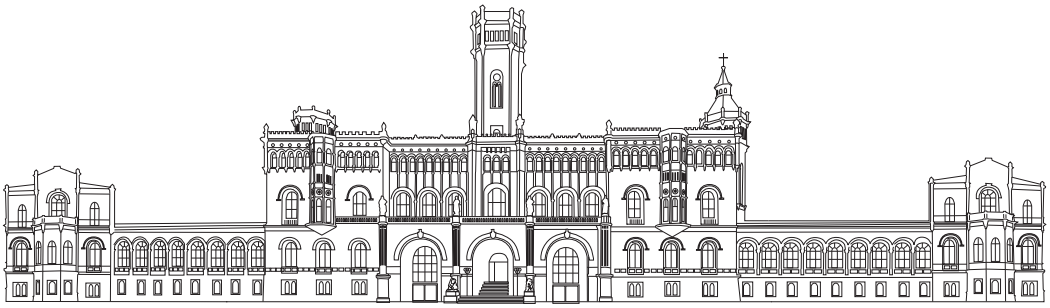

LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK
INSTITUT FÜR KOMMUNIKATIONSTECHNIK

Machine Learning Implementation on the Client-Side for Adaptive Video Streaming

Master Thesis

eingereicht von

BESHER KARKOUR

am 26. Dec 2019

Erstprüfer	:	Prof. Dr.-Ing. Markus Fidler
Zweitprüfer	:	Prof. Dr.-Ing. Jürgen Peissig
Betreuer	:	M.sc. Tilak Varisetty

EHRENWÖRTLICHE ERKLÄRUNG

Hiermit versichere ich, die vorliegende Master Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die wörtlich oder inhaltlich aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Hannover, den 26. Dec 2019

Besher Karkour

TABLE OF CONTENTS

FIGURES

TABLES

ACRONYMS

ABR Adaptive Bitrate

AI Artificial Intelligence

ANN Artificial Neural Network

API Application Programming Interface

CBR Constant Bitrate

CDN Content Delivery Network

CPU Central Processing Unit

DASH Dynamic Adaptive Streaming over HTTP

FPS Frames Per Second

FVV Free-Viewpoint Video

GPU Graphics Processing Unit

HLS HTTP Live Streaming

HS HTTP Streaming

ML Machine Learning

MSD Media Segment Duration

NSS Non-Segmented Streaming

PLR Packet Loss Rate

QoE Quality of Experience

RTT Round-Trip Time

SELU The Scaled Exponential Linear Unit

SFT Segment Fetch Time

SGD Stochastic Gradient Descent

SSH Secure Shell

TCP Transmission Control Protocol

VBR Variable Bitrate

WebGL Web Graphics Library

Teil I

TECHNICAL BACKGROUND

1 INTORDUTION

The demand for streaming multimedia is growing rapidly where 92% of internet users watch videos online in 2019 [19] with video traffic being the majority traffic on the internet. Since HTML5 introduced HTML Streaming it made playing videos in the browser much easier by removing the need for any additional plugins [23].
HTML5 makes it easy for web developers too and implementing video player to there webpages becomes as easy as adding any other HTML elements like texts or images.

At the same time, more users are joining the Internet every day. Those users have a wide variety of devices that makes video streaming more accessible. Most notably are PCs, smart televisions and cell phones. Especially the cell phone uses different generations of connections technology, which brings to attention the different data rates that users can obtain including WiFi, cable, DSL, and fiber to the home.

1.1 REAL TIME INTERACTIVE FREE-VIEWPOINT VIDEO

This work based on Free-Viewpoint Video (FVV) implemented by [20], which allows the user to interact with a video stream by changing the mouse position. This means the video content is directly controlled by the viewer.
The rendering application in the server uses open source software and open standards like FFmpeg to encode the video stream and libwebsockets, which is a lightweight C WebSocket and it is used to receive the mouse position from the client.
This application is suitable for many cases for e.g. sports or live events.
Client-side rendering sends the data from all cameras to the client in real time. This will add a high volume of load to the network. To avoid that we use the server-side rendering, where the server receives the user input using a websocket and only generates the necessary frames.

1.2 ARTIFICIAL INTELLIGENCE

In recent years the Artificial Intelligence (AI) has become an important part of computer science.
It has been introduced in many fields like: medical diagnosis, weather prediction, speech, Image recognition, financial industry and trading. Despite the great computing power of modern computers there are many tasks in which people are superior to computers. One of the strengths of human intelligence is the ability to adapt and learn, which a computer does not have

to large extent [5]. AI techniques should therefore generally help computers to acquire natural intelligence skills.
In this work, we take advantage of a relatively new javascript framework called Tensorflow.js to apply some intelligence on the Adaptive Bitrate (ABR) algorithm.
In addition, the modern browser introduced Web Graphics Library (WebGL) which is a browser interface to OpenGL to enable the execution of JavaScript code on a computer's Graphics Processing Unit (GPU) [17], that will solve the speed problem when Machine Learning (ML) is applied on the browser.

1.3 MOTIVATION

Before ABR was conceived there, were two ways you can go about streaming videos, you could either store the videos in full resolution and risk the users with poor connectivity having an abysmal streaming experience, or you could reduce the quality and resolution of the video, which effect the Quality of Experience (QoE), but now ABR is used to provide better quality (i.e bitrate) possible based on client connection capabilities.

Modern browsers provide many useful Application Programming Interface (API)s which can give us some information about the internet connection performance on the client-side and let us estimate the bandwidth and the latency of the network. Other APIs can provide information about the hosting device hardware because the video decoder algorithms in the video player require a lot of Central Processing Unit (CPU) resources and a higher bitrate requires more hardware resources. These data can be used in the ABR algorithms.

Many recent applications require video streaming with a small delay like FVV videos or video calls. To provide a good user experience the client-side buffer size should be as small as possible. This will increase the challenges since the buffer is one of the most important factors in ABR algorithms like Dynamic Adaptive Streaming over HTTP (DASH). The buffer prevents stuttering in playback during the network throughput fluctuation but with the small buffer size, the buffer will be quickly underflowed.

1.4 RESEARCH QUESTION

Most ABR algorithms depend on segmented media streaming, which allow them to estimate the network bandwidth by measuring the download time for each segment.
For interactive FVV rendering the segmented streaming adds a large encoding delay as it takes at least one sec to create the next segment [21].
Using Non-Segmented Streaming (NSS) will effectively reduce the delay between the client and the server, but also the bandwidth estimation using the downloaded segments will no longer be available. We replace the

bandwidth estimation with Round-Trip Time Round-Trip Time (RTT) measurement and video statistics to use them as indicators to our ABR algorithm using ML.
The research questions we evaluate in the thesis are:

- How accurate the RTT measurement in the browser?
- How the RTT changes influences the network bandwidth and eventually the targeted bitrate?
- How can the ML techniques applied to estimated ABR algorithm to achieve the target bitrate for a particular network profile?
- Which model parameter performs better in term of accuracy and training time?

1.5 CONTRIBUTION

Reviewing the network delay in terms of RTT in the browser using javascript and applying the measurement to ABR algorithm, which is an important parameter for low latency applications like interactive FVV. Introducing a non-symmetric smoothing function to filter the measurement from the network fluctuation. Implementing ML on the browser without any additional software or plugins using Tensorflow.js. Choosing the right Artificial Neural Network (ANN) model and the training method which serves our task. Testing with the network profile, how the RTT and packet loss rate can be used to find the optimal bitrate from the trained model.

1.6 STRUCTURE OF THE WORK

The document is structured in eight chapters, starting with an introduction that gives an overview of the thesis in chapter 1.
Chapter 2 discusses the basic concepts of video streaming and encoding.
Chapter 3 gives an introduction about machine learning and neural networks.
Chapter 4 discusses the related works, which cover the ABR algorithms.
Chapter 5 talks about the experiment setup and the used softwares.
Chapter 6 shows the collection and the measuring processes of network and video statistics, which will be the input data for the ANN.
Chapter 7 introduces the building and the training processes of the neural network and the evaluation results.
The final chapter 8 concludes the thesis with a brief summary of the work done.

2 STREAMING AND ENCODING

2.1 HTTP STREAMING

HTTP Streaming (HS) takes advantage of a conventional web server and uses it as a simple file server to stream media files, delivering small parts of the file which is called segments.
The files are usually stored in different qualities on the server for ABR. The client sends a HTTP request to the server, the server can reply with a single HTTP connection that remains open until the client closes it as shown in Fig.2.1. The response has a part of the total video which will be received and processed on-the-fly by the client.

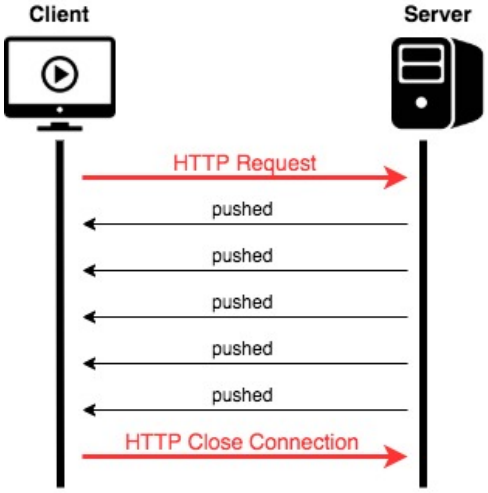

Fig. 2.1: HTTP Streaming

To guarantee there is no loss in transmitted data, HS uses Transmission Control Protocol (TCP) to bypass the firewall on the host machine [21]. But this will make the data transmission time longer especially if there are packet losses on the network, then the client must wait for the missing data to be detected and re-transmitted by the server. To overcome these problems, clients apply data buffering before playing the media [22].

2.2 ADAPTIVE BITRATE STREAMING

Because HS is segmenting media into a series of small segments this which allows ABR to be implemented. These segments will be encoded in the server in multiple bit rates and resolutions as shown in Fig.2.2.
Depending on the network conditions on the client's side, there are algorithms like DASH that makes changes to the requested quality of the video to achieve a streaming experience with less buffering.

Some of the parameter that can be adjusted with ABR are: screen size, CPU capacity, and client's buffer. Further information on these algorithms are found in chapter 4.

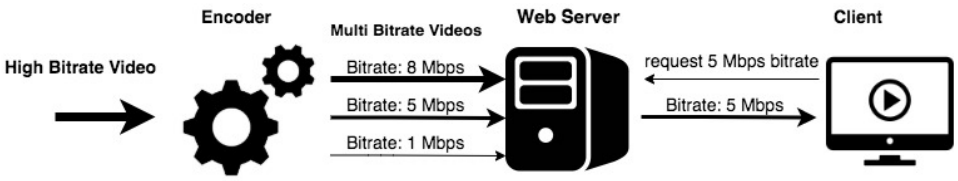

Fig. 2.2: Multi Bitrate Encoder

Some of the commercial technologies benefit from HS to (delivering ABR streaming ,are HTTP Live Streaming (HLS) developed by Apple and MPEG's standardized DASH.

2.3 VIDEO ENCODING

without the video encoder, the size of raw video files will be very big and it will not be practical to store it or transfer it through the internet. The main task of the encoder is to compress the video files into smaller sizes, the encoder uses different techniques [1] as:

- Image resizing: this will reduce the pixels needed for each frame but also reduces the details of the image. there is a direct relationship between the resolution(frame size) and the bitrate discussed more in section 2.5
- Interframe: in short the encoder will look at the differences between two frames. If the pixel didn't change its value between the two frames, it becomes redundant and it will be removed from the second frame. This will reduce the data size without reducing video quality. But the videos with higher motion will need more data to encode, which is strongly related to Variable Bitrate (VBR) (discussed in section 2.4). This method uses three types of frames, at first it needs a full-frame called keyframe, the next frame will encode the difference called a predictive frame. the third is bi-directional predictive frame, it can look backwards and forwards for other delta frames.
- Chroma subsampling: is done by reducing the color information of the video.

2.4 CONSTANT BITRATE (CBR) AND VARIABLE BITRATE (VBR)

Constant Bitrate (CBR) encodes the data in the same bitrate regardless of the content of the data. The client will receive a constant bitrate from the server so it will be easier to estimate the network bandwidth on the client's side.

In VBR the bitrate is changed depending on the detail demand of the frame sequence in the segment. This introduces additional challenges including the complex encoding pipelines for VBR. It requires more storage, and transport challenges because of the multi-time scale bitrate burstiness of VBR videos [18]. For ABR applications the Constrained VBR is usually used, which is VBR constrained to a maximum about 110% of the targeted data rate [12].

2.5 VIDEO BITRATE AND RESOLUTION RELATIONSHIP

The high video resolution needs a high bitrate and vice versa. To achieve a reasonable video quality for different resolutions the following values are recommended [7]:

Frame Size/Frame Rate	Target Bitrate (kbps)	Min Bitrate (50%)	Max Bitrate (145%)
320x240p @ 24,25,30	150	75	218
640x360p @ 24,25,30	276	138	400
1280x720p @ 24,25,30	1024	512	1485
1920x1080p @ 24,25,30	1800	900	2610
1920x1080p @ 50,60	3000	1500	4350
2560x1440p @ 24,25,30	6000	3000	8700
2560x1440p @ 50,60	9000	4500	13050
3840x2160p @ 24,25,30	12000	6000	17400
3840x2160p @ 50,60	18000	9000	26100

Table 2.1: Bitrate Resolution Values [7]

Another method to determine an average bitrate is the Kush Gauge [13]:

$$bitrate = pixelcount * framerate * f_m * 0.07 \tag{2.1}$$

Where f_m is the 'motion factor' that defines the estimated motion of the video on a scale from 1 to 4, and:

$$pixelcount = videohight * videowidth * framerate \tag{2.2}$$

3

NEURAL NETWORK

3.1 INTRODUCTION

Inspired by the nervous system, the neural network is a method of building computer programs that are able to learn and adapt to the given data on its own, much like the nervous system, the neural network consists of connected (neurons) that are placed in three kinds of layers: input layers, output layers and hidden layers as shown in Fig.3.1, there can be more than one hidden layer in the same program.

Every connection between two neurons has a specific weight that gets multiplied by the input value which controls the importance of this connection. (Fig.3.2)

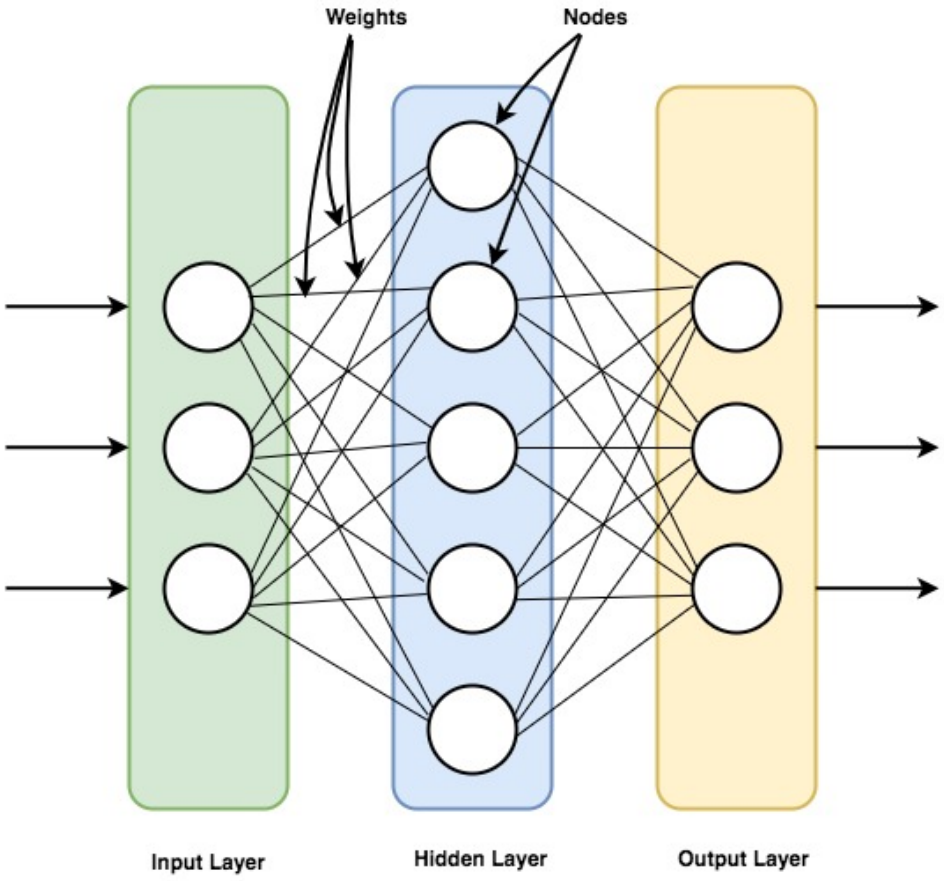

Fig. 3.1: Neural Network

Neural networks need training data to be able to predict the intended outcome, input data need to pass through all the neurons with each neuron

making its calculation and sending it to all the neurons in the next layer. After the data passes through all the layers and every neuron has applied its transformation, a label prediction will be reached for those input examples in the final layer.

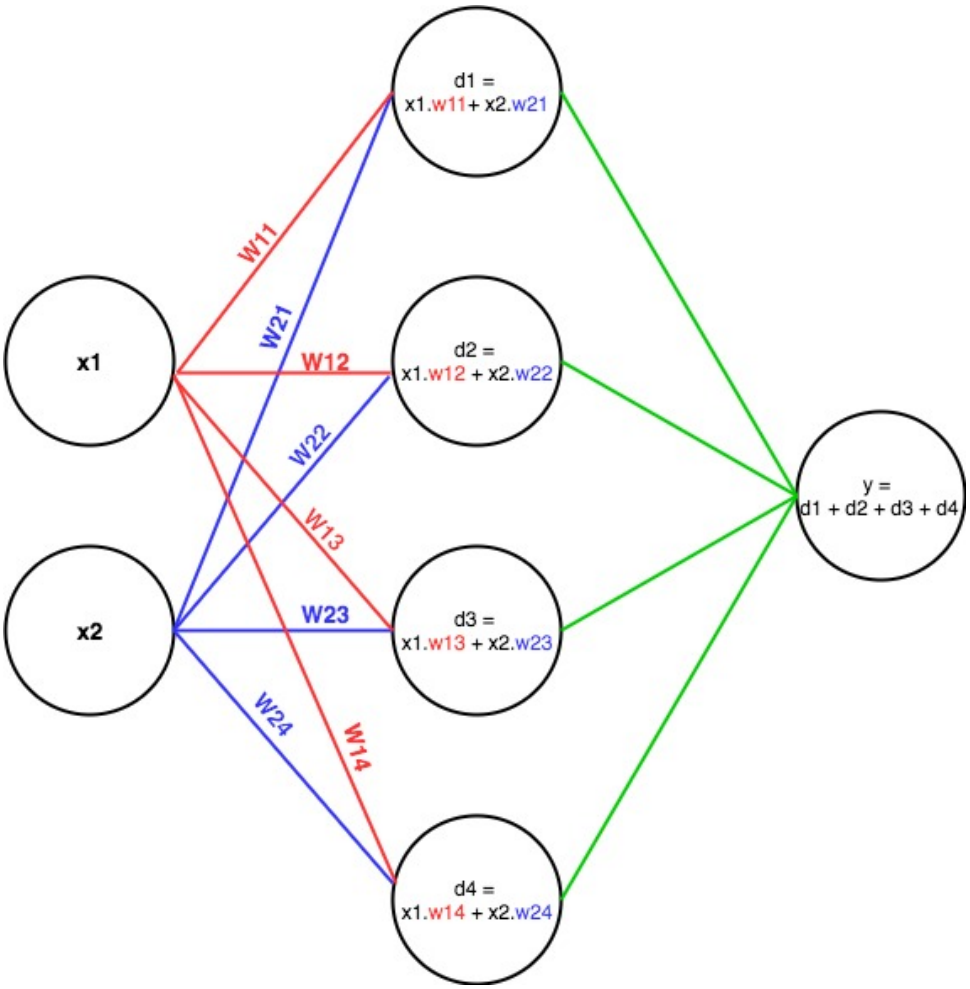

Fig. 3.2: Neural Weights

3.2 TYPES OF LEARNING

The ML learning algorithms can be classified into a variety of types, the most used ones are [2]:

- Supervised Learning: needs a data set which has input variables and output labels and it tries to learn the mapping function from the input to the output. The neural network determines the error and then adjusts the network parameters to minimize it.
- Unsupervised Learning: doesn't need a data set to teach itself. It's used when the labeled examples are not available. It learns by using a reward/punish system to refer to the preferable outcome, and the goal is not to produce a classification but to make decisions that maximize

rewards.
Unsupervised learning algorithms are usually used to extract statistical patterns from data samples [6].

- Reinforcement Learning: the model's output interacts with its environment. Those outputs (actions) affect the environment and the result will be fed again into the network input to adjust the learning process.

In a supervised learning environment, predictions are processed by a (loss function) to measure the results of the outcome in relation to the correct answer to differentiate between good and bad answers. The is key to eliminating the bad prediction by gradually changing the weights of the interconnections between neurons, and to minimize the errors and making loss function output as close as possible to zero. This is done with the help of optimizer algorithms which changes the weights in small values through multiple generations of data sets that we pass through the network in each epoch.
If there's no divergence between the estimated and the expected value then our loss function will be zero.

3.3 THE NEURAL NETWORK PARAMETERS AND HYPERPARAMETERS

The model parameters are internal variables that can be calculated by the network model from the given data i.e. the basis and the weights between neurons.
The model hyperparameters are external variables which the network model has no permission to change and cannot be estimated from the data, those are set by the programmer to adjust the learning algorithms and must be set while building the model's structure and before any training sessions. For example the optimizer, loss function, the number of layers, the number of neurons in each layer and their activation functions. Those parameters affect the speed and quality of the learning process which is largely connected to the programmer's experience.

3.4 ACTIVATION FUNCTIONS

To produce a non-linear output an activation function must be introduced which allows for a more flexible and variant functions to be created during the learning session, which increases the speed of the learning process.
The output of each neuron will be influenced by the activation function before reaching the neuron in the next layer. The most popular activation functions are:

3.4.1 *Sigmoid*

Sigmoid function will rearrange the input values of the function to values between 0 and 1 without losing any information in the data. This is very

helpful when dealing with variables that have vast ranges. It is defined by the formula [9]:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{3.1}$$

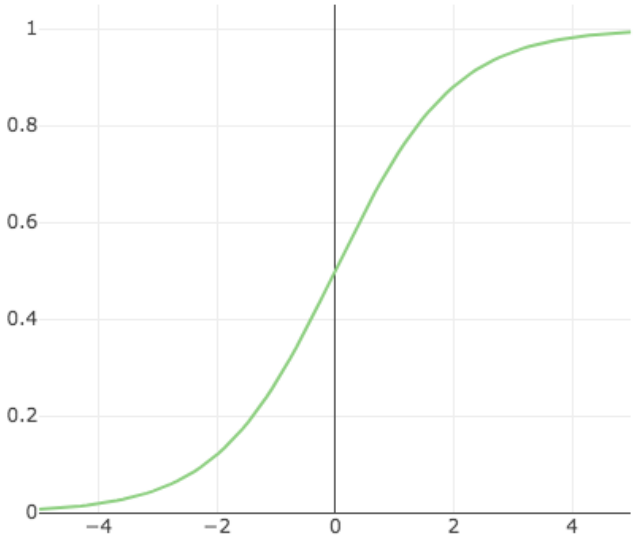

Fig. 3.3: Sigmoid Function

3.4.2 *Tanh*

Some neural networks works better with input ranges between -1 and 1, those networks can take advantage of the tanh function which represents the relationship between the hyperbolic sine and hyperbolic cosine:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.2}$$

3.4.3 *SELU*

The Scaled Exponential Linear Unit (SELU) activation function definition is [8]:

$$\text{SELU}(x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geqslant 0 \end{cases} \tag{3.3}$$

Where α and λ are pre-defined constants (α = 1.67326324 and λ = 1.05070098).
For large negative values, the SELU function will be saturated so that they are essentially inactive. It is linear for all positive values and has a small slope for negative values (Fig.3.5).

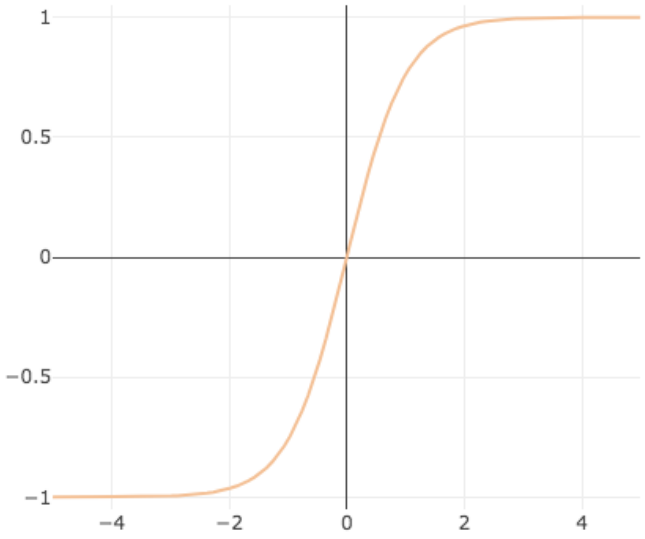

Fig. 3.4: Tanh Function

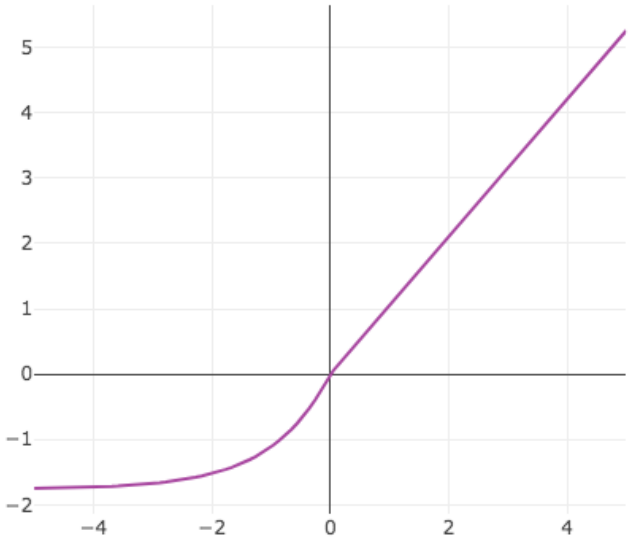

Fig. 3.5: SELU Function

3.5 LOSS FUNCTION

The value of the loss function is our indicator for the progress of the learning process, basically it's how far we are from the ideal solution, or it's the difference between the predicted value and the correct value.
Some of the advanced loss functions are [8]:

- MeanSquaredError: Computes the mean of squares of errors between the targeted outputs (labels) and predictions.
- MeanAbsoluteError: Computes the mean of absolute difference between labels and predictions
- LogCosh: Computes the logarithm of the hyperbolic cosine of the prediction error.

3.6 OPTIMIZERS

Optimizers are the functions that changes the network model by modifying the weights and basis according to the output of the loss function to make sure that the optimizers are working in the intended direction which is minimizing the value of the loss function with each epoch. Some important optimizer algorithms are: Stochastic Gradient Descent (SGD),Momentum ,RMSprop, Adagrad, Adadelta, and Adam

3.7 EPOCHS

Each time all the training data have passed through the neural network is called an epoch or a generation, higher number of epochs means more time spent in the learning process.

3.8 TENSORFLOW.JS

It's a javascript library, with it we can build and train ML models which live entirely in the browser. This has a lot of benefits, for example the user doesn't need to install any libraries, softwares or drivers to run ML, all what the user needs to do is to open the webpage.

The code will basically run over all devices no matter if it's a PC, laptop, tablet or smartphone. Additionally the most modern browsers support GPU acceleration, which will improve the runtime by allowing the ML calculations to run in GPU.
TensorFlow can be used in different programming languages and the trained model in one language can be imported and run in the other language (Fig.3.6)

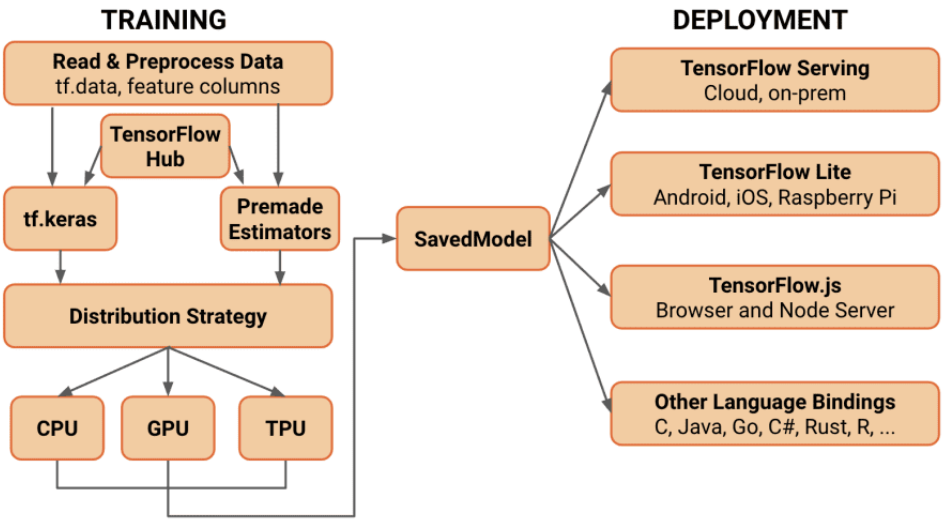

Fig. 3.6: Tensorflow Training Model [8]

4

RELATED WORKS

The ABR-Algorithms for HS in the related works depends on estimating the connection bandwidth.
Estimating the bandwidth is not so easy because HS runs in Application Layer and because of strict boundaries between layers of the OSI communication model, the HS can only see the bandwidth that TCP provides [24].

More familiar ways to estimate the bandwidth in the browser is to download a large file and measure the time of the download. This method will add unnecessary load to our application.
Since the video player is already downloading the video chunks, we can passively measure the download time for each chunk, and estimate the bandwidth with it. The data delivering rate for the video player will denote the network bandwidth.

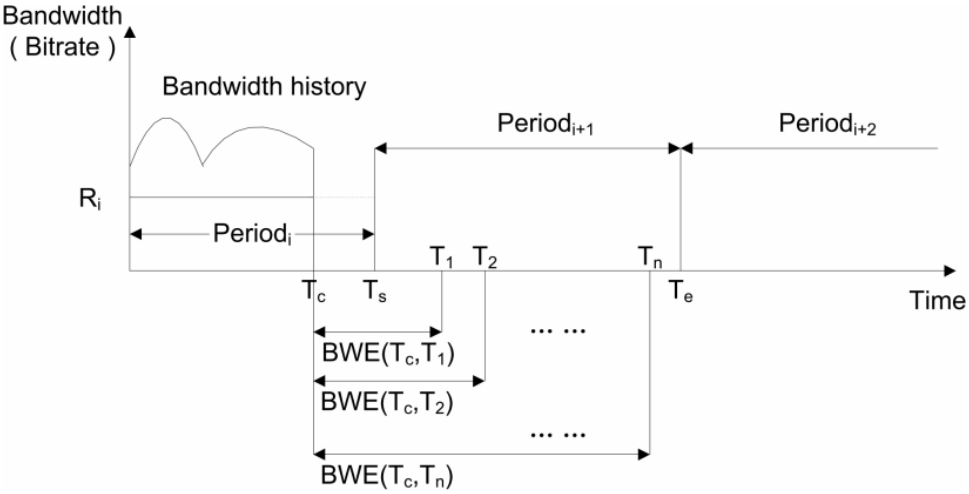

Fig. 4.1: Producer-Consumer Model for HTTP Streaming [24]

Zhou et al. in Paper [24] assumes the scenario shown in Fig.4.1, where T_c is the time in current $period_i$, the next $period_{i+1}$ is from T_s to T_e and $\{T_1, ..., T_n\}$ are time samples between T_s and T_e, B(t) is the network estimated bandwidth, and the bitrate for the next segment R_{i+1} should achieve the following condition:

$$\int_{T_s}^{t} R_{i+1}dt \leqslant \int_{T_c}^{t} B(t)(dt), \forall t \in \{T_1, \ldots, T_n\} \tag{4.1}$$

the selected bitrate for the next segment based on bandwidth history will be represented by the following equation:

$$R_{i+1} = \min\{\frac{BWE(T_c, t)}{t - T_s}, \forall t \in \{T_1, \ldots, T_n\}\} \tag{4.2}$$

Where BWE(Tc,t) denotes $\int_{T_c}^{t} B(t)(dt)$ And is calculated as the minimal amount of data that the client side downloaded during an interval with the same length (t-Tc).

Another adaptation method delivered in [14] looks at the ratio of the Media Segment Duration (MSD) to Segment Fetch Time (SFT).

$$\mu = \frac{MSD}{SFT} \tag{4.3}$$

A higher bitrate will be selected if μ is bigger than $1 + \epsilon$. Where ϵ is the bitrate switch up factor. If μ is smaller than an empirical value r_d, a lower bitrate will be selected.
In other words, the fetching time of the segment should be smaller than its playback duration. And the algorithm chooses the upper limit of the next segment bitrate.

In paper [3] the screen size that runs the video player has been considered in the ABR algorithm.
It prevents sending any unnecessary pixels to the video player if the screen size is too small to view all the pixels reducing the consumption of network bandwidth on the account of unnecessary data, delivering the same video quality to the customer more efficiently.
This way the ABR algorithm will choose the bitrate r_i is based on two conditions:
If the screen size V_i equals the resolution i and the estimated bandwidth R_t is bigger than its bitrate. If the screen size V_i larger than the resolution i but the R_t is smaller than the higher resolution. The probability of selected representation i is [3]:

$$p(i; \mathbf{r}) = \mathbb{P}(V_t = v_i, R_t > r_i) + \mathbb{P}(V_t > v_i,\ r_{i+1} \geqslant R_t > r_i) \tag{4.4}$$

The values of bandwidth and screen size are independent from each other and (4.4) can be written as:

$$p(i; \mathbf{r}) = \mathbb{P}(V_t = v_i)\mathbb{P}(R_t > r_i) + \mathbb{P}(V_t > v_i)\mathbb{P}(r_{i+1} \geqslant R_t > r_i) \tag{4.5}$$

$$R(\mathbf{r}) = \sum_{i \in \mathcal{J}} r_i p(i; \mathbf{r}) \tag{4.6}$$

Substituting (4.5) into (4.6):

$$R(\mathbf{r}) = \sum_{i \in \mathcal{I}} r_i (\mathbb{P}(V_t = v_i)\,\mathbb{P}(R_t > r_i) + \mathbb{P}(V_t > v_i)\,\mathbb{P}(r_{i+1} \geqslant R_t > r_i)) \tag{4.7}$$

Where $\mathcal{I}$ is a set of encoded representations $R(\mathbf{r})$ and the average bitrate.

The author in Paper [15] uses a software called pensieve to build and train a neural network.
ABR algorithm was generated using reinforcement learning so there is no need for training data. The ANN will interact with the environment and the reward input is the video playback statistics (Fig.4.2).

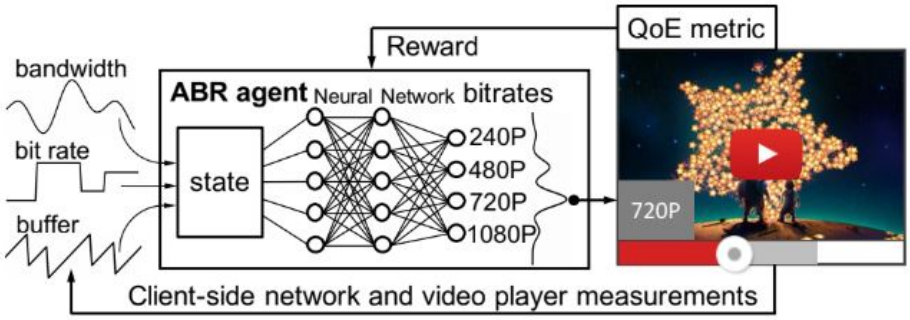

Fig. 4.2: Applying reinforcement learning to bitrate adaptation. [15]

Pensieve is installed on the server. That means the client sends always the video statistics to the server, which increase the network traffic.
In addition reinforcement learning need to interact with the environment to collect the data needed for the training [2], that leads to longer training time.

In paper [20] a ABR algorithm is implemented on a free view-piont video using NSS. The client sends the targeted bitrate to the streaming server using a websocket connection. A passive estimation for available bandwidth is measured using the video playback statistics on the client.

Algorithm 1 Signalling of Bit Rate Switch

Require: br, $\hat{b}$, β

1: **if** $\hat{b} \leqslant \beta$.br **then**
2: WebSocket Signal (0, $\hat{b}$)

3: **else if** $\hat{b} > \frac{br}{\beta}$ **then**
4: WebSocket Signal (1, $\hat{b}$)
5: **end if**

The algorithm switches to a higher or lower bitrate with help from a threshold parameter β which has a default value $\beta = 0.8$
$\hat{b}$ is the estimated bitrate and br is the desired bitrate.
The paper also compares the NSS encoding delay with segmented stream like DASH, the latter needs at least 1 second to encode one segment and received by the client (Fig.4.3).

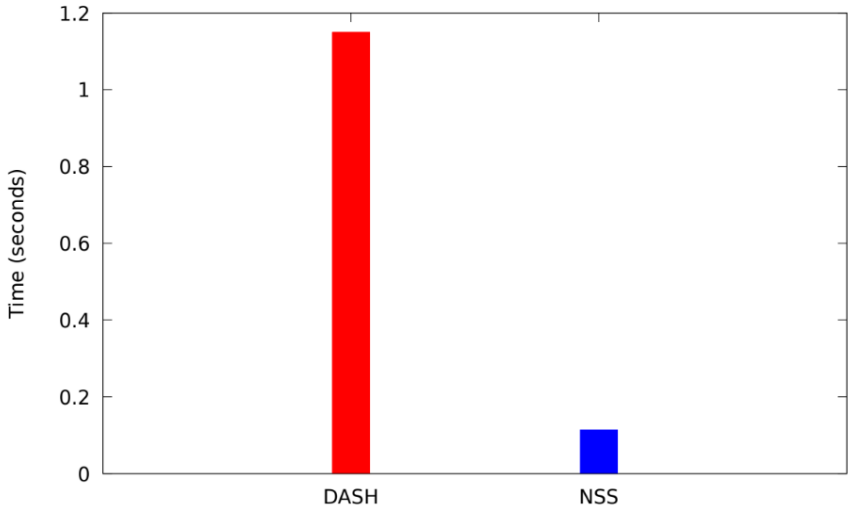

Fig. 4.3: Encoding delay comparison between NSS and DASH. [20]

To the best of our knowledge, there exist no work on the FVV streaming using the ML to find the targeted bitrate.

5

EXPERIMENT SETUP

5.1 NETWORK TOPOLOGY

The network is built with Emulab, which is a network testbed for researchers to help them in developing, debugging, and evaluating their systems [4].

The network is using dumbbell topology, which contains to group of PC's connected with two routers as shown in Fig.5.1. The first group contains two PC's, one of them is the video streaming server.
The second group contains the Client's PC's. All previous PC's run under ubuntu.
All connections between the PC's and the routers are set to 100 MBits.

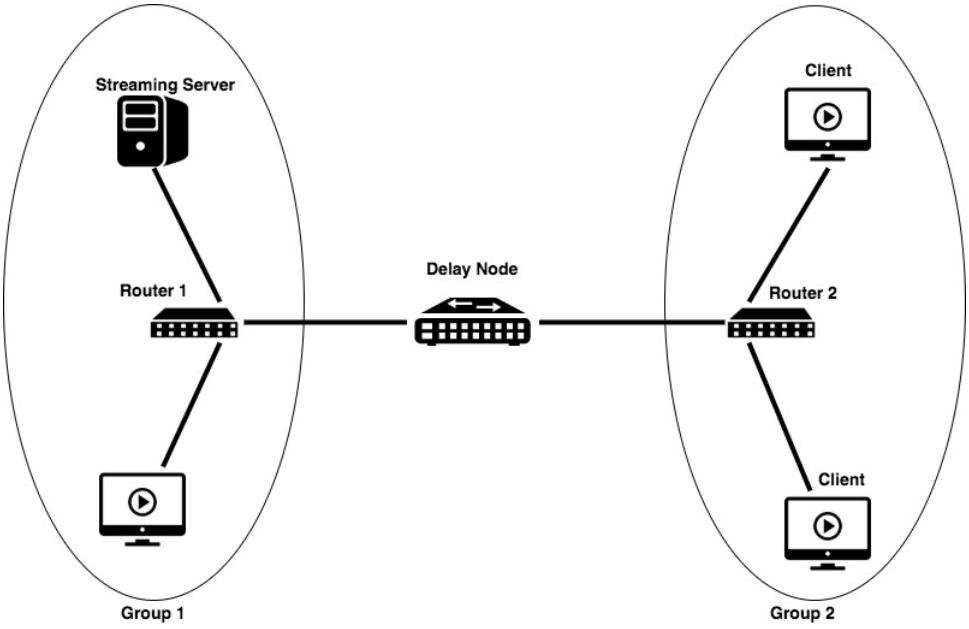

Fig. 5.1: The Network Topology

Additional PC worked as delay node is used to control the network traffic. It is located between the routers, and it's running under FreeBSD operating system [11].
IPFW is a stateful firewall which is included in the basic FreeBSD. The network latency and the network loss rate are controlled by IPFW. All PC's can be accessible through Secure Shell (SSH) connection.

5.2 SERVER/CLIENT-STRUCTURE

The client communicates with the server through the following:

- HTTP Requests: to load the webpage. Therefore we need to install a webserver lighttpd. The video will be also streamed over HTTP/TCP.
- Websocket: to send the mouse coordinates to the render. The render uses (libwebsockets) library to be able to receive data from the client.

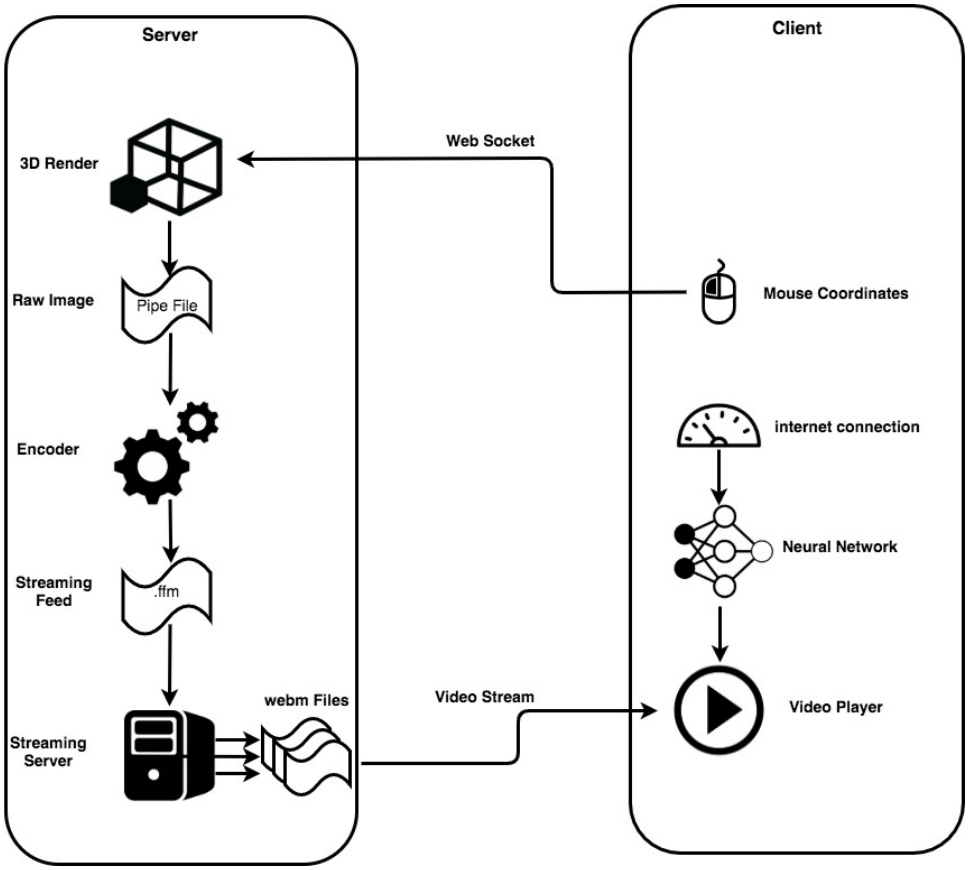

Fig. 5.2: Server Client Overview

5.2.1 *The Server*

The server-side, runs rendering application to achieve an interactive FVV [21] which receives the mouse coordinates from the client and generates 3D images. The images will be saved in a pipe file which will be in itself an input for the encoder (ffmpeg). ffmpeg will convert a series of images to a video and encode it to webm video format with 24 Frames Per Second (FPS) and feed it to the streaming server (ffserver).

Software	Type	Version
Ubuntu	Operating System	18.04.3 LTS
ffmpeg, ffserver	encoding, streaming	2.1.git
lighttpd	web server	1.4.45

Table 5.1: Server Software components

5.2.2 *The Client*

The client uses google chrome browser without plugins. The browser has all the required API's, which loads the needed libraries directly from Content Delivery Network (CDN) when the webpage starts. Some of the libraries and API are:

- Performance API: allow us to accurately calculate the time between the request and the response to estimate the network latency (supported in Chrome v.6 - 81).
- Tnsorflow.js: to allow us to build the neural network in the browser.
- WebGl API: we didn't use it directly but tnsorflow.js needs it in order to accelerate the training and prediction processes (supported in Chrome v. 56 - 81).
- Web Hypertext Application Technology Working Group (WHATWG): is embedded into HTML5 to get the video statistics.
- Vis API: to draw the diagrams directly in the browser so there is no need to export the data out of the browser to draw them.

Software	Type	Version
Ubuntu	Operating System	18.04 LTS
Tensorflow.js	Machine Learning js Library	1.4.0
Chrome	Browser	58.0.3029.96

Table 5.2: Client Software components

Teil II

IMPLEMENTATION

6 COLLECTING VIDEO AND NETWORK STATISTICS

6.1 INDEXED WEBPAGE

The indexed webpage contains the default video element provided by HTML5 to display data from the streaming server.
All the calculated information about the network status is located on the right side of the HTTP webpag, the client hardware and the playback video statistics (Fig.6.1).

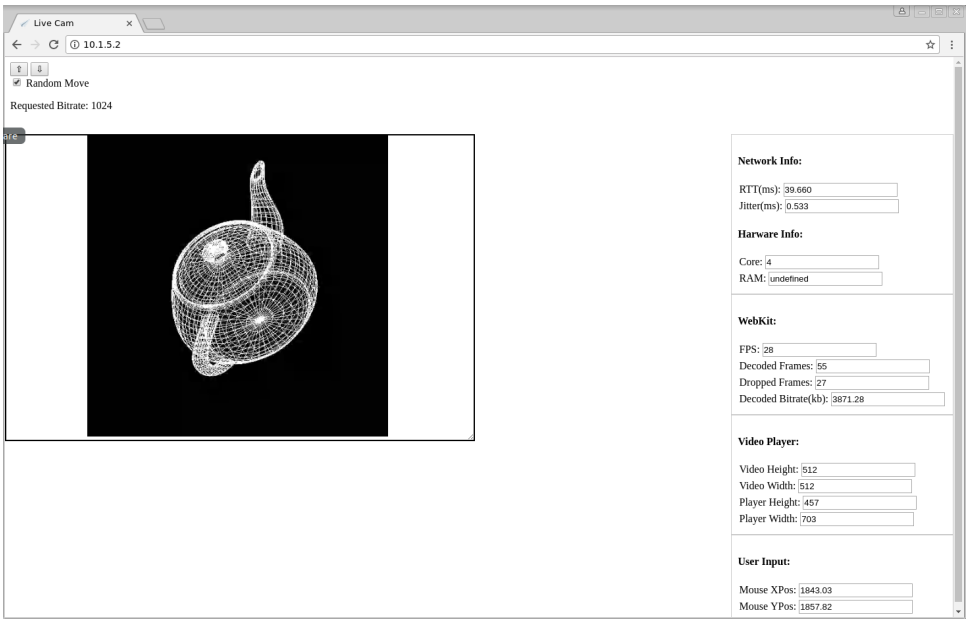

Fig. 6.1: Starting Webpage

This data is needed for preparing the training data for ML. and for predicting the required streaming bitrate.
As soon as the web page is loaded, it runs an initial test to collect information about the network statistics, then it will run the test every 5 seconds. And the video playback statistics are calculated every 1 second.
Mouse position is catched by adding an event listener to (mousemove).
The number of CPU cores can be known using hardwareConcurrency API.

6.2 RECEIVED BITRATE AND FPS

To calculate the received bitrate in the client and the fps of the video we used the solution implemented in [21] which relies on Webkit. *webkitDecodedByteCount* and *webkitDecodedFrameCount* are the main metrics we used. Those two metrics start counting when the video starts. To get the received bitrate (b)

and (fps) for every second we subtract the current value with the value one second earlier.

$$b = \frac{d(webkitDecodedByteCount)}{dt} \tag{6.1}$$

$$fps = \frac{d(webkitDecodedFrameCount)}{dt} \tag{6.2}$$

6.3 RTT

RTT will give us an important overview about the network state because it has a direct relationship to network bandwidth and Loss [16]:

$$BW = \frac{MSS}{RTT}\frac{C}{\sqrt{p}} \tag{6.3}$$

where BW is network bandwidth, MSS is maximum segment size, C is a constant, and P is loss. As we see RTT is Inversely proportional with network bandwidth.

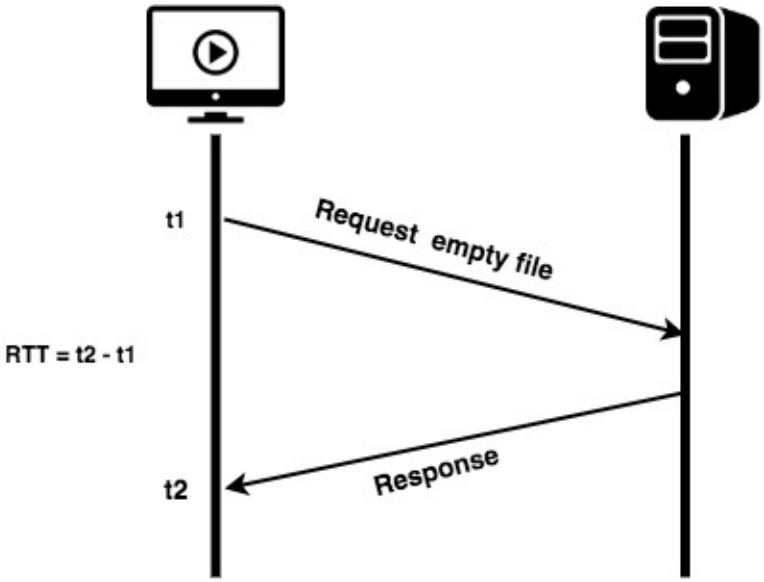

Fig. 6.2: Calculating RTT

To get RTT value an HTTP request will be sent and we'll measure the time to get the response, that means it isn't passive measuring.To reduce the measuring impact on the network load as possible we request an empty file from the server (Fig.6.2).

The flowchart in Fig.6.4 shows the measuring steps.
The measuring starts when a request is sent at the same time we start

timing and waiting for the response. If the browser doesn't support the performance API we calculate RTT using the difference between send and response time. More accurately will be using performance API to get the exact request time. Luckily almost all modern browsers support this API. We repeat these steps several times and we take the average value from it (4 times in our case).
The process time in the server to deal with the request is ignored.

To prevent cashing in the browser a random string is added to the request. The request will be as follow:
$xhr[0].open("GET", "http://10.1.5.2/empty.js" + url\_sep("http://10.1.5.2/empty.js") + "r=" + Math.random(), true);$

Where (10.1.5.2) is the ip address of the video server, empty.js is the empty file, and Math.random() generates a random number.

6.3.1 *Measurement Accuracy*

The accuracy of this measurement is tested by changing the network latency using ipfw on the delay node between the server and the client. Fig.6.3 shows the error percentage of the measurement compared with the ping function.

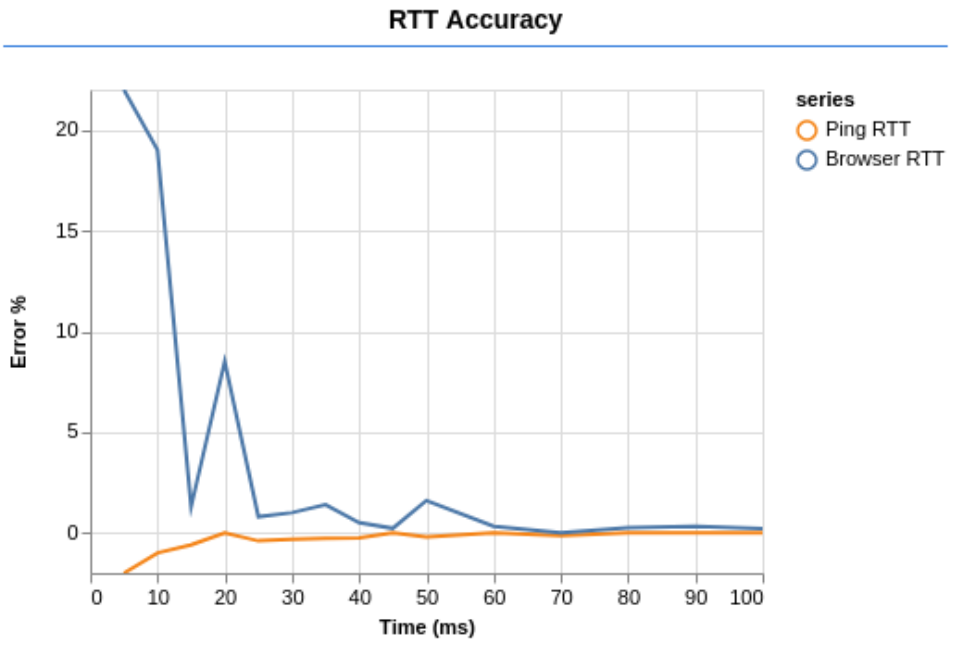

Fig. 6.3: RTT Error

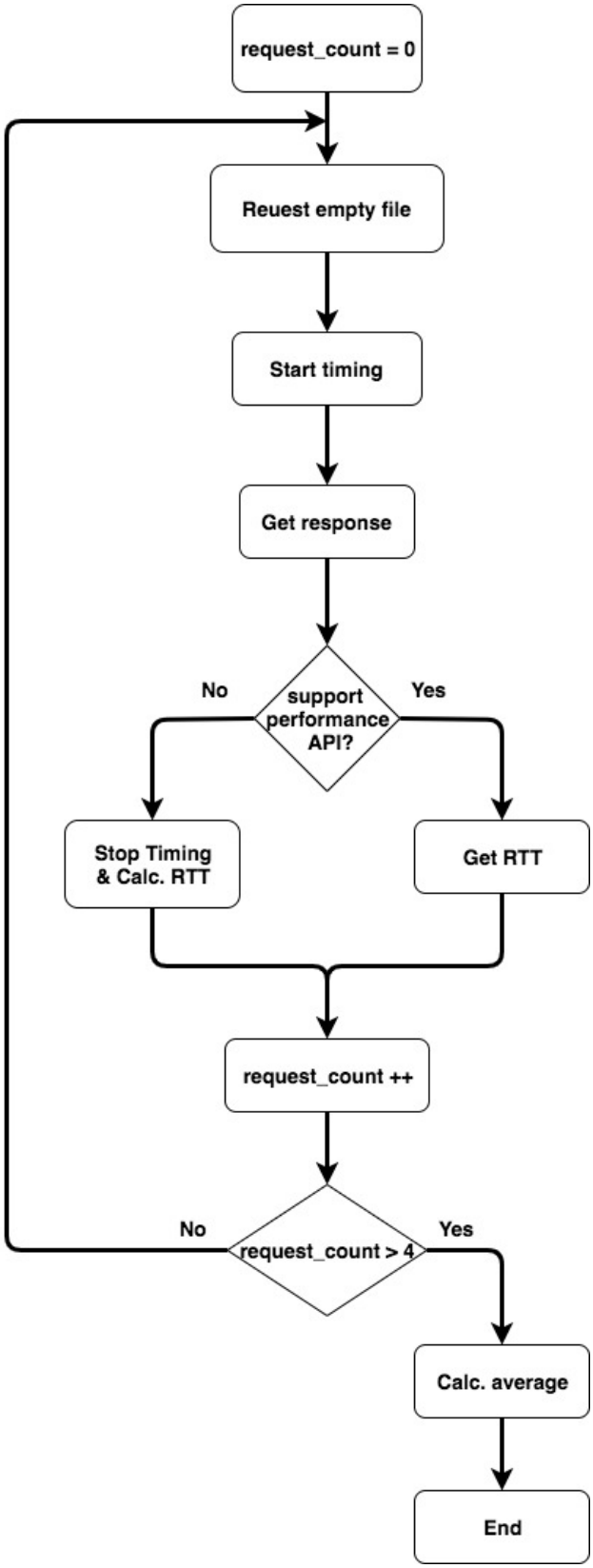

Fig. 6.4: RTT FlowChart

6.4 JITTER

Jitter is the amount of changes in latency measured in milliseconds (ms). It refers to the fluctuation in the latency of the received packets. We can calculate it by subtracting the current RTT with the last one.

6.5 SMOOTHING FUNCTION

To reduce the effect of shot bursts on the ANN input from the network fluctuation, the measured values (received bitrate, FPS and DecodedFrames) should go through a smoothing function. The used function is the exponential smoothing[21]:

$$\alpha = e^{\frac{\log(\frac{1}{2})}{\eta}} \tag{6.4}$$

Where η defines the half-life of the estimation.
The smoothing function will be applied to measured bitrate as follows:

$$\hat{b}_{(n)} = (1-\alpha).b_{(n)} + \alpha.\hat{b}_{(n-1)} \tag{6.5}$$

Considering $b_{(n)}$ is the bitrate sample, $\hat{b}_{(n)}$ is smoothed bitrate, and the current value (n) and the previous value $(n-1)$.

The half-life parameter η represents how quickly the smoothing function output will reach the end value.
If η too small the smoothed output will be changed quickly and the connection fluctuation will pass through. If it is too large the fluctuation will be filtered, but it can lead to stuttering in the video playback, when the connection conditions becomes worse.
To overcome this problem, a non-symmetric smoothing function is implemented, which has two half-lifes to react quicker, when the sample bitrate lower than it's previous smoothed bitrate.

Algorithm 2 Signalling of Bit Rate Switch

Require: $\hat{b}_{(n-1)}, b_{(n)}$

1: **if** $\hat{b}_{(n-1)} \leqslant b_{(n)}$ **then**
2: $\quad \eta = 3;$

3: **else if** $\hat{b}_{(n-1)} > b_{(n)}$ **then**
4: $\quad \eta = 1.5;$
5: **end if**
$\hat{b}_{(n)} = \mathrm{Smooth}(b_{(n)}, \eta);$

Fig.6.5 shows comparison between normal exponential smoothing function and non-symmetric smoothing function, notice that the non-symmetric

function (in blue) responds quicker when the the received bitrate goes down, but it changes it's output value slower when the bitrate goes high again.

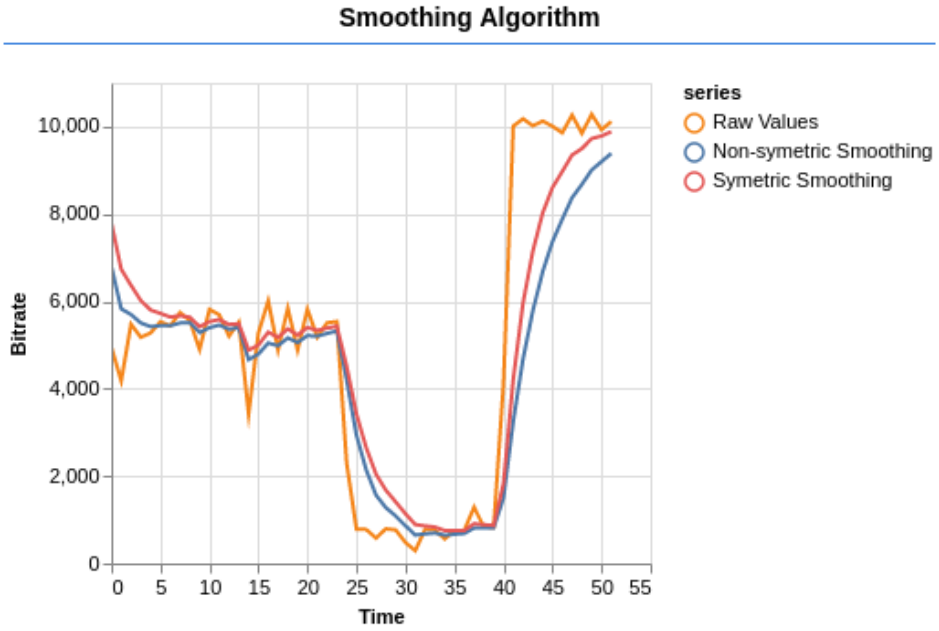

Fig. 6.5: Non-symetric Smoothing

7 NEURAL NETWORK

7.1 CREATING THE NETWORK MODEL

To apply ML in the browser the latest stable version of tensorflow.js (Version 1.4.0) is imported using CDN when the webpage is loaded. A layer API is used to build the model, which allows the developer to add a linear stack of layers to the model.

7.2 NETWORK ARCHITECTURE

A feedforward neural network architecture is applied. The Fig.7.1 shows that the network has two hidden layers, every layer has 100 nodes. The activation function on each node is (sigmoid).
the network inputs are RTT, FPS, DecodedBit, and DecodedFrames.
The output of the network is the targeted bitrate, which is one of three values: 300, 2000, 5000 bps.

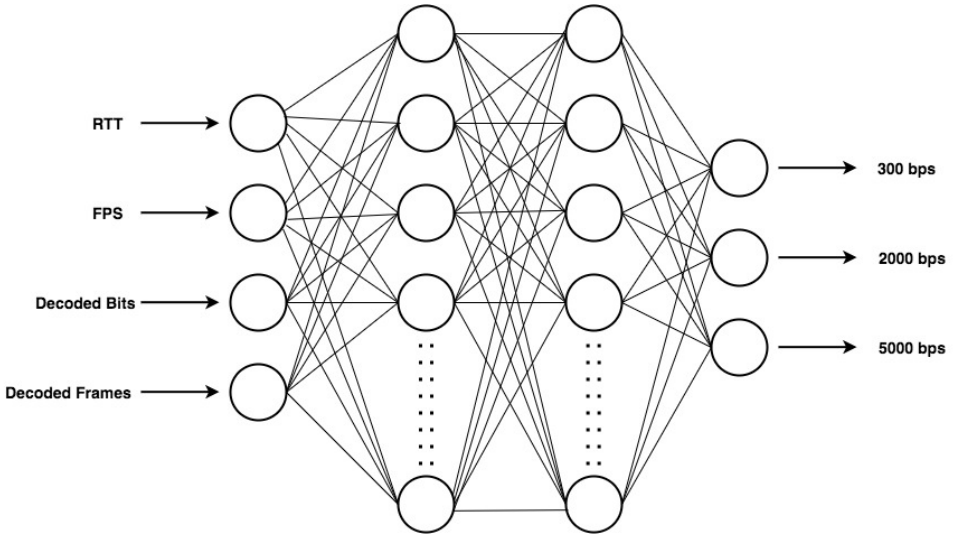

Fig. 7.1: Network Architecture

Since the ML runs in the browser, it's very important to keep the model as small as possible. A small size model will perform faster at prediction time. Also it has faster training time and takes fewer processing capacity. Especially if the user is trying to open the webpage on a smartphone. Additionally the model will take smaller storage space if it is saved in the browser data.

7.3 COLLECTING THE TRAINING DATA

To train the network we used the supervised learning method, which requires a data set that contains network input examples attached to it's output (Labeled data).
The training data set should cover all network connection cases, and it is saved as JSON file on the server. The server provides the file when the client wants to train the ANN.

To simulate different cases of the network connection, we used ipfw which runs in the delay node between the routers (see Fig.5.1). Ipfw allows us to change the network latency and the packet loss rate.
The network inputs are collected from the webpage, which are basically the video and the network statistics.
The targeted bitrate is chosen with help from iperf [10], which shows the available bandwidth in the network connection between the client and the server.

7.4 TRAINING THE MODEL

In the training phase the model parameters will be obtained using adam optimizer and the(meanSquaredError) loss function.
Training involves several steps [8]:

- Getting a batch of data to the model.
- Asking the model to make a prediction.
- Comparing that prediction with the "true"value.
- Deciding how much to change each parameter so the model can make a better prediction in the future for that batch.

Fig.7.2 shows the loss using adam optimizer with learning rate (0,001).

Using larger learning rate can some times lead to the same results with smaller number of epochs. as shown in Fig.7.3, where the learning rate is (0,1).

To make the training process more efficient, we randomize the order of the training samples by shuffling them for each epoch. It can be done by activating the shuffle parameter in the training function.
For this example the training time took about 23 second. The training time can be faster if we reduce the node number in each layer and the number of epochs.

Fig.7.4 shows the loss function using (sgd) optimizer for the training process. The prediction error in this case didn't go under 20%, that means the model prediction has less accuracy.

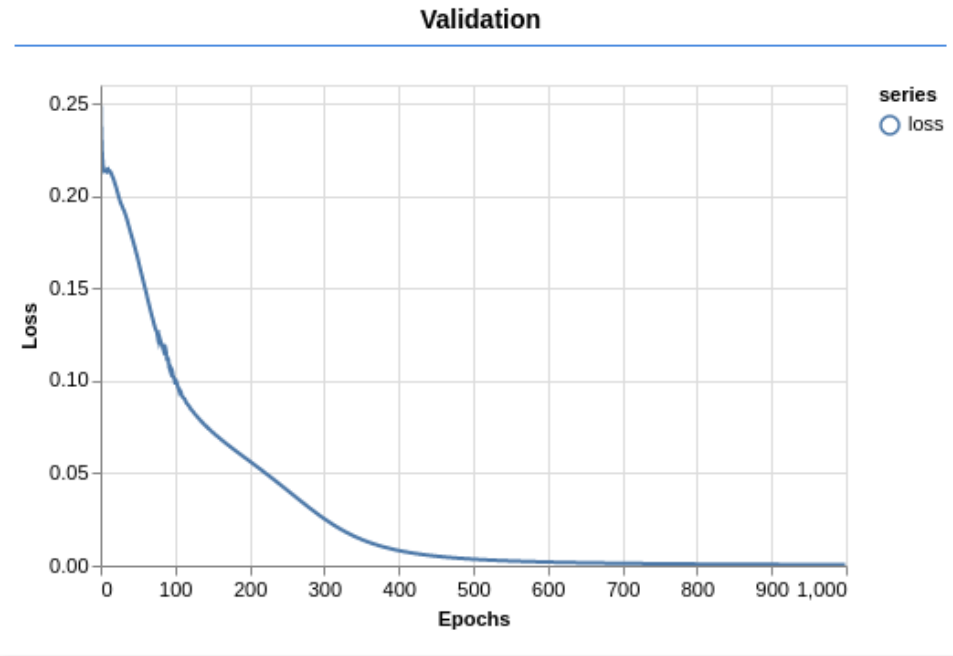

Fig. 7.2: Loss Function Output with learining rate (0,001)

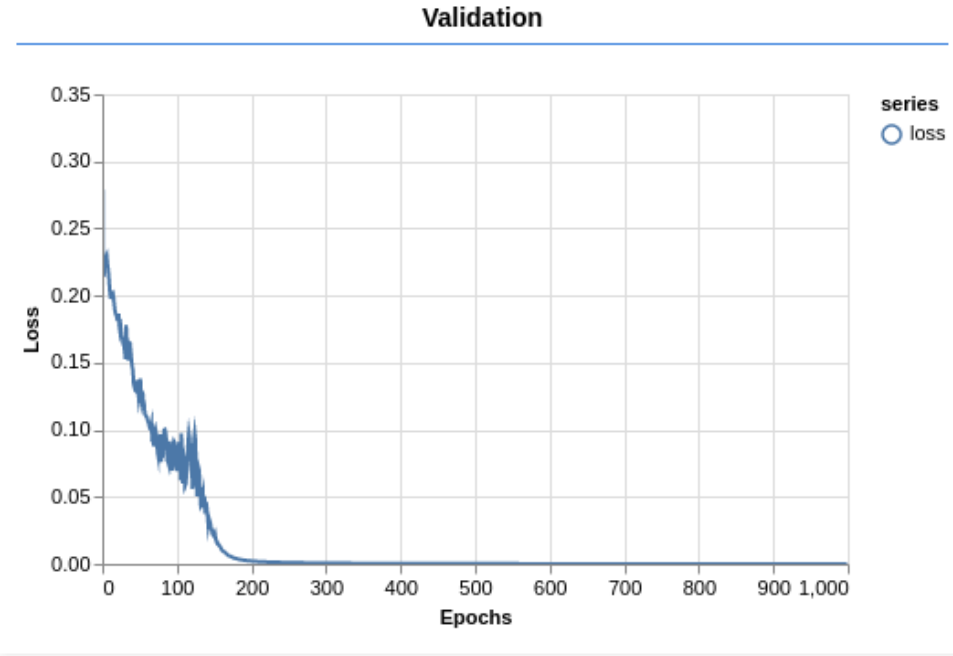

Fig. 7.3: Loss Function Output with learning rate (0,1)

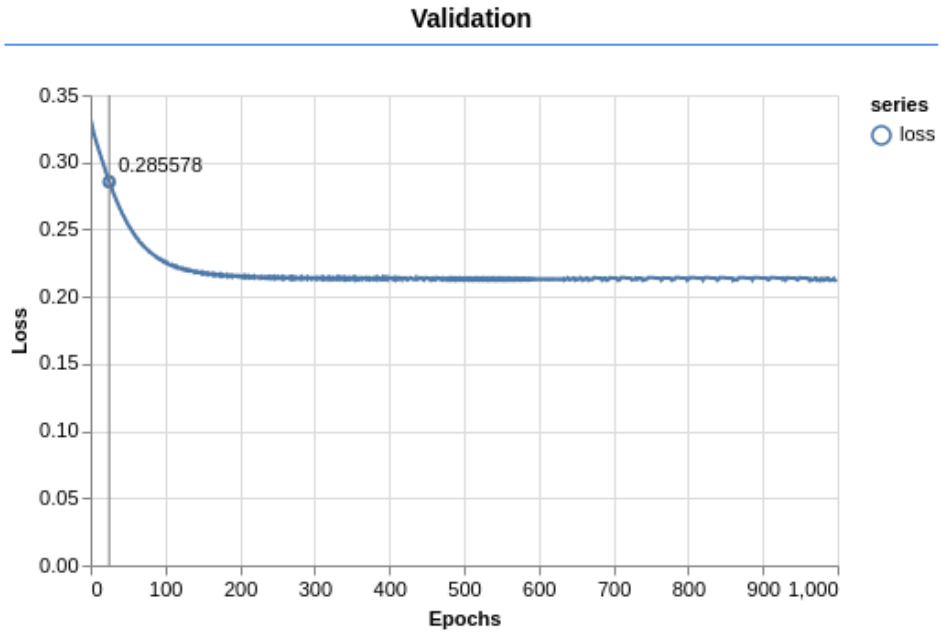

Fig. 7.4: Loss Function Output with using sgd optimizer)

7.5 SAVING THE TRAINED MODEL

The browser in the client side needs to train the model just ones, because the model will be saved in the browser local storage after finishing the training process. This will save the training time in the future use.
As Fig.7.5 shows, the trained model will be loaded if there is one in the browser local storage.

7.6 NETWORK PROFILE AND NEURAL NETWORK OUTPUT

With ipfw we set the Packet Loss Rate (PLR) to 0.5% and changed the latency between (0-100) and measured the network bandwidth between the client and the server using iperf.
Then we set PLR to 1% and remeasured the bandwidth again as shown in table 7.1.

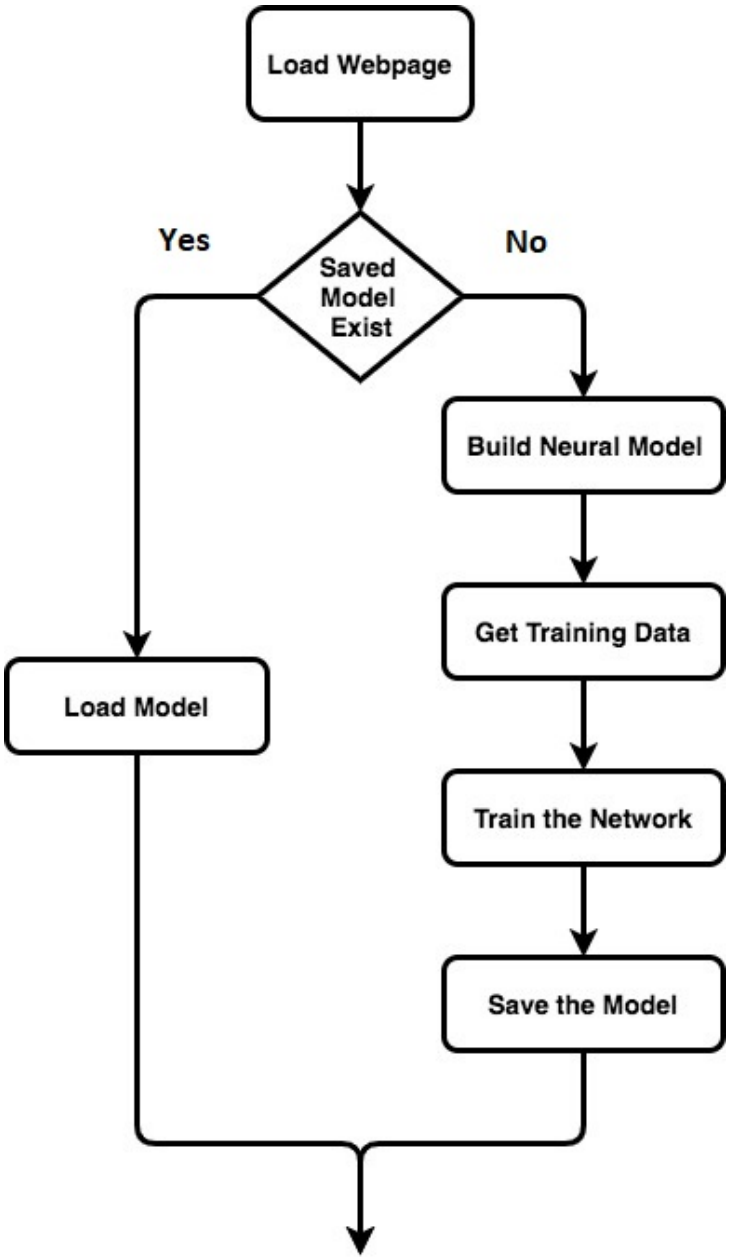

Fig. 7.5: Storing The Trained Model

RTT (ms)	BW with 0.5% plr(Mbps)	BW with 1% plr(Mbps)
0	177	112
10	16.80	12.10
20	7.32	7.59
30	6.80	3.05
40	4.30	3.05
50	3.81	2.41
60	3.60	2.24
70	2.19	2.04
80	2.09	1.45
90	2.04	1.34
100	1.82	1.04

Table 7.1: Network Bandwidth

We tested our ANN output with the same network conditions and the result shows that the ANN reacts to changes in RTT and PLR.
The targeted bitrate in the ANN output was always smaller than the network bandwidth measured by iperf to avoid video stuttering. Fig.7.6 and Fig7.7 reveal the ANN output.

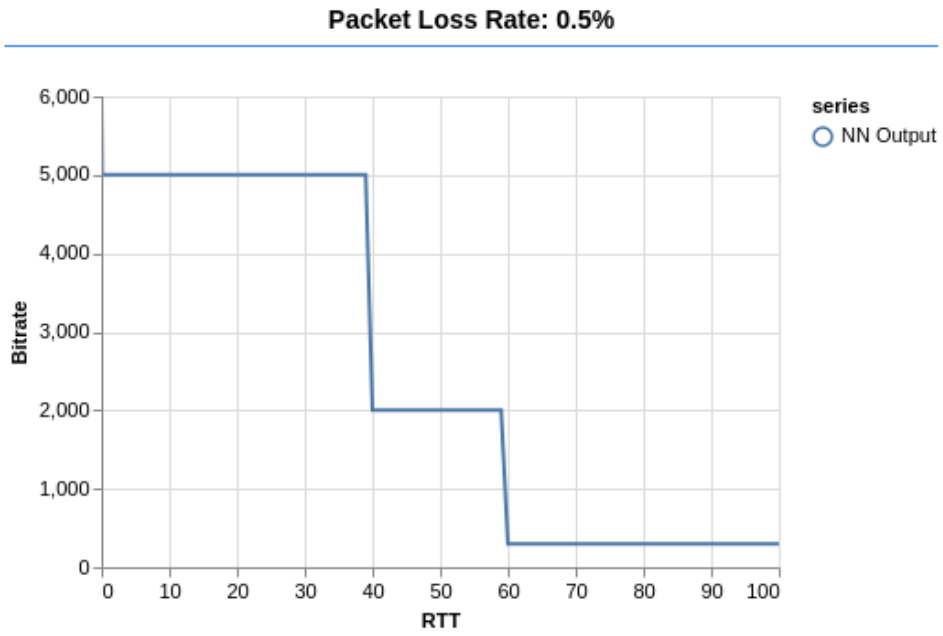

Fig. 7.6: Neural Network Output, PLR = 0.5%

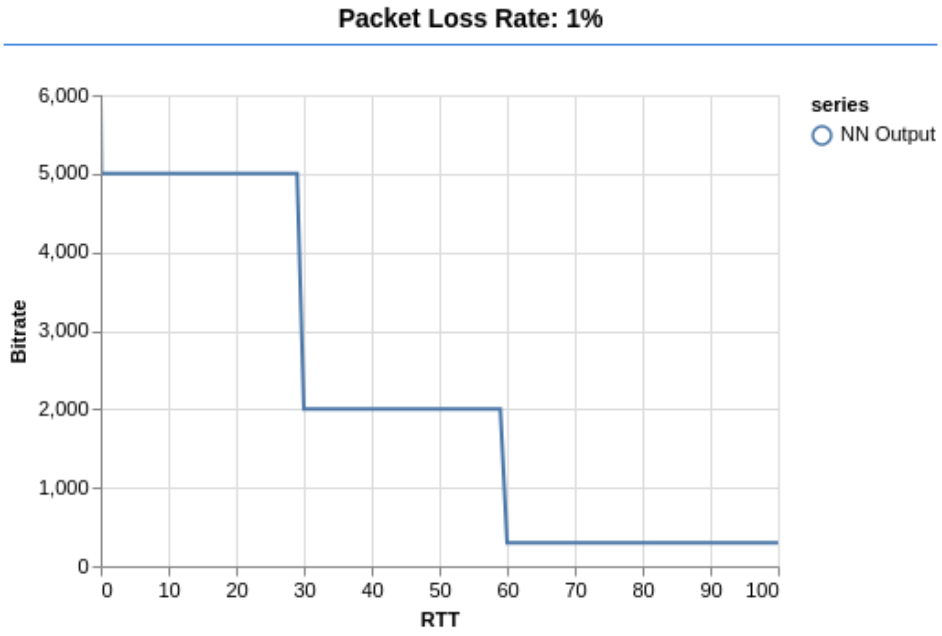

Fig. 7.7: Neural Network Output, PLR = 1%

8 CONCLUSION

In this work we implement ML on the client side without any additional software or plugins. We apply ABR solutions on low latency application (free view-point video) to select one of three different bitrates (300-2000-5000) based on network conditions.
Lightweight active RTT measurement in the browser is introduced by sending a request to the server and waiting for the response and with help from performance API we can accurately calculate the RTT, which used for network profiling
A passive measurement was introduced to acquire the video statistics using WHATWG Group, these measurements will be used later for the neural network input.
Using tensorflow.js we built and trained a feed forward neural network. We trained a variety of models with different parameters and selected the ones with better accuracy.
We saved the training model in the browser data using local storage API, so the user doesn't need to wait for the training process in the times after and then we introduced a non-symmetrical exponential smoothing function to filter the network fluctuation.
In this thesis we tested the ML on a PC which runs under ubuntu and the data set of the neural network is calculated under emulab environment. Applying ML in the browser allows us to run the ABR algorithm on a large number of different devices. In the future we can apply this ABR algorithm to other devices like smartphones and tablets which can connect to the internet in a variety of different ways like WiFi and 4G, and therefore larger data set for neural network model training will be needed.

Teil III

APPENDIX

A

CODE

HTML file:

```
<!DOCTYPE html>
<html>

<head>
  <title>Live Cam</title>
  <!-- Import TensorFlow.js -->
  <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs/dist/tf.min.js">
      </script>
  <!-- Import tfjs-vis -->
  <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis@1.0.2/dist/
      tfjs-vis.umd.min.js"></script>
  <!-- Shaka Player -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/shaka-player/2.5.5/shaka
      -player.compiled.js"></script>



</head>

<style>
  body {
    background: white;
  }
  main {
    display: grid;
    grid-template-columns: 2fr 1fr;
    justify-items: center;
  }
  .vPlayerBox {
    position: absolute;
      left: 0;
  }
  .dataBox {
    position: absolute;
      right: 0;
  }
  .dataBox div {
    border: 1px #ccc solid;
    padding: 10px;
  }
  form {
    padding: 2px;
  }
</style>

<body>
  <button id="up">&#8679</button>
  <button id="down">&#8681</button>
  <div><input type="checkbox" id="rendomMove" > Random Move</div>
```

```
  <p id="bitRateLabel">Requested Bitrate:</p>
  <br />
  <main>
    <div class="vPlayerBox" style="height:30%; width:40%;resize:both; border:2
        px solid; overflow:auto">
      <video id="myVideo" preload="auto" autoplay="autoplay" controls
        style="height:99%; width:99%">
        Your browser does not support HTML5 streaming!
      </video>
    </div>

    <div class="dataBox">
      <div class="nInfoBox">
        <h4>Network Info:</h4>
        <form>RTT(ms): <input type="text" id="rtt"></form>
        <form>Jitter(ms): <input type="text" id="jitter"></form>
        <h4>Hardware Info:</h4>
        <form>Core: <input type="text" id="hw"></form>
        <form>RAM: <input type="text" id="ram"></form>
      </div>
      <div calss="webKitBox">
        <h4>WebKit:</h4>
        <form>FPS: <input type="text" id="fps"></form>
        <form>Decoded Frames: <input type="text" id="decodedFrames"></form>
        <form>Dropped Frames: <input type="text" id="droppedFrames"></form>
        <form>Decoded Bitrate(kb): <input type="text" id="decodedBitrate"></
            form>
      </div>
      <div class="playerBox">
        <h4>Video Player:</h4>
        <form>Video Height: <input type="text" id="videoHeight"></form>
        <form>Video Width: <input type="text" id="videoWidth"></form>
        <form>Player Height: <input type="text" id="playerHeight"></form>
        <form>Player Width: <input type="text" id="playerWidth"></form>
      </div>
      <div calss="uInputBox">
        <h4>User Input:</h4>
        <form>Mouse XPos: <input type="text" id="mouseXPos"></form>
        <form>Mouse YPos: <input type="text" id="mouseYPos"></form>
      </div>
    </div>
  </main>
  <script src="JavaScript/script.js"></script>
  <script src="JavaScript/rttTesting.js"></script>
  <script src="JavaScript/neural.js"></script>
  <script src="JavaScript/draw.js"></script>
</body>

</html>
```

Main Script:

```
var mousePosSocket;
var SwitchSocket;
var videoPlayer;
var hwMemory;
var hwConcurrency;
```

```
var randomMove = false;

const MousePosUpdateRate = 30;
const calcDataHighFrequent = 1000;
const calcDataLowFrequent = 4000;

var trainingSet = [];
var bitRate = 5000;

document.addEventListener("DOMContentLoaded", () => {
    initTests();
    videoPlayer = document.getElementById('myVideo');
    videoPlayer.src = 'http://10.1.5.2:8090/test1.webm';
    hwConcurrency = window.navigator.hardwareConcurrency;
    hwMemory = navigator.deviceMemory;
    mousePosSocket = new WebSocket('ws://10.1.5.2:9000/ws/');
    mousePosSocket.binaryType = 'arraybuffer';
    document.addEventListener("mousemove", handleMoveEvent);
});

function startVideo() {
    var mpd = "EncoderPipes/Streams/livehd.mpd";
    var mpdUrl = 'https://turtle-tube.appspot.com/t/t2/dash.mpd';
    shaka.polyfill.installAll();

    var video = document.getElementById('myVideo');

    var player = new shaka.Player(video);

    player.load(mpd);

}

document.getElementById("up").addEventListener("click", () => {
    /*trainingSet.push({
        RTT: rtt,
        FPS: fps,
        DecodedBitrate: weightedDecodedBit,
        DecodedFrames: decodedFrames
    });
    console.log(trainingSet);*/
    //drawRTTError();
});

document.getElementById("down").addEventListener("click", () => {

    console.log(getNeuralPrediction());

});

function getNeuralPrediction(){
    var data = [{
        "FPS": fps,
        "RTT": rtt,
        "DecodedBitrate": weightedDecodedBit,
        "DecodedFrames": decodedFrames
    }];
    return predictBitrate(data);
```

```
}

//* random move or from mouse:
var t;
document.getElementById("rendomMove").addEventListener("click", () => {
    randomMove = (document.getElementById("rendomMove").checked) ? true :
        false;
    if (randomMove) {
        document.removeEventListener("mousemove", handleMoveEvent);
        t = setInterval(() => {
                [X, Y] = getRandomPosition();
                sendMousePos(X, Y);
            },
            MousePosUpdateRate);
    } else {
        clearInterval(t);
        document.addEventListener("mousemove", handleMoveEvent);
    }
});

function handleMoveEvent(e) {
    var X = e.clientX.toFixed(2);
    var Y = e.clientY.toFixed(2);
    sendMousePos(X, Y);
}

async function initTests() {
    // if there is no saved model
    if (localStorage.getItem("tensorflowjs_models/my-model/info") === null) {
        buildNeural();
        getTrainingData();
    } else {
        model = await tf.loadLayersModel('localstorage://my-model');
    }
    calcNetworkData();
}

function sendMousePos(X, Y) {
    document.getElementById("mouseXPos").value = X;
    document.getElementById("mouseYPos").value = Y;

    var bytearray = new Int32Array(2);
    bytearray[0] = X;
    bytearray[1] = Y;
    if (mousePosSocket.readyState === WebSocket.OPEN)
        mousePosSocket.send(bytearray.buffer);
}

var xPos = 0;
var yPos = 0;

function getRandomPosition() {
    if (xPos > 10000) {
        xPos = 0;
        yPos = 0;
    }
    xPos += Math.random() * 3;
    yPos += Math.random() * 3;
```

```
    return [xPos.toFixed(2), yPos.toFixed(2)];
}

var decodedFrames;
var oldDecodedFrames = 0;
var droppedFrames;
var oldDroppedFrames = 0;
var decodedBitrate;
var oldDecodedByterate = 0;
var fps
var weightedFPS = 12;
var weightedDecodedBit = 1000;

function calcVideoData() {
    decodedFrames = (videoPlayer.webkitDecodedFrameCount - oldDecodedFrames) *
         1000 / calcDataHighFrequent;
    oldDecodedFrames = videoPlayer.webkitDecodedFrameCount;

    droppedFrames = (videoPlayer.webkitDroppedFrameCount - oldDroppedFrames) *
         1000 / calcDataHighFrequent;
    oldDroppedFrames = videoPlayer.webkitDroppedFrameCount;

    decodedBitrate = (videoPlayer.webkitVideoDecodedByteCount -
        oldDecodedByterate) * 8 /
        calcDataHighFrequent; // 1000/1000 ms->s  B->KB
    oldDecodedByterate = videoPlayer.webkitVideoDecodedByteCount;

    fps = decodedFrames - droppedFrames;
    weightedFPS = exponentialSmoothing(weightedFPS, fps);
    //console.log("WFPS: " + weightedFPS + " weightedDecodedBit: " +
        weightedDecodedBit);

    weightedDecodedBit = exponentialSmoothing(weightedDecodedBit,
        decodedBitrate);
}

var rttArray = [];
var rtt;
var weightedRTT = 8;
var jitterArray = [];
var jitterVal = 0;
var weightedJitter = 1;


function calcNetworkData() {
    pingTest(() => {
        var sum = 0;
        rttArray.forEach(element => {
            sum = +sum + +element;
        });
        rtt = sum / rttArray.length;
        weightedRTT = exponentialSmoothing(+weightedRTT, +rtt);
        sum = 0;
        jitterArray.forEach(element => {
            sum = +sum + +element;
        });
        jitterVal = (jitterArray.length != 0) ? sum / jitterArray.length :
            jitterVal;
```

```
        weightedJitter = exponentialSmoothing(+weightedJitter, +jitterVal);


    });

    //console.log("WRTT: " + weightedRTT + " WJITT: " + weightedJitter);
}

var halflife_up = 3;
var halflife_down = 1.5;
var alpha_up = Math.exp(Math.log(0.5) / halflife_up);
var alpha_down = Math.exp(Math.log(0.5) / halflife_down);

function exponentialSmoothing(weightedInput, input) {
    if (weightedInput <= input)
        return Math.round(weightedInput * alpha_up + (1 - alpha_up) * input);
    else
        return Math.round(weightedInput * alpha_down + (1 - alpha_down) *
            input);
}

var time = 0;
function writeVideoData() {
    document.getElementById("rtt").value = rtt.toFixed(3);
    document.getElementById("jitter").value = jitterVal.toFixed(3);
    document.getElementById("hw").value = hwConcurrency;
    document.getElementById("ram").value = hwMemory;
    document.getElementById("fps").value = weightedFPS;
    document.getElementById("decodedFrames").value = decodedFrames;
    document.getElementById("droppedFrames").value = droppedFrames;
    document.getElementById("decodedBitrate").value = weightedDecodedBit;
    document.getElementById("videoHeight").value = videoPlayer.videoHeight;
    document.getElementById("videoWidth").value = videoPlayer.videoWidth;
    document.getElementById("playerHeight").value = videoPlayer.clientHeight;
    document.getElementById("playerWidth").value = videoPlayer.clientWidth;
    document.getElementById("bitRateLabel").innerHTML = "Requested Bitrate: "
        + getNeuralPrediction();

    rawBRValues.push({
        x: time,
        y: decodedBitrate
    });
    time = time + 1;
}

window.setInterval(() => {
        calcVideoData();
        writeVideoData();
    },
    calcDataHighFrequent);

window.setInterval(() => {
        calcNetworkData();
    },
    calcDataLowFrequent);
```

RTT Calculation:

```
var xhr;
var count_ping = 5;
var pingStatus = 0;
var jitterStatus = 0;
var ptCalled = false;
function pingTest(_callback) {
    if (ptCalled) return;
    else ptCalled = true;
    var startT = new Date().getTime();
    var prevT = null;
    var ping = 0.0;
    var jitter = 0.0;
    var i = 0; // counter of pongs received
    var prevInstspd = 0; // last ping time, used for jitter calculation
    xhr = [];
    rttArray = [];
    jitterArray = [];
    // ping function
    var doPing = function () {
        prevT = new Date().getTime();
        xhr[0] = new XMLHttpRequest();
        xhr[0].onload = function () {
            if (i === 0) {
                prevT = new Date().getTime(); // first pong
            } else {
                var instspd = new Date().getTime() - prevT;
                try {
                    //try to get accurate performance timing using performance
                         api
                    var p = performance.getEntries();
                    p = p[p.length - 1];
                    var d = p.responseStart - p.requestStart;
                    if (d <= 0) d = p.duration;
                    if (d > 0 && d < instspd) instspd = d;
                } catch (e) {
                    //if not possible, keep the estimate
                    console.log("Performance API not supported, using estimate
                        ");
                }
                //noticed that some browsers randomly have 0ms ping
                if (instspd < 1) instspd = prevInstspd;
                if (instspd < 1) instspd = 1;
                var instjitter = Math.abs(instspd - prevInstspd);
                if (i === 1) ping = instspd;
                /* first ping, can't tell jitter yet*/
                else {
                    ping = instspd < ping ? instspd : ping * 0.8 + instspd *
                        0.2;
                    if (i === 2) jitter = instjitter;
                    else jitter = instjitter > jitter ? jitter * 0.3 +
                        instjitter * 0.7 : jitter * 0.8 + instjitter * 0.2;
                }
                prevInstspd = instspd;
            }
            pingStatus = ping.toFixed(3);
            jitterStatus = jitter.toFixed(3);
            i++;
```

```
            if (i < count_ping) {
                doPing();
                if (pingStatus != 0) rttArray.push(pingStatus);
                if (jitterStatus != 0) jitterArray.push(jitterStatus);

            } else {
                _callback();
                ptCalled = false;
                return;
            }
        }.bind(this);
        xhr[0].onerror = function () {
            // a ping failed, cancel test
            console.log("ping failed");
            //abort
            pingStatus = "Fail";
            jitterStatus = "Fail";
            clearRequests();
            console.log("ping test failed, took " + (new Date().getTime() -
                startT) + "ms");
            ptCalled = false;
            return;
        }.bind(this);
        // send xhr
        xhr[0].open("GET", "http://10.1.5.2/empty.js" + url_sep("http
            ://10.1.5.2/empty.js") + "r=" + Math.random(), true); // random
            string to prevent caching
        xhr[0].send();
    }.bind(this);
    doPing(); // start first ping

}

//* stops all XHR activity, aggressively
function clearRequests() {
    if (xhr) {
        for (var i = 0; i < xhr.length; i++) {
            try {
                xhr[i].onprogress = null;
                xhr[i].onload = null;
                xhr[i].onerror = null;
            } catch (e) {}
            try {
                xhr[i].upload.onprogress = null;
                xhr[i].upload.onload = null;
                xhr[i].upload.onerror = null;
            } catch (e) {}
            try {
                xhr[i].abort();
            } catch (e) {}
            try {
                delete xhr[i];
            } catch (e) {}
        }
        xhr = null;
    }
}
```

```
function url_sep(url) {
    return url.match(/\?/) ? "&" : "?";
}
```

Neural Network:

```
var inputTrainingData;
var outputTrainingData;
var model;
var testingData;

function buildNeural() {
    model = tf.sequential();

    //* build neural network
    model.add(tf.layers.dense({
        inputShape: [4],
        activation: "sigmoid",
        units: 100,
    }));

    model.add(tf.layers.dense({
        activation: "sigmoid",
        units: 100,
    }));

    model.add(tf.layers.dense({
        activation: "sigmoid",
        units: 3,
    }));

    model.compile({
        loss: "meanSquaredError",
        optimizer: tf.train.adam(0.001)
    });
}

function getTrainingData() {
    fetch('./DataSet/trainingSet2.json')
        .then(response => {
            return response.json()
        })
        .then(data => {
            inputTrainingData = tf.tensor2d(convertToMatrix(data));
            outputTrainingData = tf.tensor2d(data.map(item => [
                item.BitRate == 5000 ? 1 : 0,
                item.BitRate == 2000 ? 1 : 0,
                item.BitRate == 300 ? 1 : 0,
            ]));
            trainNetwork().then(() => getTestingData());
        })
        .catch(err => {
            console.log(err)
        });
}

async function trainNetwork() {
```

```
    const startTime = Date.now();
    await model.fit(
    inputTrainingData,
    outputTrainingData,
    {
        epochs: 1000,
        shuffle: true,
        callbacks: {onBatchEnd}
    }).then(() => drawLoss(lossRate));
    const endTime = Date.now();
    console.log(endTime - startTime);
    if (confirm('Do you want to save this Model?')) {
        model.save('localstorage://my-model');
    } else {
        // Do nothing!
    }
}

var ii = 0;
var lossRate = [];
function onBatchEnd(batch, logs) {
    lossRate.push({x: ii, y: logs.loss});
    ii++;
}

//* to test our neural network accuracy
function getTestingData() {
    fetch('./DataSet/testingSetTest2.json')
        .then(response => {
            return response.json()
        })
        .then(data => {
            testingData = tf.tensor2d(convertToMatrix(data));
            model.predict(testingData).print();
        })
        .catch(err => {
            console.log(err)
        });
}

function predictBitrate(data){
    var predictedBitrate;
    tf.tidy(() => {
        var inputANN = tf.tensor2d(convertToMatrix(data));
        var results = model.predict(inputANN);
        var index = results.argMax(1).dataSync()[0];
        switch(index){
            case 0:
                predictedBitrate = 5000;
                break;
            case 1:
                predictedBitrate = 2000;
                break;
            default:
                predictedBitrate = 300;
                break;
        }
    });
```

```
    return predictedBitrate;
}

//* convert json data to matrix for neural network input
function convertToMatrix(data) {
    return data.map(item => [
        item.RTT,
        item.DecodedFrames,
        item.FPS,
        item.DecodedBitrate,
    ]);
}
```

LITERATURVERZEICHNIS

[1] Anthony Romero. What is video encoding? codecs and compression techniques, 2018. URL https://www.ibm.com. [Online; accessed November-2019].

[2] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. In *New advances in machine learning*. IntechOpen, 2010.

[3] Chao Chen, Yao-Chung Lin, Anil C. Kokaram, and Steve Benting. Encoding bitrate optimization using playback statistics for http-based adaptive video streaming. *CoRR*, abs/1709.08763, 2017. URL http://arxiv.org/abs/1709.08763.

[4] Emulab contributors. emulab frontpage, 2006. URL https://www.emulab.net/. [Online; accessed November-2019].

[5] Wolfgang Ertel. *Grundkurs künstliche Intelligenz: eine praxisorientierte Einführung*. Springer-Verlag, 2016.

[6] Zoubin Ghahramani. *Unsupervised Learning*, pages 72–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-28650-9. doi: 10.1007/978-3-540-28650-9_5. URL https://doi.org/10.1007/978-3-540-28650-9_5.

[7] Google contributors. Recommended settings for vod, 2017. URL https://developers.google.com/. [Online; accessed November-2019].

[8] Google contributors, 2018. URL https://www.tensorflow.org/. [Online; accessed November-2019].

[9] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In José Mira and Francisco Sandoval, editors, *From Natural to Artificial Neural Computation*, pages 195–201, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49288-7.

[10] iperf, 2014. URL https://software.es.net/iperf/. [Online; accessed November-2019].

[11] IPFW contributors. Ipfw, 2006. URL https://www.freebsd.org/doc/handbook/firewalls-ipfw.html. [Online; accessed December-2019].

[12] Jan Ozer. How to produce for adaptive streaming, 2012. URL https://www.streamingmedia.com/. [Online; accessed November-2019].

[13] J. Jiang, T. Fogal, C. Woolley, and P. Messmer. A lightweight h.264-based hardware accelerated image compression library. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 99–100, Oct 2016. doi: 10.1109/LDAV.2016.7874337.

[14] Chenghao Liu, Imed Bouazizi, and Moncef Gabbouj. Rate adaptation for adaptive http streaming. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pages 169–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0518-1. doi: 10.1145/1943552.1943575. URL http://doi.acm.org/10.1145/1943552.1943575.

[15] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 197–210, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4653-5. doi: 10.1145/3098822.3098843. URL http://doi.acm.org/10.1145/3098822.3098843.

[16] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997. ISSN 0146-4833. doi: 10.1145/263932.264023. URL http://doi.acm.org/10.1145/263932.264023.

[17] MDN contributors. Getting started with webgl, 2019. URL https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL. [Online; accessed November-2019].

[18] Yanyuan Qin, Shuai Hao, K. R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. Abr streaming of vbr-encoded videos: Characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 366–378, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6080-7. doi: 10.1145/3281411.3281439. URL http://doi.acm.org/10.1145/3281411.3281439.

[19] SIMON KEMP. Digital 2019: Global internet use accelerates, 2019. URL https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates. [Online; accessed November-2019].

[20] Matthias Ueberheide, Felix Klose, Tilak Varisetty, Markus Fidler, and Marcus Magnor. Web-based interactive free-viewpoint streaming: A framework for high quality interactive free viewpoint navigation. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 1031–1034, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3459-4. doi: 10.1145/2733373.2806394. URL http://doi.acm.org/10.1145/2733373.2806394.

[21] Tilak Varisetty and David Dietrich. Client-side bandwidth estimation technique for adaptive streaming of a browser based free-viewpoint application. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, HotEdgeVideo'19, pages 39–44, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6928-2. doi: 10.1145/3349614.3356021. URL http://doi.acm.org/10.1145/3349614.3356021.

[22] Wikipedia contributors. Streaming media — Wikipedia, the free encyclopedia, 2019. URL https://en.wikipedia.org/w/index.php?title=Streaming_media&oldid=928251284. [Online; accessed 28-November-2019].

[23] Xing Yan, Lei Yang, Shanzhen Lan, and Xiaolong Tong. Application of html5 multimedia. In *2012 International Conference on Computer Science and Information Processing (CSIP)*, pages 871–874, Aug 2012. doi: 10.1109/CSIP.2012.6308992.

[24] B. Zhou, J. Wang, Z. Zou, and J. Wen. Bandwidth estimation and rate adaptation in http streaming. In *2012 International Conference on Computing, Networking and Communications (ICNC)*, pages 734–738, Jan 2012. doi: 10.1109/ICCNC.2012.6167520.