26. While there are two combat stages on this range in Figure 3.7, those combat stages do not exist on the same range in Figure 3.8. In the same way, it is not possible to see the last exploration stage in Figure 3.7 and Figure 3.8.

The episode indicated with the following figures is recorded in Map 3, with the agent uses a pistol as a weapon.

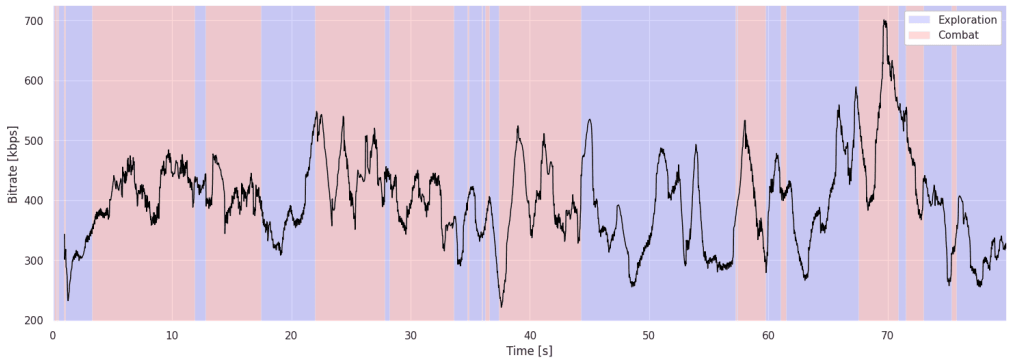


Figure 3.9: An episode recorded in Map 3 with a pistol

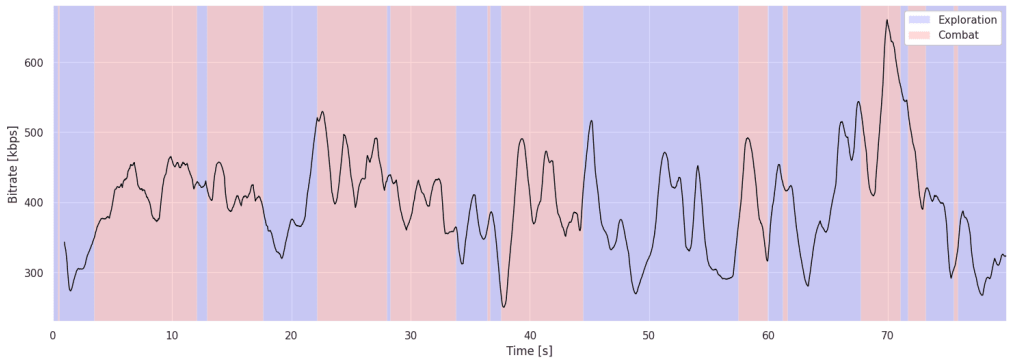


Figure 3.10: An episode recorded in Map 3 with a pistol, smoothed

The first noticeable fact in the Figures 3.9 and 3.10 is that the bitrate fluctuates in a narrower band, compared to the figures above, the episode recorded in Map 1 with a missile.

One can clearly say that there is a more prominent bitrate fluctuation compared to the figures above, the episode recorded in Map 1 with a missile. The primary reason behind these sharp fluctuations is that, as mentioned above, the agent in the first configuration uses a missile that can kill an opponent with a single shot, whereas the agent here uses a pistol that requires more shots to kill an opponent. Each pistol shot emits a fire burst, and this affects the bitrate, hence increases the level of noise.

In the same sense, since the pistol requires more shots to be fired, the required time to kill an opponent is longer than the required time in using a missile, and eventually, this results in longer combat scenes. It is easy to realize this effect by comparing the periods of combat scenes in both Figure 3.7 and Figure 3.9. It is again possible to observe the effect of smoothing here, as in the previous figures. It eliminated some noise to help for better pattern analysis.

The episode indicated in the following figures is recorded in Map 7, with the agent uses a shotgun as a weapon.

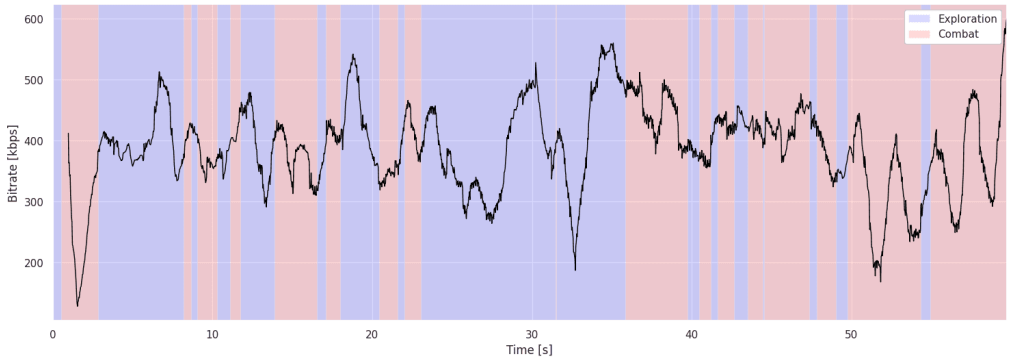


Figure 3.11: An episode recorded in Map 7 with a shotgun

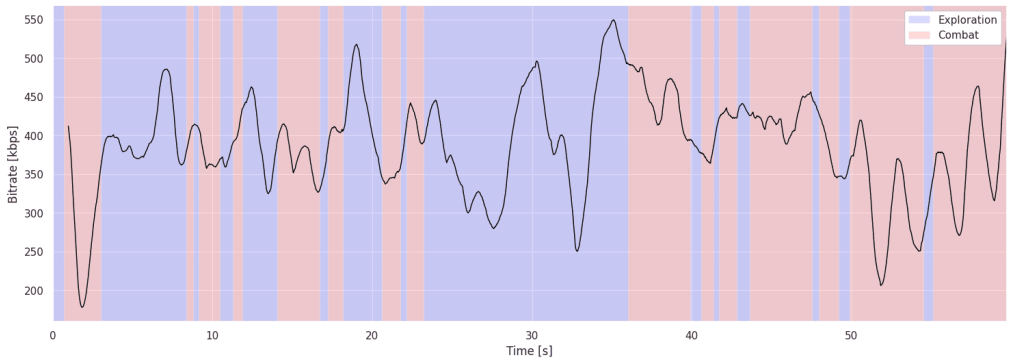


Figure 3.12: An episode recorded in Map 7 with a shotgun, smoothed

The behavior of the shotgun is similar to the behavior of the missile depicted in Figure 3.7. Both weapons have similar fire strengths, and they can kill an opponent with a single shot. The difference is the shape of the fire bursts that are emitted by the two weapons. This difference can introduce slightly different levels of noise. However, since both of the weapons can kill the opponents with a single shot, it is possible to observe the fact that the periods of combat stages are similar in Figures 3.7 and 3.11.

By considering the episodes recorded in three different maps, the existence of more sharp spikes in bitrate in Map 1 can be addressed to the structure of the corresponding map because of more dramatic scene transitions.

3.6.1 General Statistics

The rest of the analysis is based on the same map-weapon configuration for the sake of simplicity.

The episode used in these analyses is recorded on approximately 72 minutes of gameplay in Map 1 and encoded in 400 kbps. In this configuration, the agent uses a missile as a weapon, making the agent able to kill opponents with a single shot. Being able to kill the opponents with a single shot reduces the noise in the generated data since each shot brings a burst into the agent's vision.

On the other side, the agent is given *God Mode* by intervening in the game engine, which makes the agent immortal. The reason behind giving immortality to the agent is that the agent is resurrected in a random spot in the map after each death, and this causes a dramatic change in the environment instantly, hence in the bitrate.

The experiment is evaluated below in different data smoothing configurations. The window lengths used in the experiment are 9, 18, 35. Since the game runs at 35 frames-per-second, these configurations correspond to the quarter-second, half-second, and one-second time windows, respectively.

Smoothing with Quarter-second Time Window

According to Equation (3.2), Smoothing Factor α equals 0.2 for given window length 9. Smoothing the dataset in this configuration produces the following statistics throughout the episode.

Game Stage	Min	Max	Mean	Coefficient of Variation	Fraction (%)
Exploration	29	9643	1373	0.52	54.96
Combat	19	8719	1500	0.59	45.04

Table 3.4: Overall frame statistics throughout an episode

The minimum, maximum, and mean frame sizes illustrated in Table 3.4 are measured in bytes, and they indicate the statistics of frames seen in each kind of game stage throughout the episode. Mean frame sizes of both stages are pretty close to each other. Therefore it is not a strong indicator here.

Coefficient of Variation (CV) is a statistical measure that evaluates the distribution of data samples around the Mean of the data. As seen in Equation (3.3), the Coefficient of Variation is the ratio of Standard Deviation to the Mean.

$$CV = \frac{\sigma}{\mu} \tag{3.3}$$

where σ is the Standard Deviation of the data, and μ is the Mean of the data.

Coefficient of Variation is a good measure in comparing the variations of different data series, even when the means of those series are totally different. According to the test by Forkman (2009) [18], the two Coefficient of Variation values belonging to the exploration and combat stages are significantly different from each other, considering the sample sizes. Therefore, it is possible to discriminate those stages from each other.

The fraction, which indicates the distribution of frames across game stages, is an essential indicator because if there is an imbalance in a dataset, Machine Learning models usually tend to learn the dominating data. However, even though the number of frames belonging to the Exploration stage is higher than the number of frames belonging to the Combat stage in this case, there is no significant difference to cause a bias. Therefore there is no need to use techniques such as oversampling to balance the dataset.

Table 3.5 shows the overall bitrate statistics per game stage throughout the episode. The bitrate values are measured in *kbps*. Mean bitrates of both stages are similar; therefore, it is hard to deduce by only looking at this indicator.

Game Stage	Min Bitrate	Max Bitrate	Mean Bitrate
Exploration	89.60	794.72	392.45
Combat	76.88	754.49	408.80

Table 3.5: Overall bitrate statistics throughout an episode

Figure 3.13 indicates the dispersion of frames of each game stage over the different bitrate bands.

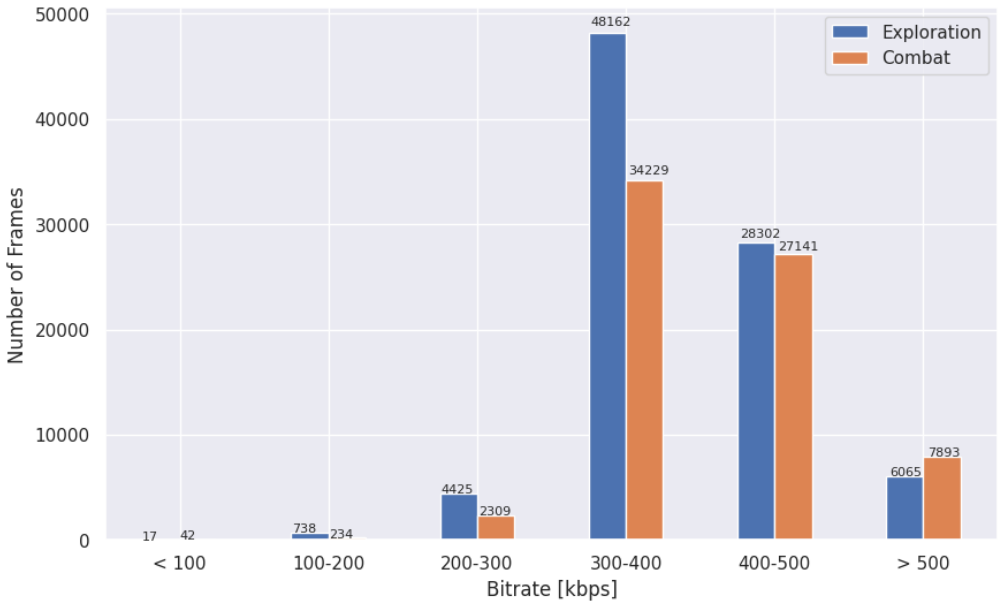


Figure 3.13: Frame distribution histogram over the different bitrate bands

According to Figure 3.13, the number of frames that belong to the bitrate band up to 200 kbps is significantly low compared to the rest of the bitrate bands. Therefore, they are neglectable in the context of this experiment. On the other hand, the vast amount of frames belong to the bitrate band between 300 kbps and 500 kbps, especially from 300 kbps to 400 kbps. Hence, this band is the most important one.

Transition Matrix 3.6 below indicates the transition frequencies from one stage to another. According to the result, it is possible to observe that most of the transitions happen between the same game stages.

Game Stage	Exploration	Combat
Exploration	0.98	0.02
Combat	0.02	0.98

Table 3.6: Game stage transition matrix

Smoothing with Half-second Time Window

According to Equation (3.2), Smoothing Factor α equals approximately 0.105 for given window length 18, and this results in similar results in the former evaluation. Following statistics are staged below for completeness.

Game Stage	Min	Max	Mean	Coefficient of Variation	Fraction [%]
Exploration	23	9643	1373	0.52	55.02
Combat	19	8719	1499	0.59	44.98

Table 3.7: Overall frame statistics throughout an episode

Changing the window length w from 9 to 18 affects the game stages slightly. By considering the total number of frames in the current episode, this adjustment results in the change of approximately 100 frames. However, there is still no significant difference to cause a bias. Besides, the two Coefficient of Variation values belonging to the exploration and combat stages are still significantly different than each other, considering the sample sizes. Hence, it is still possible to discriminate those stages from each other.

Game Stage	Min Bitrate	Max Bitrate	Mean Bitrate
Exploration	93.20	775.21	392.51
Combat	82.60	729.94	408.74

Table 3.8: Overall bitrate statistics throughout an episode

Based on Tables 3.7 and 3.8, mean frame sizes and mean bitrate values of the both stages are highly close each other. Therefore, it would not be reliable to make an observation by only looking at these indicators.

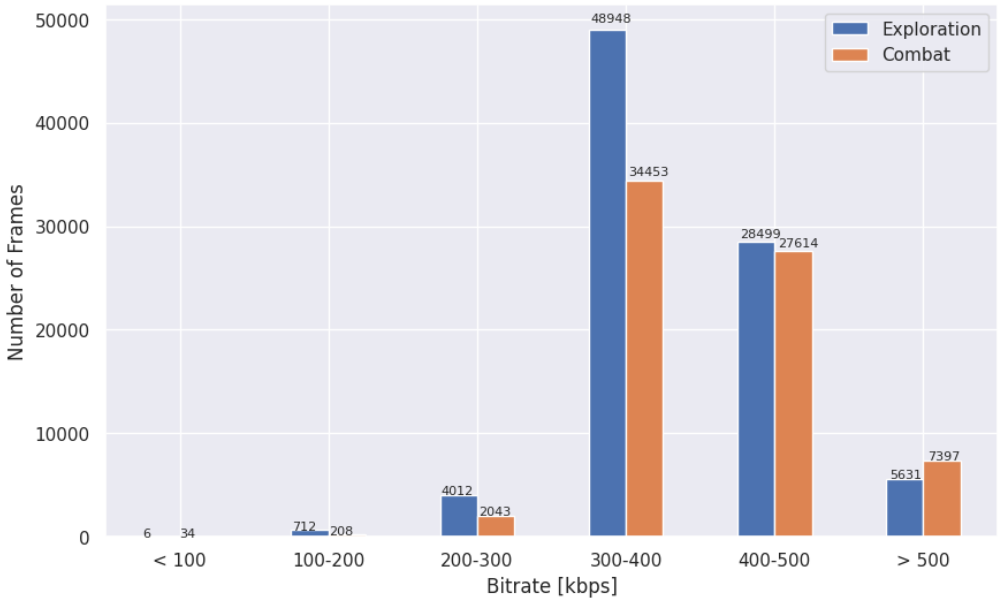


Figure 3.14: Frame distribution histogram over the different bitrate bands

Since the distribution of frames over exploration and combat stages is almost equal to the scenario in the former evaluation, distribution and transition of the frames are pretty identical to the former case. The bitrate band up to 200 kbps, even up to 300 kbps, is still neglectable. The most critical bitrate band is still the band starting from 300 kbps to 500 kbps.

Game Stage	Exploration	Combat
Exploration	0.98	0.02
Combat	0.02	0.98

Table 3.9: Game stage transition matrix

Transition Matrix 3.9 indicates that changing the Smoothing Factor does not affect the fact that the great majority of the transitions occur between the scenes belonging to the same game stage.

Smoothing with One-second Time Window

According to Equation (3.2), Smoothing Factor α equals approximately 0.055 for given window length 35, and this results in highly similar results in the former evaluations. Following statistics are staged below for completeness.

Game Stage	Min	Max	Mean	Coefficient of Variation	Fraction (%)
Exploration	23	9643	1372	0.52	55.14
Combat	19	8719	1500	0.60	44.86

Table 3.10: Overall frame statistics throughout an episode

Game Stage	Min Bitrate	Max Bitrate	Mean Bitrate
Exploration	101.14	733.17	392.32
Combat	98.97	687.63	409.02

Table 3.11: Overall bitrate statistics throughout an episode

According to Tables 3.10 and 3.11, mean frame sizes and mean bitrate values in both game stages are still close to each other. Still, using these metrics would not be reliable.

As expected, adjustment of the window length w from 18 to 35 still does not bring much difference. By considering the total number of frames in the current episode, this adjustment results in the change of approximately 300 frames. Thus, there is still no significant difference to cause a bias.

Game Stage	Exploration	Combat
Exploration	0.99	0.01
Combat	0.02	0.98

Table 3.12: Game stage transition matrix

Transition Matrix 3.12 shows that the number of in-stage transitions in the exploration stage is slightly higher than the previous cases, and in the same sense, the number of transitions from exploration stage to combat stage is slightly lower than the previous cases. However, the difference is neglectable.

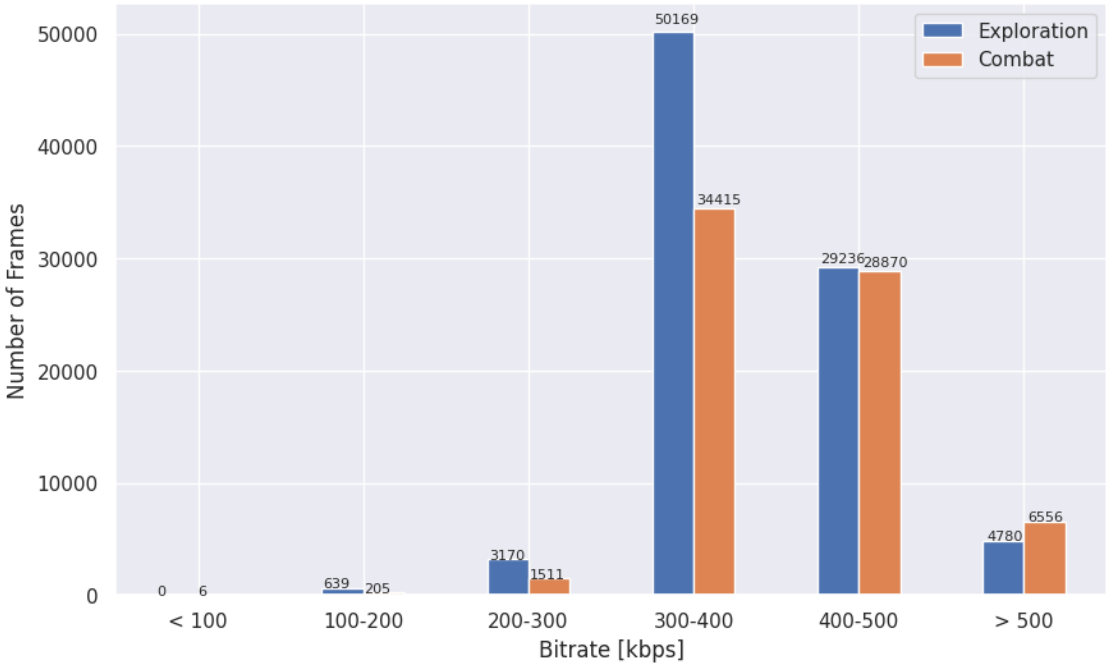


Figure 3.15: Frame distribution histogram over the different bitrate bands

Since the distribution of frames over exploration and combat stages is almost equal to the scenarios in the former evaluations, distribution and transition of the frames are quite the same in the former case. The bitrate band up to 200 kbps, even up to 300 kbps, is still neglectable. The most critical bitrate band is still the band starting from 300 kbps to 500 kbps.

Considering the results of all three smoothing configurations, changing the smoothing window length w does not make a significant difference in game stage average bitrates, yet it is still helpful to decrease the level of noise.

3.6.2 Variation in Average Bitrate of Adjacent Series of Stages

One approach followed in this thesis study -to deduct a pattern among game stage transitions- is to evaluate the average bitrate values stage-by-stage and how the stages' average bitrates are affected during transitions from the exploration stage to combat stage and vice versa.

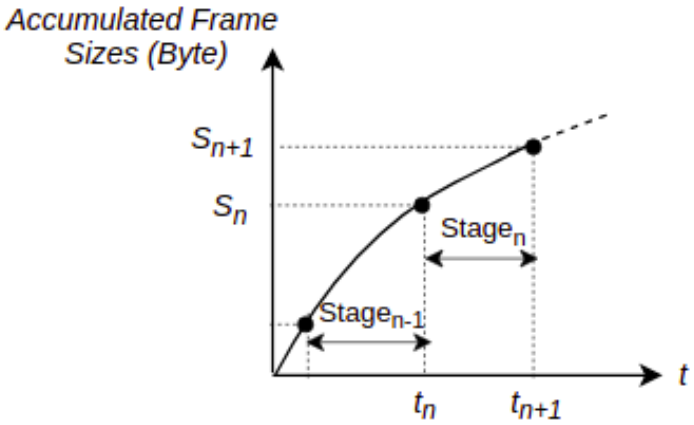


Figure 3.16: Average bitrate calculation per stage

$$\mu_n = \frac{S_{n+1} - S_n}{t_{n+1} - t_n} \cdot 0.008 = \frac{\sum_{t=n}^{n+1} s_t}{t_{n+1} - t_n} \cdot 0.008 \tag{3.4}$$

where μ_n is the average bitrate of a game stage, S_n is the accumulated size in bytes at the beginning of a stage n, S_{n+1} is the accumulated size in bytes at the end of a stage n, and s_t is the frame size in bytes at any time t.

Pseudocode 4 illustrates the mechanism of average bitrate calculation stage-by-stage.

Pseudocode 4: Average Bitrate Calculation per Game Stage

```
Smooth the bitrate and game stage data                          ▷ Pseudocode 3
foreach (index, row) in dataset.csv do
    if index = 0 then
        accumulatedSize ← row["size"]
        row['accSize']      ← accumulatedSize
        continue
    end if
    row['accSize']      ← row['size'] + accumulatedSize
    accumulatedSize ← row['accSize']
end foreach
avgBitrates ← dict()
foreach (index, row) in dataset.csv do
    if index = 0 then
        tmpRow ← row
        continue
    end if
    if row["gamestage"] != tmpRow["gamestage"] then              ▷ Stage change
        timeDiff      ← row['time'] − tmpRow['time']
        stageBitrate  ← (row['accSize'] − tmpRow['accSize']) / timeDiff
        timeMidStage ← tmpRow['time'] + timeDiff / 2
        avgBitrates[timeMidStage] ← (stageBitrate, tmpRow['gamestage'])
        tmpRow ← row
    end if
end foreach
```

After calculating the average bitrate of each adjacent game stage, data structure *avgBitrates* in Pseudocode 4 keeps each game stage's average bitrate and the corresponding game stage in its values. Thus, it is easy to calculate the variation of average bitrates among adjacent series of game stages by iterating over this data.

Considering the transition from stage *n-1* to the stage n in Figure 3.16, the variation of average bitrates is calculated by Equation (3.5).

$$Var = \frac{\mu_n - \mu_{n-1}}{\mu_{n-1}} \tag{3.5}$$

where μ_n is the average bitrate of the latter stage, μ_{n-1} is the average bitrate of the former stage.

μ_{n-1}, the former stage's average bitrate, is the baseline bitrate. According to baseline bitrates, each variation is categorized in one bandwidth class to achieve fine-grained evaluation.

Smoothing with Quarter-second Time Window

Since the episode is approximately 72 minutes of gameplay, it is impossible to fit all the average bitrate indicators into a single figure in terms of visibility. Therefore, all the figures are limited to one and a half minutes of gameplay to see a pattern.

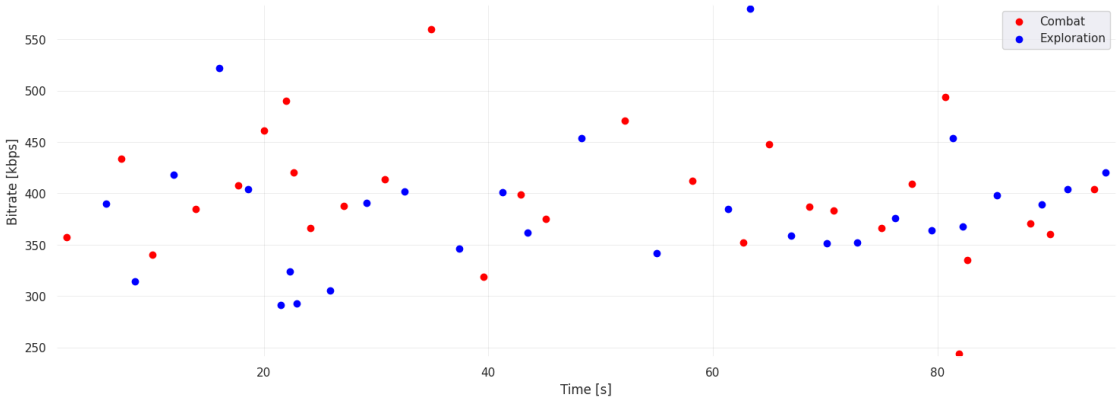


Figure 3.17: Average bitrates per stage, smoothed with window length w 9

Tables 3.13 and 3.14 depict the average bitrate variation statistics of game stage transitions between the exploration and the combat stages. The leftmost column indicates the baseline bitrate, the average bitrate of the former stage just before transition.

Exploration → Combat	# Transitions	Min Var.	Max Var.	Avg Var.
0-200 kbps	9	40.82 %	339.05 %	135.85 %
200-300 kbps	99	- 51.22 %	228.16 %	48.98 %
300-350 kbps	258	- 50.93 %	125.71 %	20.42 %
350-400 kbps	687	- 54.69 %	297.30 %	8.94 %
400-450 kbps	400	- 50.35 %	64.76 %	- 0.11 %
450-500 kbps	121	- 55.97 %	126.30 %	- 6.4 %
500-∞ kbps	104	- 75.05 %	32.40 %	- 21.42 %

Table 3.13: Average bitrate variation stats, transitions from exploration to combat

Combat → Exploration	# Transitions	Min Var.	Max Var.	Avg Var.
0-200 kbps	5	55.71 %	184.28 %	103.46 %
200-300 kbps	93	- 55.70 %	143.85 %	36.89 %
300-350 kbps	213	- 58.94 %	110.84 %	16.07 %
350-400 kbps	497	- 67.01 %	263.38 %	1.67 %
400-450 kbps	409	- 59.95 %	85.18 %	- 6.91 %
450-500 kbps	243	- 54.96 %	59.42 %	- 14.5 %
500-∞ kbps	218	- 75.41 %	113.74 %	- 24.16 %

Table 3.14: Average bitrate variation stats, transitions from combat to exploration

As seen in Figure 3.13, since the number of frames up to 200 kbps is quite neglectable, the tables above spare only one chunk for this bitrate range while fragmenting the rest of the ranges by the length of 50 kbps chunks. The second column indicates the number of transitions for each bitrate range.

Tables 3.13 and 3.14 indicate that most of the transitions occur in the baseline bitrate band between 350 kbps and 450 kbps, while the number of the transitions up to 200 kbps is almost zero as expected.

All the transitions from the exploration stage to the combat stage and vice versa up to the baseline bitrate 200 kbps always increase the bitrate, considering the minimum variation in this band is positive. Besides, according to the average variation values, one can say that the most significant variations usually occur in this band. On the other hand, while the value of baseline bitrate increases up to 400 kbps, the value of average variation decreases, which makes total sense. Meanwhile, according to the average variation values, transitions in both directions until the baseline bitrate of 400 kbps usually increase the bitrate. The bitrate seems to remain still in the transitions from the exploration stage to the combat stage from 400 kbps to 450 kbps. However, transitions in the opposite direction usually decrease the bitrate in the same band. After the baseline bitrate of 450 kbps, transitions in both directions tend to decrease the bitrate. Transitions after the baseline bitrate of 500 kbps can decrease the average bitrate even up to 75%.

Should the two transitions are compared by considering the average variation values, one can say that the transitions from the exploration stage to the combat stage up to the baseline bitrate of 350 kbps usually increase the bitrate more than the opposite transitions in the same baseline bitrate band. Likewise, the transitions from the combat stage to the exploration stage after the baseline bitrate of 400 kbps usually decrease the bitrate more than the opposite transitions in the same baseline bitrate band.

Smoothing with Half-second Time Window

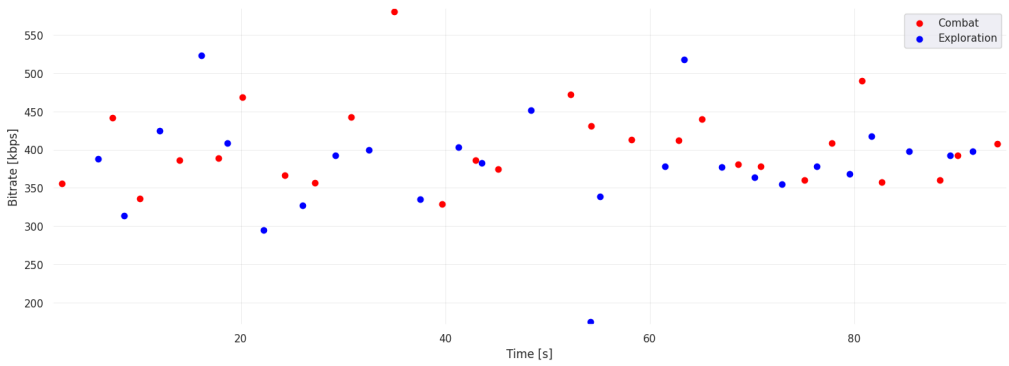


Figure 3.18: Average bitrates per stage, smoothed with window length w 18

Exploration → Combat	# Transitions	Min Var.	Max Var.	Avg Var.
0-200 kbps	6	105.71 %	404.29 %	165.31 %
200-300 kbps	54	- 22.67 %	147.64 %	42.66 %
300-350 kbps	210	- 43.22 %	123.89 %	20.95 %
350-400 kbps	609	- 55.01 %	85.19 %	8.43 %
400-450 kbps	370	- 52.27 %	95.80 %	- 0.72 %
450-500 kbps	101	- 49.69 %	35.16 %	- 9.31 %
500-∞ kbps	82	- 66.90 %	43.06 %	- 20.14 %

Table 3.15: Average bitrate variation stats, transitions from exploration to combat

Combat → Exploration	# Transitions	Min Var.	Max Var.	Avg Var.
0-200 kbps	4	100.00 %	141.14 %	121.68 %
200-300 kbps	74	- 15.57 %	134.45 %	44.31 %
300-350 kbps	172	- 54.97 %	117.70 %	14.83 %
350-400 kbps	448	- 81.72 %	119.95 %	3.34 %
400-450 kbps	356	- 55.20 %	224.66 %	- 5.89 %
450-500 kbps	203	- 71.63 %	69.83 %	- 14.74 %
500-∞ kbps	175	- 59.62 %	113.74 %	- 24.08 %

Table 3.16: Average bitrate variation stats, transitions from combat to exploration

Tables 3.15 and 3.16 indicate that most of the transitions occur in the baseline bitrate band between 350 kbps and 450 kbps, while the number of the transitions up to 200 kbps is almost zero as expected.

All the transitions from the exploration stage to the combat stage and vice versa up to the baseline bitrate 200 kbps always increase the bitrate, considering the minimum variation in this band is positive. Besides, one can say that the most significant variations usually occur in this band according to the average variation values.

While the value of baseline bitrate increases up to 400 kbps, the value of average variation decreases, which makes total sense. Meanwhile, according to the average variation values, transitions in both directions until the baseline bitrate of 400 kbps usually increase the bitrate. In the transitions from the exploration stage to the combat stage in the band between 400 kbps and 450 kbps, the bitrate seems to remain still. However, transitions in the opposite direction in the same band usually decrease the bitrate. After the baseline bitrate of 450 kbps, transitions in both directions tend to decrease the bitrate.

After the baseline bitrate of 500 kbps, transitions from the exploration stage to the combat stage and vice versa can decrease the average bitrate up to 67% and 60%, respectively. These values are lower than the values in the former case, smoothing with the window length w 9.

Should the two transitions are compared considering the average variation values, one can say that the transitions from the exploration stage to the combat stage up to the baseline bitrate 350 kbps usually increase the bitrate more than the opposite transitions in the same baseline bitrate band. In the same sense, the transitions from the combat stage to the exploration stage after the baseline bitrate 400 kbps usually decrease the bitrate more than the opposite transitions in the same baseline bitrate band.

Smoothing with One-second Time Window

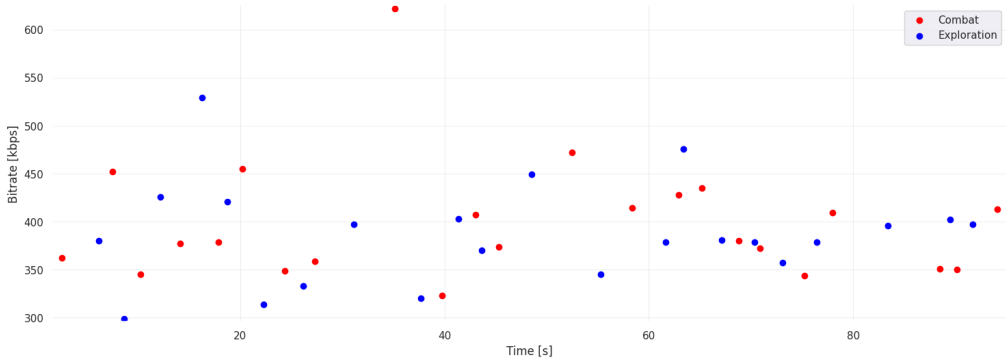


Figure 3.19: Average bitrates per stage, smoothed with window length w 35

Exploration → Combat	# Transitions	Min Var.	Max Var.	Avg Var.
0-200 kbps	6	80.57 %	308.87 %	145.25 %
200-300 kbps	45	- 25.64 %	123.08 %	42.29 %
300-350 kbps	148	- 33.33 %	79.47 %	19.31 %
350-400 kbps	536	- 43.85 %	84.55 %	9.30 %
400-450 kbps	317	- 47.76 %	86.89 %	- 1.63 %
450-500 kbps	80	- 50.91 %	28.78 %	- 7.55 %
500-∞ kbps	58	- 62.43 %	28.13 %	- 20.69 %

Table 3.17: Average bitrate variation stats, transitions from exploration to combat

Combat → Exploration	# Transitions	Min Var.	Max Var.	Avg Var.
0-200 kbps	1	135.63 %	135.63 %	135.63 %
200-300 kbps	61	- 27.84 %	140.89 %	40.20 %
300-350 kbps	132	- 59.08 %	90.79 %	16.46 %
350-400 kbps	361	- 60.96 %	100.55 %	3.51 %
400-450 kbps	324	- 50.11 %	51.24 %	- 6.49 %
450-500 kbps	179	- 61.46 %	39.41 %	- 15.60 %
500-∞ kbps	132	- 49.91 %	58.78 %	- 24.04 %

Table 3.18: Average bitrate variation stats, transitions from combat to exploration

Tables 3.17 and 3.18 indicate that the most of the transitions occur in the baseline bitrate band between 350 kbps and 450 kbps, while the number of the transitions up to 200 kbps is almost zero as expected.

Considering the minimum bitrate variation up to the baseline bitrate of 200 kbps is positive in both cases, all the transitions between the exploration stage and the combat stage always increase the bitrate in this band, even though the transitions from the combat stage to the exploration stage can be accepted as an outlier since there is only one transition in this band. Besides, according to the average variation values, one can say that the most considerable variations usually occur in this band.

As the value of baseline bitrate increases up to 400 kbps, the value of average variation decreases, which makes total sense. Meanwhile, according to the average bitrate variation values, transitions in both directions until the baseline bitrate of 400 kbps usually increase the bitrate. After the baseline bitrate of 400 kbps, transitions in both directions tend to decrease the bitrate more and more.

After the baseline bitrate 500 kbps, transitions from the exploration stage to the combat stage and vice versa can decrease the average bitrate up to 62% and 50%, respectively. These values are lower than the values in the former case, smoothing

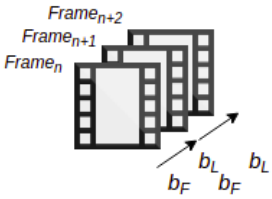
with the window length w 18.

Should the two transitions are compared by considering the average bitrate variation values, one can say that the transitions from the exploration stage to the combat stage up to the baseline bitrate of 350 kbps usually increase the bitrate more than the opposite transitions in the same baseline bitrate band. In the same sense, the transitions from the combat stage to the exploration stage after the baseline bitrate of 400 kbps usually decrease the bitrate more than the opposite transitions in the same baseline bitrate band.

According to the results of the three test configurations, the average bitrate variation values indicate that it can be possible to discriminate the exploration and the combat game stages during most of the baseline bitrate bands, except the baseline bitrate band between 400 kbps and 450 kbps.

3.6.3 Bitrate Variation During Intra-stage and Inter-stage Transitions

In this section of the study, the question of *"What can be observed if the bitrate variations during intra-stage and inter-stage transitions are analyzed?"* is asked to understand whether there is a pattern between the transitions. For this purpose, variation of each transition is calculated by Equation (3.6).



$$Var = \frac{b_L - b_F}{b_F} \tag{3.6}$$

where b_L is the bitrate after the transition, and b_F is the bitrate before the transition (baseline bitrate).

Pseudocode 5 illustrates the calculation of intra-stage and inter-stage transition variations.

Pseudocode 5: Calculation of Intra-stage and Inter-stage Transition Variations

```
1: Smooth the bitrate and game stage data                    ▷ Pseudocode 3
2: varExpCombat, varCombatExp ← dict()
3: varExpExp, varCombatCombat ← dict()
4: i ← 0
5: while index < len(data) - 1 do
6:     formerStage, latterStage ← data[index, stage], data[index + 1, stage]
7:     baselineBitrate ← data[index, bitrate]
8:     variation ← (data[index+1, bitrate] − baselineBitrate) / baselineBitrate
9:     if formerStage = EXP and latterStage = CMB then
10:        varExpCombat[baselineBitrate] ← variation
11:    else if formerStage = CMB and latterStage = EXP then
12:        varCombatExp[baselineBitrate] ← variation
13:    else if formerStage = EXP and latterStage = EXP then
14:        varExpExp[baselineBitrate] ← variation
15:    else if formerStage = CMB and latterStage = CMB then
16:        varCombatCombat[baselineBitrate] ← variation
17:    end if
18:    i ← i + 1
19: end while
```

In Pseudocode 5, *EXP* and *CMB* are used as global variables to define the exploration and combat stages. At the end, the data structures *varExpCombat*, *varCombatExp*, *varExpExp* and *varCombatCombat* keep the relevant transitions as key-value pairs where the baseline bitrate as a key, and the corresponding variation as a value.

After saving the transitions, each data structure is split into different baseline bitrate bands, and the relevant operations such as getting the minimum, maximum, mean are applied.

The analysis results of all transition directions are illustrated in the following tables.

Exploration → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 1.15 %	0.47 %	- 0.22 %
200-300 kbps	- 1.34 %	4.78 %	0.81 %
300-350 kbps	- 2.10 %	3.90 %	0.23 %
350-400 kbps	- 3.44 %	3.37 %	0.16 %
400-450 kbps	- 3.39 %	2.53 %	- 0.07 %
450-500 kbps	- 2.32 %	2.06 %	- 0.20 %
500-∞ kbps	- 3.37 %	1.58 %	- 0.23 %

Table 3.19: Bitrate variation stats, transitions from exploration to combat

Combat → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	1.80 %	5.33 %	3.62 %
200-300 kbps	- 1.75 %	3.13 %	0.53 %
300-350 kbps	- 1.61 %	1.80 %	0.03 %
350-400 kbps	- 2.67 %	2.21 %	0.05 %
400-450 kbps	- 1.83 %	2.36 %	0.13 %
450-500 kbps	- 2.17 %	2.52 %	- 0.08 %
500-∞ kbps	- 1.93 %	1.69 %	0.06 %

Table 3.20: Bitrate variation stats, transitions from combat to exploration

Exploration → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 1.15 %	0.47 %	- 0.22 %
200-300 kbps	- 1.34 %	4.78 %	0.81 %
300-350 kbps	- 2.10 %	3.90 %	0.23 %
350-400 kbps	- 3.44 %	3.37 %	0.16 %
400-450 kbps	- 3.39 %	2.53 %	- 0.07 %
450-500 kbps	- 2.32 %	2.06 %	- 0.20 %
500-∞ kbps	- 3.37 %	1.58 %	- 0.23 %

Table 3.21: Bitrate variation stats, transitions from exploration to exploration

Combat → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	1.80 %	5.33 %	3.62 %
200-300 kbps	- 1.75 %	3.13 %	0.53 %
300-350 kbps	- 1.61 %	1.80 %	0.03 %
350-400 kbps	- 2.67 %	2.21 %	0.05 %
400-450 kbps	- 1.83 %	2.36 %	0.13 %
450-500 kbps	- 2.17 %	2.52 %	- 0.08 %
500-∞ kbps	- 1.93 %	1.69 %	0.06 %

Table 3.22: Bitrate variation stats, transitions from combat to combat

According to the tables above, the average bitrate variation values indicate that analyzing frame-to-frame transitions, whether intra-stage or inter-stage, does not provide helpful insight into the data. These values show that the bitrate is not affected significantly during single frame transitions. Therefore, this methodology became impractical for this thesis study.

3.6.4 Bitrate Variation on a Time Span Around Stage Transitions

Another approach followed in this thesis study to deduct a pattern among game stage transitions is calculating the bitrate variation in each game stage transition moment by taking that moment as an origin and evaluating the bitrate values k frames before and k frames after the origin.

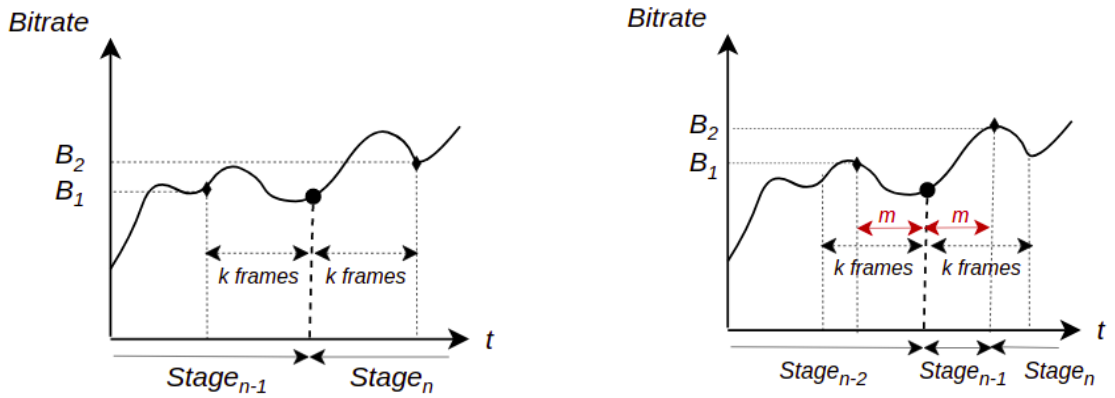


Figure 3.20: Calculation of bitrate variation around stage transitions

The point that should be noted in these figures while calculating the bitrate variation is the difference in the locations of markers B_1 and B_2. Bitrate variations

around stage transition moments are calculated by Equation (3.7).

$$Var = \frac{B_2 - B_1}{B_1} \tag{3.7}$$

As mentioned above, the methodology proposes to advance k frames back and forth around the moment of game stage transitions. By taking the game stage transition moment as the origin, the left figure is the case of observing the same former game stage in each back-pass through k frames and observing the same latter game stage in each forward-pass through k frames without an interruption of another game stage.

However, if there is an intervention by another game stage in one of the directions (in m-th frame from the origin), bitrate variation calculation is performed as in the right figure.

Pseudocode 6 illustrates the logic behind the methodology. Line 10 illustrates the decision depicted in Figure 3.20.

Pseudocode 3.6: Calculation of Bitrate Variation Around Stage Transitions

```
1:  Smooth the bitrate and game stage data                    ▷ Pseudocode 3
2:  frameRange      ← k                                      ▷ k = 9 or 18
3:  bitrateVariances ← dict()
4:  foreach (index, row) in dataset.csv do
5:      if index = 0 then
6:          tmpRow ← row
7:          continue
8:      end if
9:      if row['gamestage'] != tmpRow['gamestage'] then      ▷ Stage change
10:         if another transition m frames back or forth then   ▷ m < frameRange
11:             baselineBitrate ← dataset.csv[index − m]
12:             latterBitrate   ← dataset.csv[index + m]
13:         else
14:             baselineBitrate ← dataset.csv[index − k]
15:             latterBitrate   ← dataset.csv[index + k]
16:         end if
17:         key ← (row["time"], baselineBitrate)
18:         variation ← (latterBitrate − baselineBitrate) / baselineBitrate
19:         bitrateVariances[key] ← (variation, tmpRow['stage'], row['stage'])
20:         tmpRow ← row
21:     end if
22: end foreach
```

After calculating the bitrate variation over a time span around each game stage transition, data structure $bitrateVariances$ in Pseudocode 6 keeps each transition time stamp, baseline bitrate when a transition occurred, bitrate variation due to a transition, and game stages before and after a transition. By iterating over this data structure, it is easy to group the transitions for statistical analyses based on the baseline bitrates and the transition directions.

As in previous analyses, this methodology is also evaluated in three different smoothing configurations.

Smoothing with Quarter-second Time Window

In each smoothing configuration, the experiments are conducted using two different frame ranges for the transitions from the exploration stage to the combat stage and vice versa.

The tables below depict the bitrate variation statistics over a time span around game stage transitions between the exploration and the combat stages, depending on frame range k. The leftmost column indicates the baseline bitrate, B_1 in Figure 3.20.

Experiment with Frame Range k 9

Exploration → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 23 %	73 %	25.50 %
200-300 kbps	- 27 %	66 %	15.59 %
300-350 kbps	- 17 %	58 %	7.80 %
350-400 kbps	- 25 %	34 %	2.00 %
400-450 kbps	- 28 %	27 %	- 2.28 %
450-500 kbps	- 25 %	24 %	- 4.35 %
500-∞ kbps	- 39 %	18 %	- 7.22 %

Table 3.23: Bitrate variation stats around transitions from exploration to combat

Combat → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	45 %	173 %	84.75 %
200-300 kbps	- 21 %	78 %	13.10 %
300-350 kbps	- 14 %	36 %	5.73 %
350-400 kbps	- 22 %	44 %	2.70 %
400-450 kbps	- 21 %	35 %	0.41 %
450-500 kbps	- 38 %	30 %	- 1.69 %
500-∞ kbps	- 32 %	31 %	- 3.96 %

Table 3.24: Bitrate variation stats around transitions from combat to exploration

Experiment with Frame Range *k* 18

Exploration → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 26 %	149 %	37.46 %
200-300 kbps	- 45 %	90 %	23.04 %
300-350 kbps	- 17 %	73 %	10.44 %
350-400 kbps	- 29 %	50 %	3.20 %
400-450 kbps	- 34 %	41 %	- 3.23 %
450-500 kbps	- 31 %	24 %	- 7.69 %
500-∞ kbps	- 51 %	24 %	- 12.82 %

Table 3.25: Bitrate variation stats around transitions from exploration to combat

Combat → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	45 %	300 %	119.17 %
200-300 kbps	- 21 %	112 %	24.27 %
300-350 kbps	- 25 %	83 %	9.90 %
350-400 kbps	- 43 %	63 %	3.86 %
400-450 kbps	- 37 %	70 %	0.01 %
450-500 kbps	- 38 %	43 %	- 2.53 %
500-∞ kbps	- 42 %	39 %	- 7.78 %

Table 3.26: Bitrate variation stats around transitions from combat to exploration

Since the number of frames up to the baseline bitrate of 200 kbps is quite neglectable as seen in Figure 3.13, the tables above spare only one chunk for this bitrate band while fragmenting the rest of the bands by the length of 50 kbps chunks. The second column indicates the number of transitions for each bitrate range.

If the transitions in the same direction are compared according to the frame range k, it is possible to say that selecting the frame range k 18 would affect the bitrate more than the case of selecting the frame range k 9. More strictly speaking, up to baseline bitrate 400 kbps, selecting the frame range k 18 increases the bitrate more than the case of selecting the frame range k 9. On the contrary, selecting the frame range k 18 decreases the bitrate more than the other case after the baseline bitrate band of 400 kbps.

Regardless of the frame range k, the values of average variation in the tables indicate that while transitions in the direction of combat stage up to the baseline bitrate of 200 kbps usually increases the bitrate, transitions in the opposite direction in the same band always increases the bitrate, since the corresponding values of minimum variance are positive. Besides, considering the average variation values, one can say that the most significant bitrate variations usually occur in this band. Transitions from the combat stage to the exploration stage up to the baseline bitrate of 400 kbps usually increase the bitrate more than the opposite transitions in the same baseline bitrate band. In the same sense, the transitions from the exploration stage to the combat stage after the baseline bitrate of 400 kbps usually decrease the bitrate more than the opposite transitions in the same baseline bitrate band.

In all cases above, as the value of baseline bitrate increases up to 400 kbps, the value of average variation decreases, as expected. Meanwhile, according to the average variation values, transitions in both directions until the baseline bitrate of 400 kbps usually increase the bitrate. During transitions from the combat stage to the exploration stage in the baseline bitrate band between 400 kbps and 450 kbps, the bitrate seems to remain still, however in the same band, transitions in the opposite direction usually slightly decrease the bitrate. After the baseline bitrate of 450 kbps, transitions in both directions tend to decrease the bitrate.

Smoothing with Half-second Time Window

Experiment with Frame Range *k* 9

Exploration → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 17 %	57 %	17.83 %
200-300 kbps	- 25 %	71 %	15.81 %
300-350 kbps	- 14 %	56 %	6.30 %
350-400 kbps	- 22 %	31 %	1.94 %
400-450 kbps	- 23 %	23 %	- 2.11 %
450-500 kbps	- 26 %	15 %	- 4.85 %
500-∞ kbps	- 34 %	15 %	- 7.36 %

Table 3.27: Bitrate variation stats around transitions from exploration to combat

Combat → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	26 %	136 %	64.25 %
200-300 kbps	- 17 %	63 %	13.18 %
300-350 kbps	- 14 %	43 %	5.15 %
350-400 kbps	- 20 %	36 %	2.44 %
400-450 kbps	- 19 %	34 %	0.51 %
450-500 kbps	- 32 %	25 %	- 2.16 %
500-∞ kbps	- 26 %	30 %	- 2.57 %

Table 3.28: Bitrate variation stats around transitions from combat to exploration

Experiment with Frame Range *k* 18

Exploration → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 31 %	139 %	27.75 %
200-300 kbps	- 46 %	91 %	22.66 %
300-350 kbps	- 16 %	59 %	9.19 %
350-400 kbps	- 31 %	45 %	2.89 %
400-450 kbps	- 26 %	33 %	- 2.86 %
450-500 kbps	- 30 %	21 %	- 8.08 %
500-∞ kbps	- 47 %	22 %	- 12.88 %

Table 3.29: Bitrate variation stats around transitions from exploration to combat

Combat → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	26 %	253 %	94.83 %
200-300 kbps	- 17 %	111 %	22.73 %
300-350 kbps	- 20 %	55 %	8.56 %
350-400 kbps	- 38 %	60 %	3.64 %
400-450 kbps	- 34 %	68 %	0.18 %
450-500 kbps	- 32 %	41 %	- 2.57 %
500-∞ kbps	- 38 %	35 %	- 6.55 %

Table 3.30: Bitrate variation stats around transitions from combat to exploration

If the transitions in the same direction are compared according to the frame range k, it is possible to say that selecting the frame range k 18 would affect the bitrate more than the case of selecting the frame range k 9. More strictly speaking, up to baseline bitrate 400 kbps, selecting the frame range k 18 increases the bitrate more than the case of selecting the frame range k 9. On the contrary, selecting the frame range k 18 decreases the bitrate more than the other case after the baseline bitrate band of 400 kbps.

Regardless of the frame range k, the values of average variation in the tables indicate that while transitions in the direction of combat stage up to the baseline bitrate 200 kbps usually increases the bitrate, transitions in the opposite direction in the same band always increases the bitrate since the corresponding values of minimum variance are positive. Besides, considering the average variation values, one can say that the most significant bitrate variations usually occur in this band. Transitions from the combat stage to the exploration stage up to the baseline bitrate 400 kbps usually increase the bitrate more than the opposite transitions in the same baseline bitrate band. In the same sense, the transitions from the exploration stage to the combat stage after the baseline bitrate 400 kbps usually decrease the bitrate more than the opposite transitions in the same baseline bitrate band.

As the value of baseline bitrate increases up to 400 kbps, the value of average variation decreases, as expected. Meanwhile, according to the average variation values, transitions in both directions until the baseline bitrate 400 kbps usually increase the bitrate. During transitions from the combat stage to the exploration stage in the baseline bitrate band between 400 kbps and 450 kbps, the bitrate seems to remain still, however in the same band, transitions in the opposite direction usually slightly decrease the bitrate. After the baseline bitrate 450 kbps, transitions in both directions tend to decrease the bitrate.

Increasing the smoothing window length w, hence decreasing the Smoothing Factor α, results in a slight decrease in bitrate variation, as expected.

Smoothing with One-second Time Window

Experiment with Frame Range *k* 9

Exploration → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 20 %	76 %	10.14 %
200-300 kbps	- 24 %	50 %	10.64 %
300-350 kbps	- 11 %	31 %	4.74 %
350-400 kbps	- 16 %	27 %	1.71 %
400-450 kbps	- 20 %	14 %	- 1.73 %
450-500 kbps	- 19 %	10 %	- 3.80 %
500-∞ kbps	- 26 %	11 %	- 6.59 %

Table 3.31: Bitrate variation stats around transitions from exploration to combat

Combat → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	1 %	97 %	34.67 %
200-300 kbps	- 11 %	34 %	9.21 %
300-350 kbps	- 10 %	37 %	3.60 %
350-400 kbps	- 15 %	27 %	1.59 %
400-450 kbps	- 18 %	24 %	0.64 %
450-500 kbps	- 21 %	22 %	- 1.57 %
500-∞ kbps	- 19 %	23 %	- 1.96 %

Table 3.32: Bitrate variation stats around transitions from combat to exploration

Experiment with Frame Range *k* 18

Exploration → Combat	Min Var.	Max Var.	Avg Var.
0-200 kbps	- 20 %	86 %	21.43 %
200-300 kbps	- 45 %	110 %	16.64 %
300-350 kbps	- 15 %	59 %	7.36 %
350-400 kbps	- 24 %	44 %	2.66 %
400-450 kbps	- 23 %	22 %	- 2.33 %
450-500 kbps	- 33 %	19 %	- 7.08 %
500-∞ kbps	- 39 %	17 %	- 10.36 %

Table 3.33: Bitrate variation stats around transitions from exploration to combat

Combat → Exploration	Min Var.	Max Var.	Avg Var.
0-200 kbps	1 %	97 %	46.50 %
200-300 kbps	- 11 %	94 %	16.47 %
300-350 kbps	- 14 %	47 %	6.73 %
350-400 kbps	- 25 %	49 %	2.41 %
400-450 kbps	- 25 %	47 %	0.51 %
450-500 kbps	- 26 %	33 %	- 2.43 %
500-∞ kbps	- 32 %	28 %	- 5.56 %

Table 3.34: Bitrate variation stats around transitions from combat to exploration

Should the transitions in the same direction are compared according to the frame range k, it is possible to say that selecting the frame range k 18 would affect the bitrate more than the case of selecting the frame range k 9. Selecting the frame range k 18 increases the bitrate more than selecting the frame range k 9 up to the baseline bitrate of 400 kbps. On the contrary, selecting the frame range k 18 decreases the bitrate more than the other case after the baseline bitrate band of 400 kbps.

Regardless of the frame range k, the values of average variation in the tables indicate that while transitions in the direction of combat stage up to the baseline bitrate of 200 kbps usually increases the bitrate, transitions in the opposite direction in the same band always increases the bitrate, since the corresponding values of minimum variance are positive. Besides, considering the average variation values, one can say that the most significant bitrate variations usually occur in this band. Transitions from the combat stage to the exploration stage up to the baseline bitrate 400 kbps usually increase the bitrate more than the opposite transitions in the same baseline bitrate band. In the same sense, the transitions from the exploration stage to the combat stage after the baseline bitrate 400 kbps usually decrease the bitrate more than the opposite transitions in the same baseline bitrate band.

As the value of baseline bitrate increases up to 400 kbps, the value of average variation decreases, as expected. Meanwhile, according to the average variation values, transitions in both directions until the baseline bitrate 400 kbps usually increase the bitrate. After this point, transitions in both directions tend to decrease the bitrate.

This analysis explicates that increasing the smoothing window length w up to 35, corresponding to a window of one second, causes slightly more information loss than the other smoothing settings. Therefore, it would be better to proceed with smoothing with either window length w 9 or 18. It can be possible to discriminate the exploration and the combat game stages from each other in both cases.

3.7 Deep Learning Architecture and Frameworks Used in the Study

After having completed the dataset analysis, this section is devoted to explaining the fundamental concepts of Deep Learning. Before diving into the model structure, training, and test processes, it is essential to understand the core concepts such as Recurrent Neural Networks and the frameworks used in this thesis study.

3.7.1 Artificial Neural Networks

An Artificial Neural Network -usually referred Neural Network- is a set of cells (neurons) that emulates the working process of the human brain to learn and recognize underlying patterns or relations in vast amounts of data.

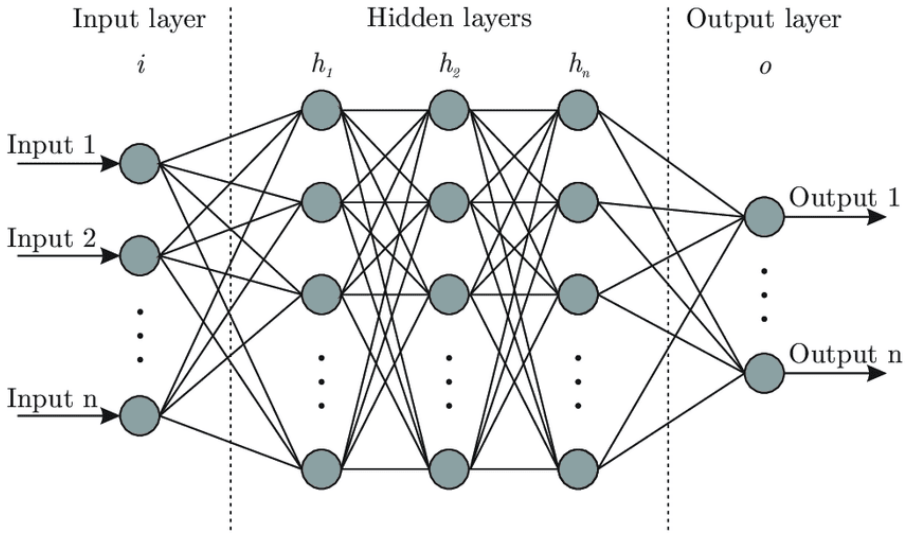


Figure 3.21: An illustration of a simple Neural Network, reproduced from [19]

A Neural Network is made up of layers of nodes that are linked together. These nodes are called neurons, and they are basically mathematical functions that collect and process information related to the network architecture. The connections between neurons are called edges. Neurons and edges have weights that the Neural Network itself continuously updates during the learning process.

Every single neuron in a network receives multiple inputs from the neurons that are interconnected with it, aggregates those inputs considering the weights (a weight may increase or decreases the input signal) of the edges of each input, and if and only if it is activated, feeds the forward neurons with its output.

Activation of a neuron is basically a mathematical decision, an *Activation Function* with a pre-defined threshold. A neuron is triggered if the aggregated and processed input crosses its threshold. Neurons in different layers in a network may operate different kinds of activations depending on the purpose of the application.

Neural Networks are gaining more and more popularity day by day and are widely used in various application domains, e.g., from fault detection in production to auto-piloting technologies and fraud detection in stock markets. One promising application of Neural Networks would be implementing the idea of this thesis study for better QoE assessments in Cloud Gaming.

3.7.2 Recurrent Neural Networks

Recurrent Neural Networks are specific application types of Artificial Neural Networks. They are tailored for sequential data, e.g., time-series, and they are commonly used in temporal use cases such as Natural Language Processing (NLP), Speech Recognition, and translation.

While Feed-forward Neural Networks assume that inputs and outputs are independent of one another, Recurrent Neural Networks differentiate themselves by using the information inherited from previous inputs while processing their current input. For example, the idiom “Break a leg!” means “Good luck!”. If this idiom is processed by a Feed-forward Neural Network, it would not be possible to capture the real meaning because the word “break” means literally “breaking something” for the network. This word gets a different meaning when combined with the word “leg", and only a Recurrent Neural Network can capture this relation due to their temporal nature.

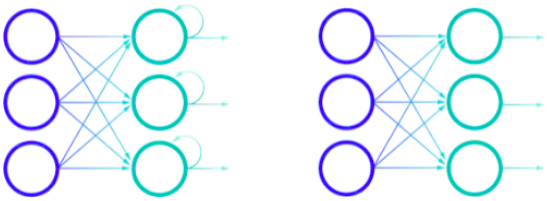


Figure 3.22: An illustration of Recurrent Neural Networks, reproduced from [20]

Figure 3.22 illustrates the main difference between Recurrent Neural Networks (the graph on the left) and Feed-forward Neural Networks (the graph on the right). The loops on the left represent the internal state, “memory”, to process sequences of inputs.

One of the critical differences between these two types of Neural Networks is that while Feed-forward Neural Networks have different weights on each node, Recurrent Neural Networks share the same weight within each network layer.

In order to understand the loops (internal states) on Figure 3.22, the figure below unrolls the representation above.

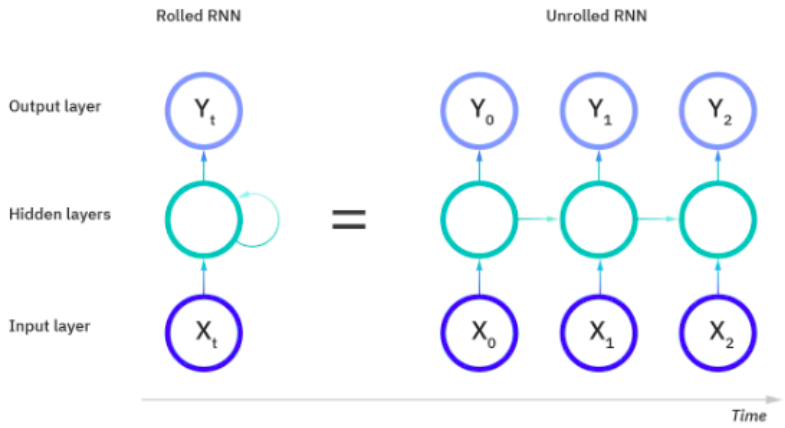


Figure 3.23: An unrolled illustration of RNNs, reproduced from [20]

3.7.3 LSTM: Long Short-term Memory

No matter how Recurrent Neural Networks seem quite helpful in processing data sequences in theory, they are having difficulties in capturing long-term dependencies. For example, a "vanilla" Recurrent Neural Network can do a pretty good job of predicting the last word in a sentence, such as "There are flowers in the garden.". However, when it comes to a sentence such as "I grew up in Turkey. I have spent my twenty years there and lived in several cities. I speak fluent Turkish.", the network most likely fails to capture this long-term dependency between the country and the language. Due to the problems such as *Vanishing Gradients* -which is not discussed in this thesis study since it is too much detail for the context-, as the gap between related inputs grows, the network becomes unsuccessful.

In order to tackle such problems, Long Short-term Memory (LSTM) is introduced by Hochreiter & Schmidhuber (1997) [21]. LSTM is the revolutionary implementation of Recurrent Neural Networks that can comprehend long-term dependencies. Because of their outstanding performance compared to the traditional Recurrent Neural Networks, LSTMs are widely used today.

The figures depicted below illustrates the structural difference between standard Recurrent Neural Networks and Long Short-term Memory Networks.

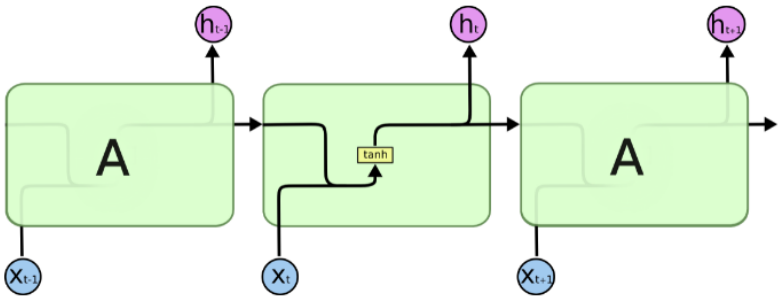


Figure 3.24: Cell structure of a traditional RNN, reproduced from [22]

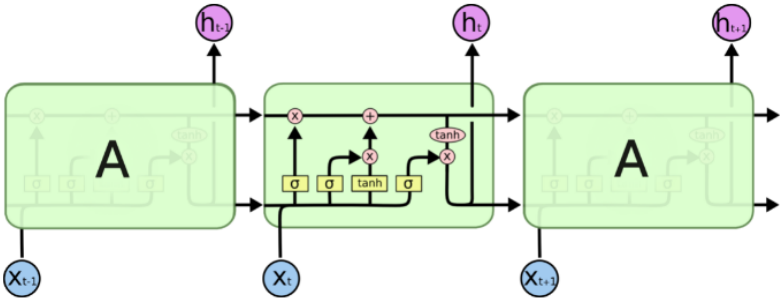


Figure 3.25: Cell structure of an LSTM Network, reproduced from [22]

While the repeating cells in traditional Recurrent Neural Networks are composed of a single *tanh* Layer, LSTMs replace this structure with four layers as seen in Figure 3.25.

The top horizontal line seen in Figure 3.25 is the most crucial element in LSTMs, which is the cell state. Cell state flows from a cell to another through this flow. LSTM can manipulate the cell state through the gates. The Sigmoid Layer (σ) controls how much data should be passed through. It outputs a value ranging from 0 to 1, meaning "block all" and "let all pass", respectively.

3.7.4 PyTorch and PyTorch Lightning

PyTorch is an open-source Machine Learning framework developed by Facebook AI. Characteristically, it is a modular and flexible Deep Learning framework designed for academic research, product development, and product deployment. It is an optimized Tensor library that interacts with Graphics Processing Unit (GPU) and Central Processing Unit (CPU).

A Tensor is an object in a vector space in mathematics, defining multi-directional relations among a set of objects. In the context of Machine Learning and PyTorch, Tensors are data containers that can accommodate multi-dimensional data. It is possible to embody a one-dimensional tensor as a vector, a two-dimensional tensor as a matrix.

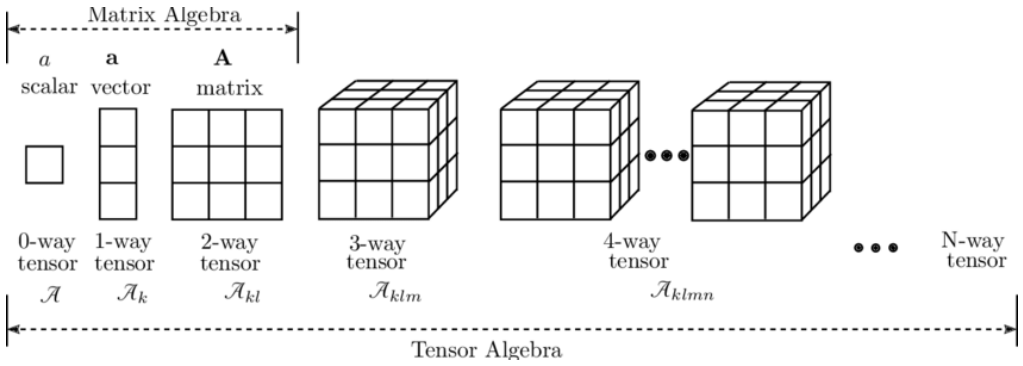


Figure 3.26: Illustrations of different shaped Tensors, reproduced from [23]

PyTorch Lightning is a lightweight PyTorch wrapper that provides a high-level API for PyTorch to build and deploy Machine Learning models quickly. It was started being developed to solve the problems of the PyTorch Community who were having problems while, e.g., multi-GPU training, TPU (Tensor Processing Unit) training, 16-bit precision.

PyTorch Lightning scales the models better to run on any processing units from CPUs to TPUs without changing the model structure. It decouples the research code from the core code, therefore allows us to create more readable codes and more reproducible models. In addition, it automates the training and optimization processes and provides seamless integration with popular visualizing and logging frameworks. In this thesis study, TensorBoard is used for logging and visualizing purposes.

3.8 Implementation of the Model

This section is devoted to explaining the core components of the model creation and training processes. Once having described the process at a high level, each component in the high-level scheme is explained in the following sub-sections.

3.8.1 High Level Design of the Model

Figure 3.27 illustrates the high level design of the model.

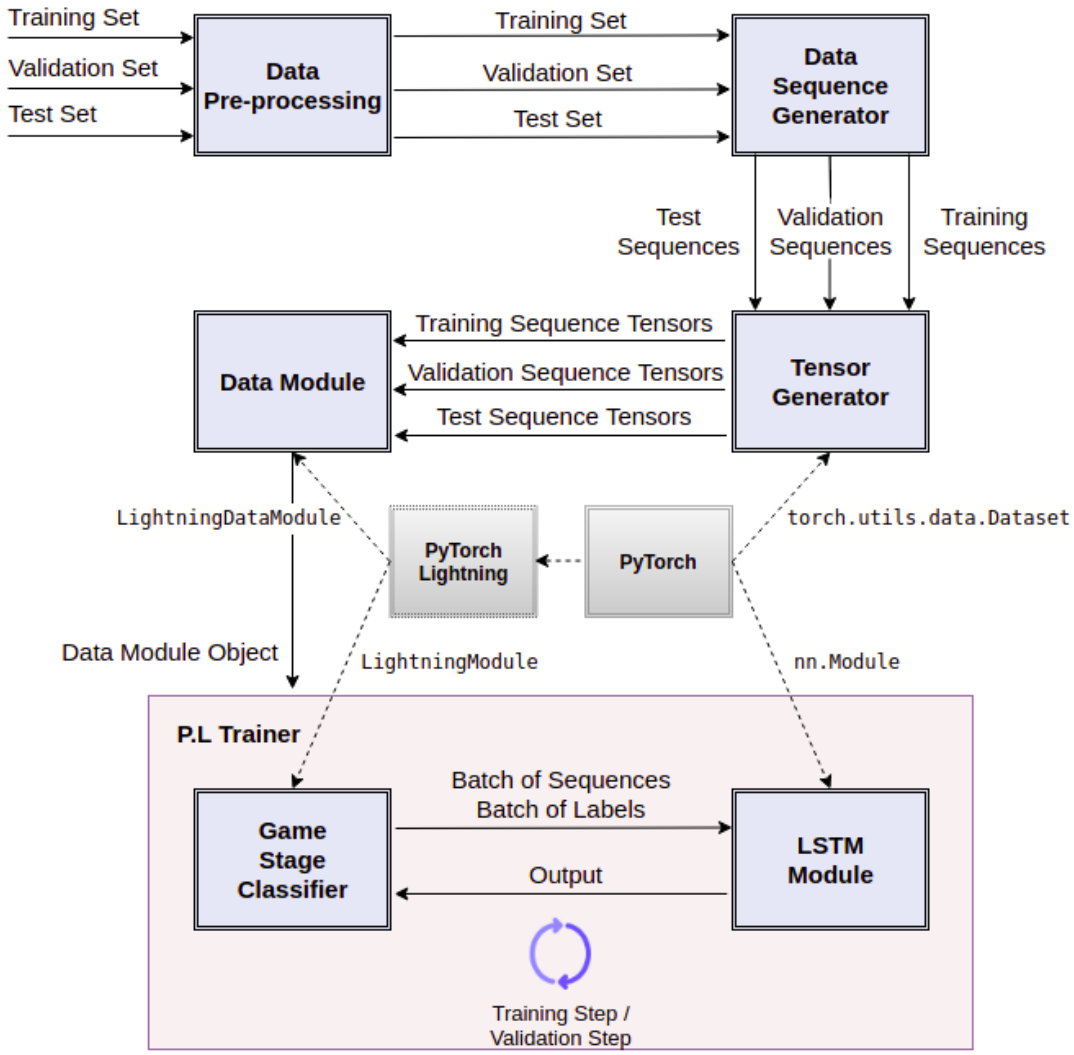


Figure 3.27: An illustrations of the model at high level

The dataset is partitioned into training, validation, and test sets at the beginning of the process. The splits are passed through the pre-processing operation. The resulting sets are forwarded into *Data Sequence Generator* module to create training, validation, and test sequences.

PyTorch and PyTorch Lightning operate on data structures named Tensors, as discussed in Section PyTorch and PyTorch Lightning. Therefore, the sequences are converted into Tensors. The training, validation, and test sequence Tensors are passed to *Data Module* to create the train, validation, and test Data Loaders to be used in the training phase. These Data Loaders are packed into a Data Module object to be forwarded to the PyTorch Lightning *Trainer* module.

The *Trainer* module of PyTorch Lightning is a wrapper class that interacts with the core PyTorch to automate the training process. The *Game Stage Classifier* module first initiates an LSTM (Long Short-term Memory) object, and in each iteration, it forwards a batch of data sequences and receives a corresponding output back. These iterations form the training and test steps.

3.8.2 Data Partitioning

The dataset is generated after the set of operations in Section Gameplay Dataset Generation. Based on the analysis results, Smoothing Window Length w is selected as 18. There is no big difference in results between a quarter-second window and a half-second window, but using a one-second window causes some information loss. The dataset is read by using the *read_csv* method of Pandas (2020) [24]. Pandas is an open-source data analysis library developed for Python.

Figure 3.28 illustrates the game stage distribution of the dataset.

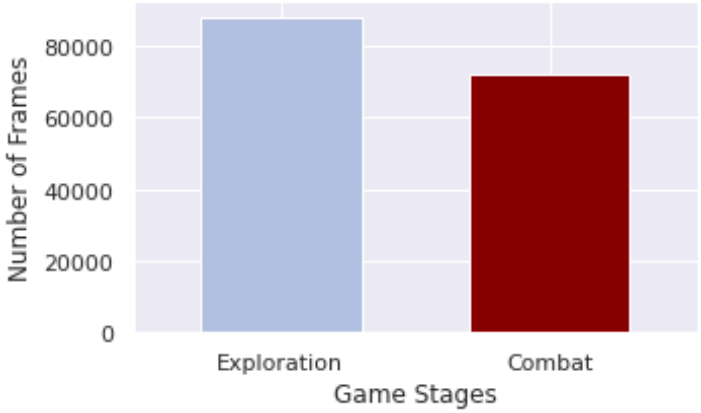


Figure 3.28: Game stage distribution of the dataset

Since there is no significant difference between the number of frames of each game stage, the frame distribution over the exploration and the combat stages is balanced. There is no need to do oversampling to prevent bias.

In order to use the dataset to train the Deep Learning model, it should be partitioned into training, validation, and test sets. Although the creation of a validation set is not a mandatory step, it is considered a good practice.

train_test_split method of Scikit-learn (2013) [25] is used to split the dataset. Scikit-learn is an open-source Machine Learning library for predictive data analysis, with seamless integration with Python.

The following code block is an illustration of how the dataset is read and partitioned into training, validation, and test sets.

```
import numpy  as np
import pandas as pd

data = pd.read_csv('dataset.csv')

X_train, y_train = data[['bitrate']], data[['gamestage']]

X_train, X_test, y_train, y_test = train_test_split(X_train, y_train,
                                                    test_size=0.2,
                                                    shuffle=False)
X_train, X_val,  y_train, y_val  = train_test_split(X_train, y_train,
                                                    test_size=0.2,
                                                    shuffle=False)
```

The parameter *test_size* is set to 0.2 after each call. This setting leads to obtaining the test set as 20% of total data, and while the remaining 80% of the data goes into the training set. The validation set receives the 20% of the training set.

Disabling the shuffling prevents the data from being shuffled while getting split. It is not a good idea to shuffle time-series data because it cause breaking the temporal structure of the data.

Figure 3.29 illustrates the data distribution over training, validation and test splits after partitioning.

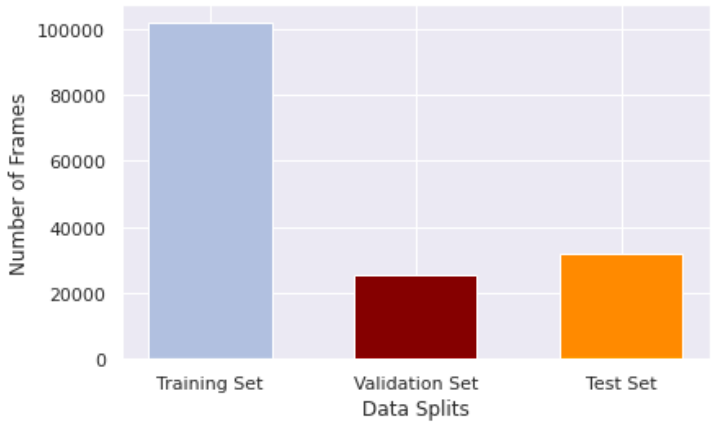


Figure 3.29: An illustration of data distribution after partitioning

3.8.3 Data Pre-processing

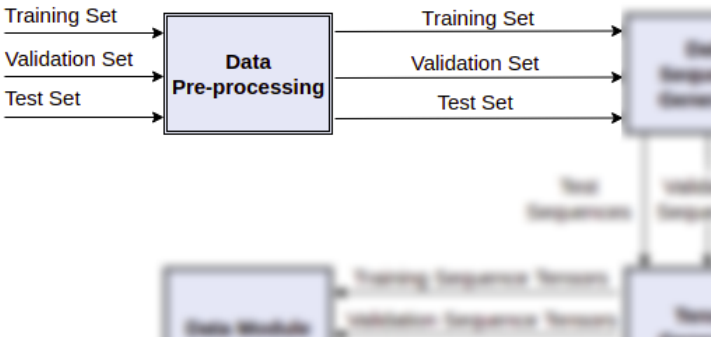


Figure 3.30: The data pre-processing snippet of high level design

The data pre-processing in this section should not be confused with the data pre-processing discussed in Section Gameplay Dataset Pre-processing. The former pre-processing step is conducted during the dataset generation. The data pre-processing operation discussed in this section is to prepare the dataset to be used in the training phase of the Deep Learning model. The technique used in this thesis study is Feature Scaling.

In a significant amount of datasets used in Machine Learning applications, the values in data fluctuate substantially. For example, in a dataset of a supermarket inventory, the values in the "price" feature vary between 100 and 100.000, while the values in the "quantity" feature vary between 1 and 100. In such cases, objective functions in Machine Learning algorithms may not perform well enough. For example, algorithms of some classifiers measure the distance between data samples. If one of the features has a significantly greater range of values as in the example, this feature may suppress the other one, which would result in the model biased towards the dominating feature.

In a case of a Deep Learning application, as it is in this case, randomly initialized weights in a network may struggle to cope with the scale of features, and this may cause activation functions to saturate. Feature Scaling is also vital for appropriate *Regularization* (for the penalization of coefficients) and for fastening the convergence of *Gradient Descents*.

In order to prevent such problems, the data splits are scaled by using the technique named Standardization. Standardization, also called *Z-score Normalization*, is a technique to give each data sample zero mean and unit variance.

$$x' = \frac{x - \mu}{\sigma} \tag{3.8}$$

where x is a data sample of the feature, x' is the scaled value of the data sample, μ is the Mean of the feature and σ is the Standard Deviation of the feature.

StandardScaler method of Scikit-learn exactly performs the technique depicted in Equation (3.8). The critical detail to note here is that only the training set is fit to the *scaler* object in order to avoid data leak to test and validation sets.

```
from   sklearn.preprocessing import StandardScaler
import pandas as pd

scaler = StandardScaler()
scaler = scaler.fit(X_train)

X_train_scaled = pd.DataFrame(data    = scaler.transform(X_train),
                              index   = X_train.index,
                              columns = X_train.columns)
X_val_scaled   = pd.DataFrame(data    = scaler.transform(X_val),
                              index   = X_val.index,
                              columns = X_val.columns)
X_test_scaled  = pd.DataFrame(data    = scaler.transform(X_test),
                              index   = X_test.index,
                              columns = X_test.columns)
```

3.8.4 Data Sequence Generation

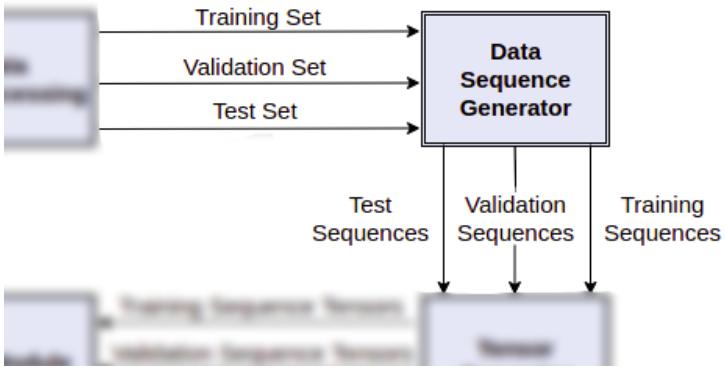


Figure 3.31: The sequence creation snippet of the high level design

In order to feed a PyTorch model with time-series data, data should be passed to the model sequence by sequence.

Pseudocode 7 illustrates the logic behind the data sequence creation.

Pseudocode 7: Function Definition for Sequence Creation

```
1: function createSequences(featureData, targetData, sequenceLength):
2:     Input  : featureData: DataFrame,
3:              targetData : DataFrame,
4:              sequenceLength: Integer
5:     Output: sequences: list()
6:     sequences ← list()
7:     i ← 0
8:     while i < len(featureData) - sequenceLength do
9:         seq   ← featureData[i : i + sequenceLength]
10:        label ← targetData[i + sequenceLength]['gamestage']
11:        sequences ← sequences + (seq, label)
12:        i ← i + sequenceLength
13:    end while
14: return sequences
15: end function
```

The important point to note in Pseudocode 7 is that all sequences should have the same length to be compatible with PyTorch. Therefore, line 6 ensures that the residual data is excluded. Another option to achieve the same purpose would be padding, but it is unnecessary to use the residual data.

As the design criteria, sequence length is chosen as 5, and a label of any sequence is the label of the data sample that comes right after the last element of the sequence.

After creating the function *createSequences* as in Pseudocode 7, train, validation, and test sequences are created by using the scaled data as follows:

```
trainSequences      = createSequences(X_train_scaled, y_train, SEQUENCE_LENGTH)
validationSequences = createSequences(X_val_scaled,   y_val,   SEQUENCE_LENGTH)
testSequences       = createSequences(X_test_scaled,  y_test,  SEQUENCE_LENGTH)
```

3.8.5 Creation of the Model Core

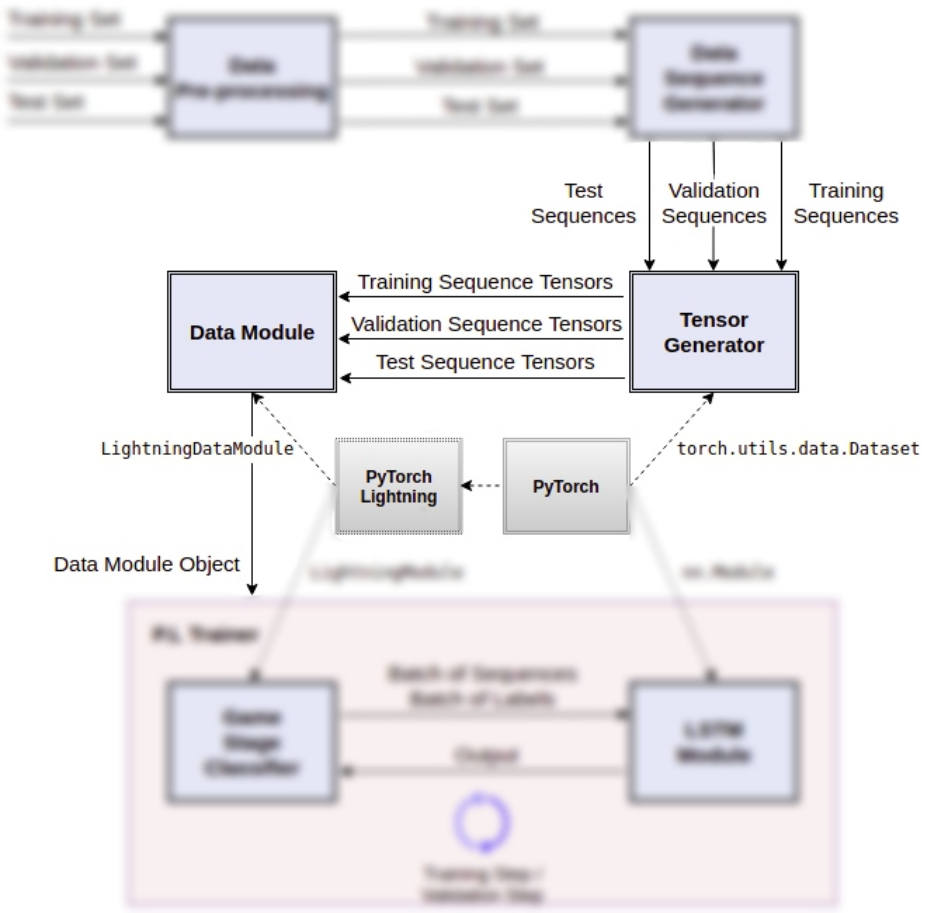


Figure 3.32: The Data Module object creation snippet of the high level design

Once having created the training, validation, test data sequences, it is essential to discuss the Data Module.

The Data Module is a class written for this model that encapsulates the required steps to process the data. It inherits PyTorch Lightning *LightningDataModule*. In the most basic sense, it receives the generated training, validation, and test data sequences and outputs the corresponding Data Module object to be used in the training phase. The following part explains this process step by step.

First of all, the Data Module object is initiated as follows, with a batch size equals 64:

```
dataModule = DataModule(trainSequences, validationSequences,
                        testSequences , batch_size=BATCH_SIZE)
```

In Section PyTorch and PyTorch Lightning, it is stated that PyTorch (and also other well-known frameworks such as TensorFlow) operates on data structures named Tensors. Therefore, PyTorch expects to receive the dataset in the form of a Tensor dataset.

Once the Data Module object having initiated, the class calls the Tensor Generator Class that is inherited from PyTorch *Dataset* Class. The duty of this generator class is to convert each sequence-label pair to the form of tensor sequences and passing them back to the Data Module. When the Data Module receives those tensor sequences, it creates the training, validation, and test tensor datasets from the relevant sequences. Finally, these tensor datasets are used to create the corresponding Data Loaders as follows.

```
def train_dataloader(self):
    return DataLoader(self.trainDataset,
                      batch_size=self.batchSize,
                      shuffle=False,
                      num_workers=cpu_count())

def val_dataloader(self):
    return DataLoader(self.valDataset,
                      batch_size=self.batchSize,
                      shuffle=False,
                      num_workers=cpu_count())

def test_dataloader(self):
    return DataLoader(self.testDataset,
                      batch_size=self.batchSize,
                      shuffle=False,
                      num_workers=cpu_count())
```

Data Loaders wrap the datasets to provide easy iteration over them. As a result, the initiated Data Module object that encapsulates the training, validation, and test Data Loaders is used during the model training.

3.8.6 Training Phase of the Model

After the creation of the Data Module object, the Game Stage Classifier module is created for the interactions with the core of the network (LSTM Module) through PyTorch Lightning *Trainer* API during the training.

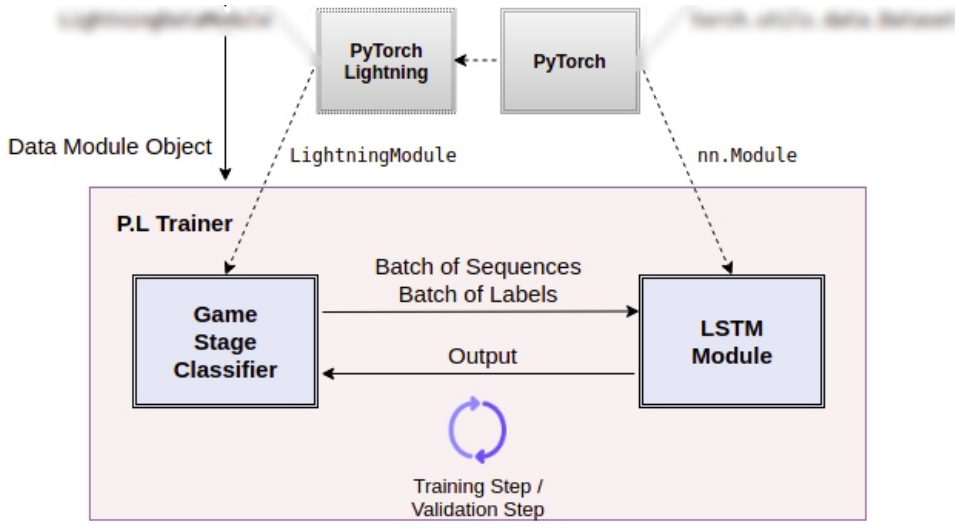


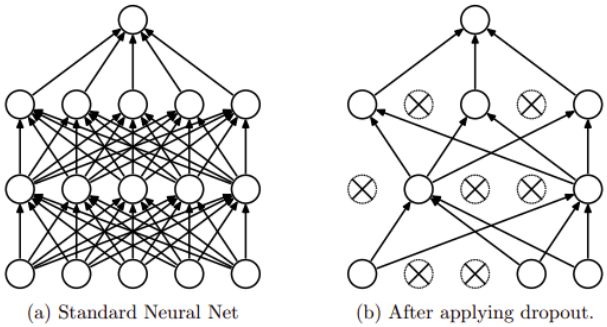
Figure 3.33: The training phase of the high level design

During the construction of Game Stage Classifier (inherited from PyTorch Lightning *LightningModule*), the LSTM Module (inherited from PyTorch *nn.Module*) is created based on the features of the dataset. Referring to Figures 3.21 and 3.25, the number of cells in each hidden layer and the number of layers are assigned 256, 3, respectively.

One essential technique worth mentioning here is the Dropout Regularization.

Dropout Regularization, also known as *Dilution*, is an important regularization technique, introduced by Hinton et al. (2014) [26], in Artificial Neural Networks to prevent model overfitting and proposes the idea of randomly and temporarily ignoring "dropping-out" some of the network nodes during the training.

According to this technique, randomly selected nodes are dropped out in each *forward* call during the training phase.



(a) Standard Neural Net (b) After applying dropout.

Figure 3.34: An illustration of Dropout Regularization, reproduced from [26]

Created LSTM Module brings Dropout Layers on top of each LSTM Layer output excluding the last layer, with dropout probability equal to 0.75.

The output of an LSTM node is linked to a PyTorch layer named *Linear*. This class is a linear unit that applies a linear transformation to received data.

After completing the creation of the LSTM network, Game Stage Classifier Node defines a Loss Function inside its constructor. Loss Function, also known as *Error Function*, is a mathematical term that maps decisions to their related costs. In optimization problems, the goal is to minimize the Loss Function. Since Machine Learning has been all about optimization, Loss Functions measure how wrong the predictions are, i.e., how far an estimated prediction from the actual value in each prediction. Based on this distance, the model optimizes itself iteratively until reaching the minimum distance and the minimum possible loss.

The Loss Function used in this model is the PyTorch *CrossEntropyLoss* which combines the Softmax and the Negative Log-Likelihood Losses. This Loss Function is ideal in multi-class classification tasks, but in the case of binary classification tasks, Cross-Entropy is calculated as the average Cross-Entropy across all data. Although the current dataset consists of two classes, Cross-Entropy Loss still works well enough, and it is also suitable for datasets with more than two game stages.

Another core component of the model is the optimizer. Optimizer is a mechanism that acts based on the outcome of the Loss Function to optimize the model by adjusting the network weights to make the model closer to the minimum loss possible. Due to its computational efficiency, the optimizer used in this model is the Adam Optimizer, introduced by Kingma & Ba (2017) [27].

The creation of the Game Stage Classifier and the LSTM Modules concludes the construction of the network core. There is still a need to set some components for the training phase. The first component discussed here is the Model Checkpoint. *ModelCheckpoint* is a PyTorch Lightning class that saves the model periodically based on the defined metrics.

```
callbackCheckpoint = ModelCheckpoint(
    dirpath  = "checkpoints",
    filename = "best_checkpoint",
    verbose  = True,
    save_top_k = 1,
    monitor  = "val_loss",
    mode     = "min")
```

According to the checkpoint defined in the code block, the model monitors the Validation Loss and seeks the minimum Validation Loss value. If the model reaches a lower Validation Loss value after an iteration, it overrides the existing model parameters with the current ones.

save_top_k defines the number of best models to be saved, and this configuration ensures to save only the best model.

The second concept to be discussed here is Early Stopping. Early Stopping is another metric monitoring mechanism that terminates the training early if the model stops improving, i.e., settles on a plateau. *EarlyStopping* is a PyTorch Lightning class that monitors the training on the defined metrics and terminates the training if it decides that the model is not improving anymore.

```
callbackEarlyStopping = EarlyStopping(
    monitor="val_loss",
    patience=EARLY_STOP_AFTER_EPOCHS,
    min_delta=EARLY_STOP_MIN_DIFFERENCE)
```

According to the Early Stopping defined above, the model monitors the Validation Loss in each iteration. In the default configuration, *patience* defines the number of iterations without any improvement, after which the training phase will be terminated. This parameter is set to *EARLY_STOP_AFTER_EPOCHS* which is defined as 5. *min_delta* defines the minimum change in the monitored metric that would be accepted as an improvement. This parameter is set to *EARLY_STOP_MIN_DIFFERENCE* which is defined as 0.005. In this particular setting, if the change of Validation Loss is below 0.005, that change is not qualified as an improvement.

The last essential component for the training phase is the PyTorch Lightning *Trainer* that automates the training phase by using the Data Module object together with the Game Stage Classifier.

```
trainer = pl.Trainer(
    logger = TensorBoardLogger("logs", name="Doom_Experiment"),
    checkpoint_callback = callbackCheckpoint,
    callbacks = [callbackEarlyStopping],
    max_epochs = NUM_EPOCHS,
    gpus = 1,
    progress_bar_refresh_rate = 30)
```

TensorBoardLogger is used to monitor the model performance, and defining a logger enables to log the process throughout the training. The defined checkpoint and Early Stopping callbacks are introduced to the Trainer. *max_epochs* defines the number of training epochs after which the training phase will be terminated, and it is set to *NUM_EPOCHS* which is defined as 100.

As discussed at the beginning of Section Training Phase of the Model, Game Stage Classifier Module is created as below.

```
model = GamestageClassifier(n_features=NUM_FEATURES,
                            n_classes =NUM_CLASSES)
```

The way of the Data Module object creation has already been explained in Section Creation of the Model Core. After these essential steps, the training phase starts with fitting the model to the Data Module through the *Trainer* API.

```
trainer.fit(model=model,
            datamodule=dataModule)
```

At the beginning of the training phase, the *Trainer* constructs all the modules discussed in Sections Creation of the Model Core and Training Phase of the Model.

The Game Stage Classifier fetches a batch of sequences and a corresponding batch of labels from the Data Module in each iteration. It forwards the batches to the LSTM Module.

LSTM Module first flattens the parameters and then pushes the data into the LSTM network. The output of the last layer of the LSTM network passes through the final layer, which is the Linear unit. The LSTM Module returns the outcome of the Linear unit.

When the Game Stage Classifier receives the outcome of the Linear Unit, it calls the Loss Function, which is the *CrossEntropyLoss* in this case. The Loss Function evaluates the received outcome (predicted classes) by comparing them with the actual labels. As a result of this evaluation, Game Stage Classifier calculates the loss value. Finally, the optimizer updates the network weights according to the calculated loss value to achieve the minimum loss.

At the end of each training step, the model runs a validation step to observe how the model behaves on unseen data, which is the validation data. Based on this performance, the model is updated, and the following epoch is run.

According to the model performance in each epoch, the created Checkpoint Callback compares the Validation Loss of the current epoch with the minimum one achieved so far. In case of achieving a lower Validation Loss than the last saved model's Validation Loss, this callback saves the model as the best model so far.

In the same sense, the created Early Stopping Callback compares the current Validation Loss with the former one, and if the difference is lower than the defined threshold, it marks the current epoch as an "epoch without an improvement". If the number of adjacent epochs without an improvement reaches a certain threshold, the model's training process is terminated by the callback.

3.8.7 Test Phase of the Model

Creating the testing module is way more straightforward than the creation of the training process.

It is stated in the training phase that a set of parameters that provide the best model performance is saved as a checkpoint in each iteration. Therefore, before starting to test phase, first, the best version of the model, in terms of Validation Loss, is retrieved from the saved checkpoints.

```
trainedModel = GamestageClassifier.load_from_checkpoint(
    checkpoint_path = trainer.checkpoint_callback.best_model_path,
    n_features = NUM_FEATURES,
    n_classes  = NUM_CLASSES)

trainedModel.freeze()
```

The loaded model is frozen because it is used only for inference. Freezing disables the Dropout and gradient calculations.

The test dataset to be used for inference is already prepared and wrapped by the Data Module object. Each item of the test dataset keeps a sequence-label pair in the form of Tensors. When the inference starts, it iterates over each item in the test dataset, and in each iteration, the item is split into sequence and label. The sequence is forwarded to the loaded model, and the class prediction according to the input sequence is inserted into a data container. In the same sense, the actual label is inserted into a data container. Once having iterated over all the items, the actual and the predicted classes are compared to calculate the model performance.

Pseudocode 8 illustrates the mechanism behind the function that runs a test on a test dataset.

Pseudocode 8: Definition of the Test Function

```
function runTest(trainedModel, testDataset):
    Input  : trainedModel    : GamestageClassifier→model
             testDataset     : dataModule→testDataset
    Output: confusionMatrix: sklearn.metrics→confusion_matrix
    predictions ← list()
    labels      ← list()
    foreach item in testDataset do
        sequence   ← item['sequence']
        label ← item['label']
        _ , output ← trainedModel(sequence)
        prediction ← torch.argmax(output, dim = 1)
        predictions ← predictions + prediction.item()
        labels ← labels + label.item()
    end foreach
    confusionMatrix ← confusion_matrix(labels, predictions)
return confusionMatrix
end function
```

Confusion Matrix, also known as Error Matrix, is a useful performance measurement technique in Machine Learning to measure model accuracy. The axes of a Confusion Matrix represent actual and predicted target classes. Instead of observing only the ratio of correctly classified data samples, a Confusion Matrix allows us to observe how the predictions disperse over different target classes.

The evaluation of the inference results is analyzed in the following chapter.

Chapter 4

Evaluation of the Results

The final chapter is devoted to analyzing the outcomes of different experiments. Since the structure of the test process has already been discussed, this chapter solely focused on the results of the experiments. All of the experiment results are evaluated by using confusion matrices.

4.1 Experiments with Vanilla Machine Learning Methods

The first experiment is conducted to evaluate the dataset with the traditional Machine Learning methods. The theoretical backgrounds or model architectures of the methods used in this section are not examined in detail to not detract from the main focus of the study.

Even though the created model is a Deep Learning model, or the dataset consists of time-series data, this experiment is carried out only to have a starting point, a reference point, and to see what would be the outcome in case of applying some of the "Vanilla" Machine Learning Methods to the dataset.

The first algorithm implemented in this section is K-means Clustering. K-means Clustering is an Unsupervised Learning algorithm that aims at partitioning data into k-clusters. Each data sample is placed into a cluster whose center is the closest to it among all clusters. Since it is an Unsupervised Learning algorithm, K-means Clustering does not aware of any label in its nature.

The created model for this algorithm processes only the bitrate data, and it is not aware of any game stage. The metric used by the model is the distance between the data samples, in the most basic sense. The result of the K-means Clustering algorithm is evaluated in two aspects: Silhouette Score and Accuracy Score.

Silhouette Value for each data sample in the dataset refers to how similar a data sample is to the cluster it was placed in compared to other clusters. The value varies between -1 and 1, where the higher values represent a better placement and the lower values represent otherwise. Silhouette Score is computed by taking the average of each Silhouette Value corresponding to each data sample.

The result of the experiment indicates a Silhouette Score equals 0.51. This result indicates that there is a poor performance while placing the data samples into the correct clusters.

If the predicted game stages, which are two clusters in this case, and the actual game stages are compared, the result of this experiment is as follows.

Figure 4.1: The experiment results of the K-means Clustering algorithm

According to the confusion matrix above, the K-means Clustering algorithm can not provide adequate performance. The figure indicates that both data samples belonging to the exploration and the combat game stages are correctly classified by almost 48%, which basically means a random classification. The result can be considered as consistent with the Silhouette Score and as pretty weak as expected, since the temporal relation between the data samples has been broken.

The second algorithm implemented in this section is K-nearest Neighbors. Unlike the K-means Clustering, it is a Supervised Learning algorithm, which means that the model learns based on target classes. K-nearest Neighbors algorithm is a Supervised Learning algorithm that can be used both for classification and regression tasks and proposes the idea that similar things exist close to each other.

In its simplest version in a classification task as in this thesis study, when a K-nearest Neighbors algorithm seeks to classify a data sample, the k closest data

samples to the sample to be classified are fetched, and the majority label of those closest data samples is assigned to the sample to be classified.

Comparison of the predicted game stages and the actual game stages indicates the following result in Figure 4.2.

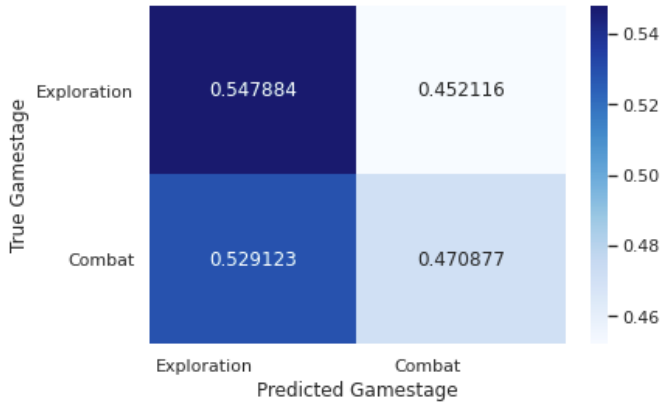


Figure 4.2: The experiment results of the K-nearest Neighbors algorithm

Although the classification accuracy of the exploration stages is higher than the case in the K-means Clustering algorithm, the K-nearest Neighbors algorithm still lacks the correct classification ability. The model can barely achieve 50% accuracy, considering the True Positives and True Negatives are 54% and 47%, respectively.

The last algorithm implemented for this experiment is the Support Vector Machines. Support Vector Machines is another type of Supervised Learning algorithm that can be used both for classification and regression tasks. In a data space of any dimension, the Support Vector Machines algorithm aims at finding the hyperplane with the largest possible margin that separates the closest samples of each data class for the lowest generalization error. Since the algorithm is extensively heavyweight for large datasets as such, only a portion of the dataset is used for this experiment, and the result of the experiment is as follows.

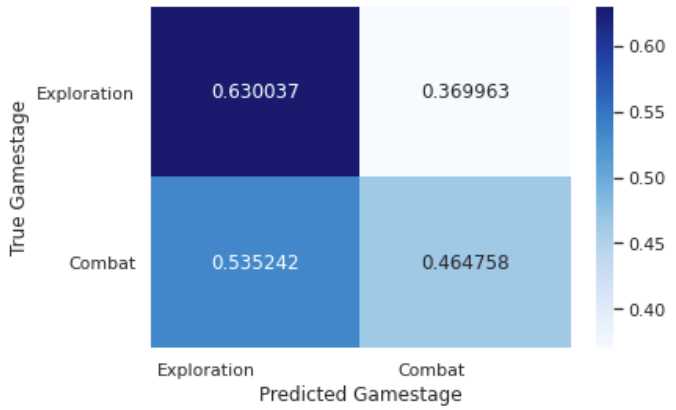


Figure 4.3: The experiment results of the Support Vector Machines algorithm

Support Vector Machines can be considered a more robust algorithm than the former methods due to the flexibility of using complex kernels. As a result of this power, the model can classify the exploration stages with 63% accuracy, which is apparently higher than the former algorithms, yet it is still highly insufficient in the classification of the combat stages. Therefore, the overall performance is still pretty low. Besides, it should be noted that due to the kernel complexity, it is almost impossible to apply this model to the actual dataset because of its size.

The experiments above are held using the Grid Search technique to tune hyperparameters to seek the best performing hyperparameter configurations. Nevertheless, the performance of the models cannot exceed a certain threshold besides the heavyweight workloads of the algorithms.

The reason behind this problem is frankly quite obvious: the bitrate data samples are meaningful only when they are evaluated as a sequence. Clustering algorithms or traditional Supervised Learning algorithms treat data as a set of singular data points, which breaks the temporal relation between the samples. Hence, models cannot achieve decent performance.

4.2 Experiments with the Created Model

In this section, the test results of the main Deep Learning model are evaluated.

The hyperparameters in the created Deep Learning Model are explained throughout the previous chapter in detail, but it might be helpful to summarize the decisions. As stated in the previous chapter, analyses of the dataset indicated that smoothing with a time window of half-second is the most suitable option. Therefore, the dataset is smoothed with a half-second time window for the experiment.

The batch size is defined as 64; hence the training process is accelerated. Although the skeleton of the model remains the same, the following results are obtained in different network complexities.

The first experiment is orchestrated to see whether the model can bring an acceptable performance in a reasonably basic network architecture. This network consists of 32 LSTM nodes in a single layer. The figure below is automatically generated by the TensorBoard and illustrated the training and validation Learning Curves of a model with a single layer that accommodates 32 LSTM nodes over a varying number of frames. The graphs provide insight into the training and validation of the model.

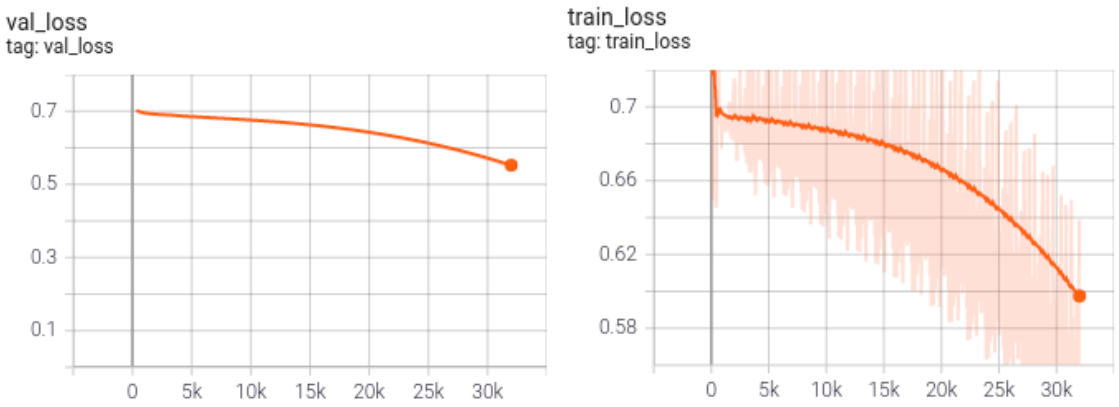


Figure 4.4: Learning Curves of the model with 32 LSTM nodes on 1 layer

Both curves decrease as the number of processed frames increases. The decrease of the loss is an expected fact since as long as a model is trained, it makes fewer mistakes. Therefore, the loss decreases. However, these curves indicate that the model cannot learn because even after 30k frames, the loss cannot decrease below 60%. Hence, the model can barely achieve 40% classification accuracy. The shadow behind the curve of train loss is not important. TensorBoard automatically smoothes the graph to get rid of the noise introduced during the training phase.

The second experiment increased the model complexity to see the minimum required complexity to achieve decent performance. This network consists of 64 LSTM nodes in a single layer.

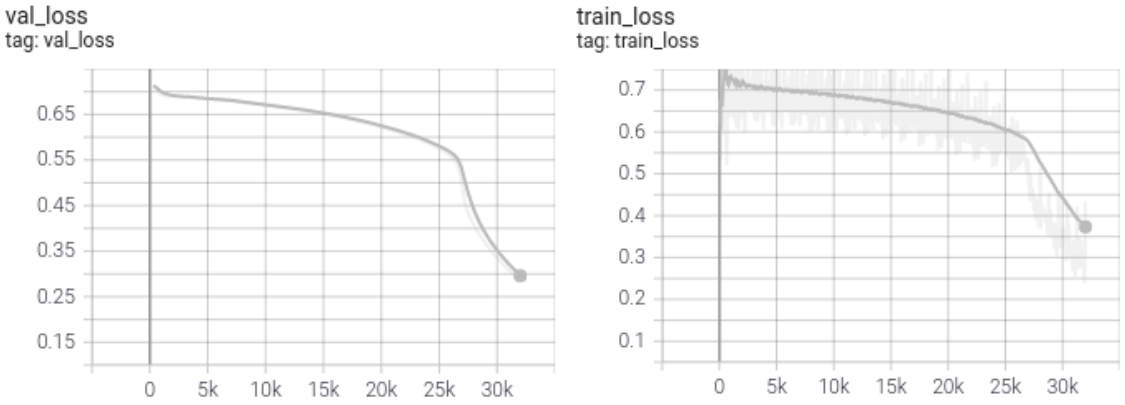


Figure 4.5: Learning Curves of the model with 64 LSTM nodes on 1 layer

According to the figures above, increasing the model complexity improved the model performance significantly. The training and validation losses decreased over time simultaneously, as expected, and after 30k frames, they reached approximately 30% loss, which refers to an acceptable model accuracy.

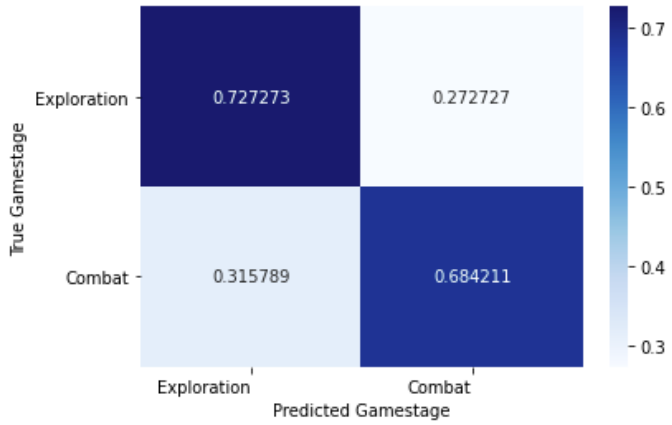


Figure 4.6: Confusion Matrix of the model with 64 LSTM nodes on 1 layer

The Confusion Matrix indicates the model performance on the unseen data, which is the test dataset. Based on the values, one can say that the model accuracy is approximately 70%, and the model is able to classify the exploration game stages slightly more accurately than the combat stages. This network configuration can be accepted as a threshold to achieve an adequate model performance.

The third experiment adopted an architecture with 128 LSTM nodes on double layers, which can be assumed as a significant change compared to the former architectures.

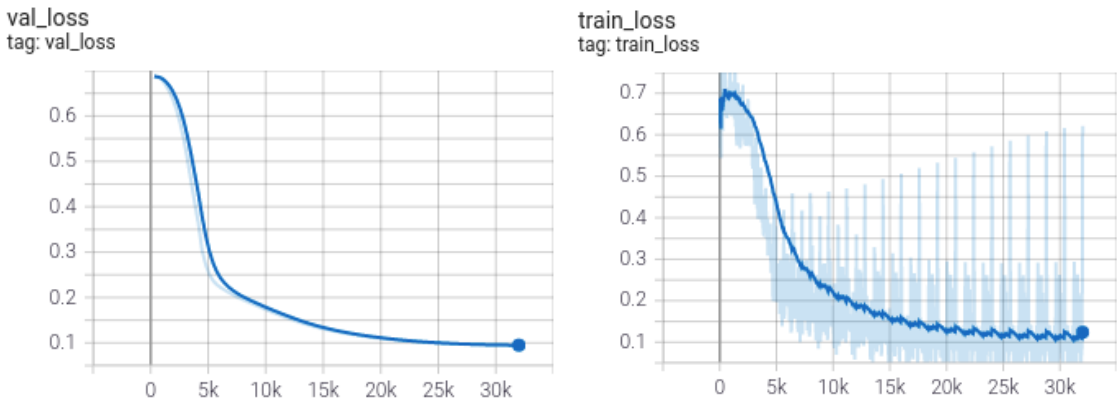


Figure 4.7: Learning Curves of the model with 128 LSTM nodes on 2 layers

Figure 4.7 presents the learning curves that are dramatically better than the curves in the former architectures. The significant change in the model architecture brought significant change to the learning phase. The decrease in both validation and training curves indicates that the model did not face an overfitting problem because, in such a case, the validation loss usually remains stable in a high value as the training loss decreases. Besides the excellent fit, the validation loss indicates that the model reaches approximately 90% accuracy in 20k frames, which is way shorter than the former cases.

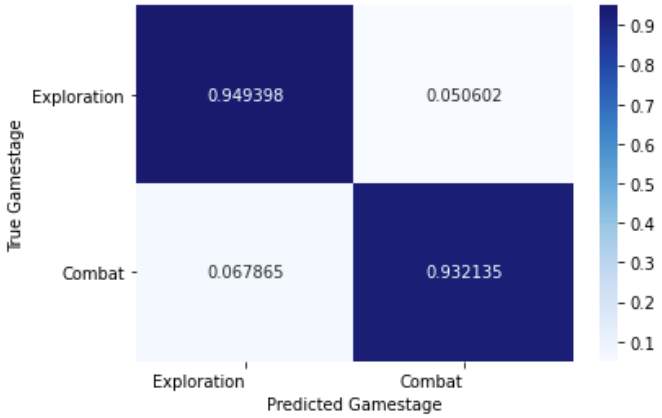


Figure 4.8: Confusion Matrix of the model with 128 LSTM nodes on 2 layers

According to the Confusion Matrix above, it is possible to say that the model can reach an impressive accuracy on the test dataset for both game stages.

The following experiment is held in a network architecture of 256 LSTM nodes in triple layers. The figures below illustrate the validation and training losses.

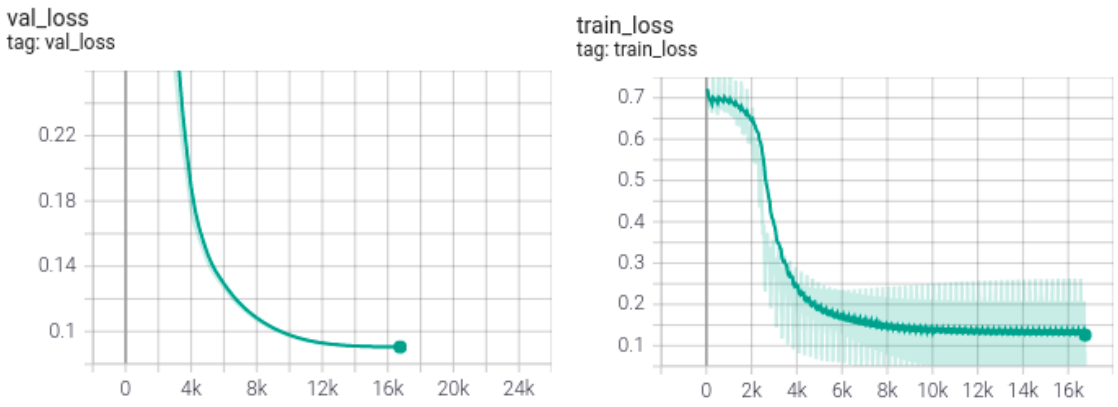


Figure 4.9: Learning Curves of the model with 256 LSTM nodes on 3 layers

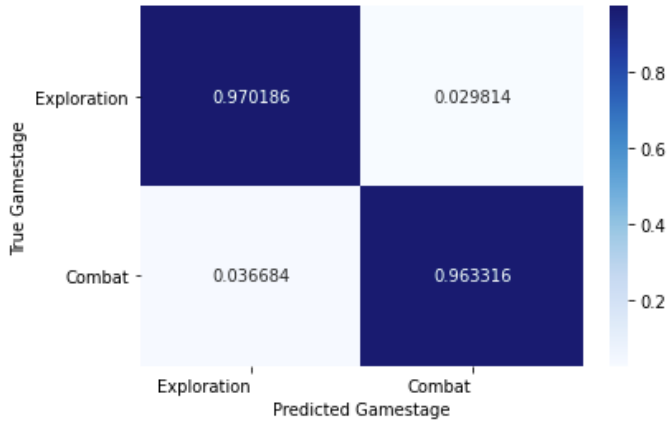


Figure 4.10: Confusion Matrix of the model with 256 LSTM nodes on 3 layers

According to Figure 4.9 and Confusion Matrix 4.10, one can say that the last model architecture is the best model architecture by far. Besides achieving an outstanding accuracy on the unseen data for both game stages, the model can reach 90% accuracy with approximately 10k frames and can converge after approximately 17k frames. The former architecture, where there are 128 LSTM nodes with two layers, can reach 90% accuracy after 20k frames and can converge after approximately 30k frames.

In the last two experiments, the accuracy of correctly classified exploration stages is slightly higher than the accuracy of correctly classified combat stages. The reason behind this would be a thin bias towards the exploration stages since the number of frames belonging to the exploration stage is higher than the number of frames belonging to the combat stage, as it is depicted in Figure 3.28.

Another experiment idea is testing the created model using a manually recorded dataset instead of using the dataset generated by the Deep Reinforcement Learning Agent. In order to implement this idea, manual control is activated by disabling the agent, and the exact frame-label pairs recording mechanism is implemented to the manual control.

The manually recorded dataset is significantly smaller than the previous experiments' dataset because playing the game manually while the frame-label pairs are being recorded in the background is quite memory-consuming. Figure 4.11 illustrates the game stage distribution of the manually recorded dataset. According to the figure, the distribution of the exploration and the combat game stages is approximately 60% to 40%, respectively. Therefore, it is possible to say that there is no steep difference between the number of stages that would cause a bias.

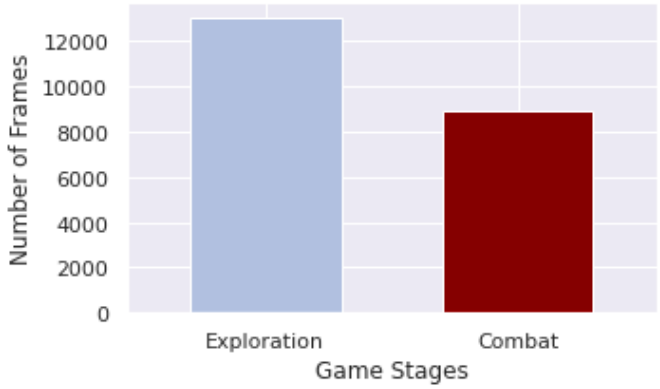


Figure 4.11: Game stage distribution of the manually recorded dataset

Figure 4.12 illustrates the result of the experiment conducted by using this manually recorded dataset. The result is quite similar to the previous results, yet there is a slight difference. Unlike the former experiments, the accuracy of correctly classified combat stages is slightly higher than the accuracy of correctly classified exploration stages in this outcome. Nevertheless, there is no critical difference between the accuracies to require thinking about.

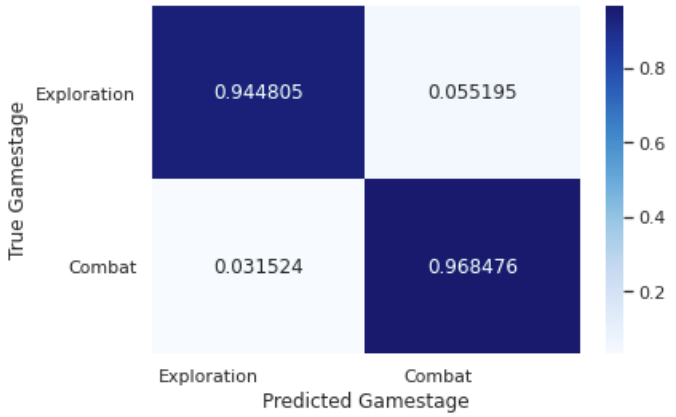


Figure 4.12: Confusion Matrix of the experiment with manually recorded dataset

Once having seen these impressive results, an idea arose to test the model's sanity with a corrupted dataset. For this purpose, the target column of the same dataset was iterated label by label, and labels are either inverted or left untapped randomly.

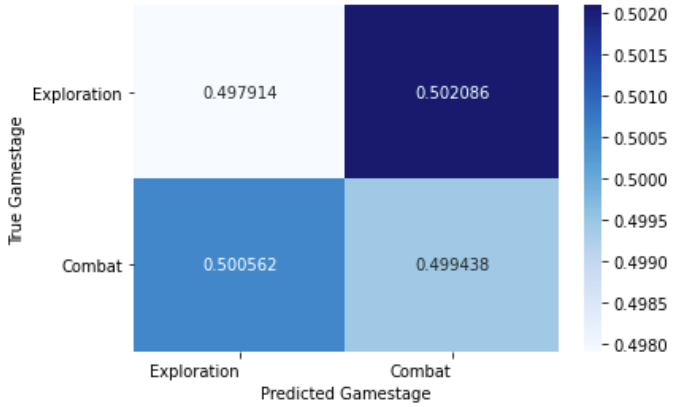


Figure 4.13: Confusion Matrix of the target corruption experiment

As shown in Figure 4.13, the result is entirely random as expected. The model fails in predicting the game stage. It is unable to deduct a pattern between the frames because of the label corruption.

The objective of the final experiment is to run cross-map tests to observe how does a model which is trained on a specific game map perform on another map.

In order to evaluate this idea, three different game maps are used in this experiment. Three different models are trained on these three maps, and each one of the models is tested on all maps. Table 4.1 indicates the experiment results.

Map ID		Map 1		Map 2		Map 3	
		Exp.	Combat	Exp.	Combat	Exp.	Combat
Map 1	Exp.	0.970	0.029	0.929	0.070	0.931	0.068
	Combat	0.036	0.963	0.058	0.941	0.072	0.927
Map 2	Exp.	0.934	0.065	0.978	0.021	0.922	0.077
	Combat	0.061	0.938	0.020	0.979	0.080	0.919
Map 3	Exp.	0.937	0.062	0.927	0.072	0.963	0.036
	Combat	0.080	0.919	0.063	0.936	0.031	0.968

Table 4.1: The experiment results of the cross-map testing

In this experiment, it is decided to indicate the values in a table instead of using Confusion Matrices because pacing nine different confusion matrices would take much place and would cause confusion.

In Table 4.1, a block identified by a row map identifier x with column map identifier y depicts a confusion matrix of an experiment where the model is trained on map x and tested on map y. As in the former confusion matrices, the column-wise game stages indicate the actual game stages, whereas the row-wise game stages indicate the predicted game stages.

According to the table, all the models trained on a particular map and tested on the same map achieved impressive test accuracies. The models trained and tested on different maps achieved lower accuracies than the models trained and tested on the same map, yet they performed well enough. This difference is reasonably expected.

Even though the impressive results on cross-map tests seem surprising, this can be considered reasonable. Doom is a very primitive game. Almost all the maps are pretty small and primitive. They are highly similar to each other, and the game flow on these different maps is almost the same. Besides, the agent and enemies have limited action space. All these facts together lead to similar bitrate fluctuations and hence to similar model performances. Most likely that in implementing the same or similar techniques to today's games, the performance of models would dramatically change.

Chapter 5

Conclusion

Due to the increasing interest in Cloud Gaming, the increasing complexity of games, and the inadequacy of current Cloud Gaming service providers, especially in competitive games such as the First Person Shooter (FPS) genre, the need for better in-game Quality of Experience (QoE) assessments are more than before. In order to achieve better in-game QoE, this thesis study investigates if there is a correlation between the stages of a game being streamed to a player and the stream's bitrate, and proposes the idea of taking the automated in-game QoE assessments one step further by creating a Deep Reinforcement Learning (DRL) agent capable of classifying game stages based on the instant bitrate values of the stream.

Based on the proposed methodology, a Deep Reinforcement Learning agent plays a game to generate a dataset. The generated dataset is analyzed in detail in terms of the idea's feasibility, and the analysis results indicate that there might be a correlation between the bitrate and the game stages. Based on this correlation, a Deep Learning model is created, trained, and tested extensively.

The test results show that it is possible to classify instant game stages with high accuracy by processing the corresponding instant bitrate values. Therefore, besides many in-game QoE optimization possibilities, Cloud Gaming service providers can achieve better resource allocation among their players by implementing this idea. Hence, it can be possible to conduct better in-game Quality-of-Experience assessments in Cloud Gaming.

One question that cannot be answered in this thesis study is how successful this methodology would be in a state-of-the-art complex game, as an old-school game had to be used due to some of the limitations discussed in detail in the study. This question will remain open until the method is tested in today's games.

Bibliography

[1] Kjell Brunnström, Katrien De Moor, Ann Dooms, Sebastian Egger-Lampl, Marie-Neige Garcia, Tobias Hossfeld, Satu Jumisko-Pyykkö, Christian Keimel, Chaker Larabi, Bob Lawlor, Patrick Le Callet, Sebastian Möller, Fernando Pereira, Manuela Pereira, Andrew Perkis, Antonio Pinheiro, Ulrich Reiter, Peter Reichl, Raimund Schatz, and Andrej Zgank. *Qualinet White Paper on Definitions of Quality of Experience*. 03 2013.

[2] Sarah Feldman. The Sky Is the Limit for Cloud Gaming. https://www.statista.com/chart/17501/cloud-gaming/, 2019.

[3] Asif Laghari, Hui He, Kamran Ali, Rashid Laghari, Imtiaz Halepoto, and Asiya Khan. Quality of Experience (qoe) in Cloud Gaming Models: A Review. *Multiagent and Grid Systems*, 15:289–304, 10 2019.

[4] Ivan Slivar, Mirko Suznjevic, Lea Skorin-Kapov, and Maja Matijasevic. Empirical qoe Study of In-home Streaming of Online Games. *Annual Workshop on Network and Systems Support for Games*, 2015, 01 2015.

[5] Michael Jarschel, Daniel Schlosser, Sven Scheuring, and Tobias Hossfeld. An Evaluation of QoE in Cloud Gaming Based on Subjective Tests. 06 2011.

[6] German Sviridov, Cedric Beliard, Andrea Bianco, Paolo Giaccone, and Dario Rossi. Removing Human Players From the Loop: Ai-assisted Assessment of Gaming qoe. pages 1160–1165, 07 2020.

[7] Wikipedia contributors. Doom (1993 video game) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Doom_(1993_video_game)&oldid=1028755673, 2021.

[8] Wikipedia contributors. Wolfenstein 3d — Wikipedia, the free encyclopedia, 2021. [Online; accessed 12-July-2021].

[9] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. Vizdoom Competitions: Playing Doom From Pixels. *IEEE Transactions on Games*, 2018.

[10] Guillaume Lample and Devendra Singh Chaplot. Playing FPS Games with Deep Reinforcement Learning. 2017.

[11] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. Visual Doom AI Competition (VDAIC). http://vizdoom.cs.put.edu.pl/, 2017.

[12] Stone Hausknecht. Deep Recurrent Q-learning for Partially Observable MDPs. *arXiv*, (1507.06527), 2015.

[13] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. Vizdoom. http://vizdoom.cs.put.edu.pl/, 2018.
[14] Suramya Tomar. Converting Video Formats with FFmpeg. *Linux Journal*, 2006(146):10, 2006.
[15] Wikipedia contributors. Video Compression Picture Types — Wikipedia, the Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Video_compression_picture_types&oldid=994295522, 2020.
[16] Charles Holt. Forecasting Seasonals and Trends by Exponential Weighted Moving Averages. *International Journal of Forecasting*, 20:5–10, 03 2004.
[17] W.I. King. *The Elements of Statistical Method*. Macmillan, 1920.
[18] Johannes Forkman. Estimator and Tests for Common Coefficients of Variation in Normal Distributions. *Commun. Stat. Theory Methods*, 38, 01 2009.
[19] Facundo Bre, Juan Gimenez, and Víctor Fachinotti. Prediction of Wind Pressure Coefficients on Building Surfaces Using Artificial Neural Networks. *Energy and Buildings*, 158, 11 2017.
[20] IBM. Recurrent Neural Networks. https://www.ibm.com/cloud/learn/recurrent-neural-networks, 2020.
[21] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural Computation*, 9:1735–80, 12 1997.
[22] Christopher Olah. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.
[23] Dmytro Shulga, Oleksii Morozov, Volker Roth, Felix Friedrich, and Patrick Hunziker. Tensor B-Spline Numerical Methods for PDEs: a High-Performance Alternative to FEM. 04 2019.
[24] The Pandas Development Team. pandas-dev/pandas: Pandas, February 2020.
[25] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API Design for Machine Learning Software: Experiences from the Scikit-learn Project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.
[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.
[27] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2017.