TÉCNICO LISBOA

Neural Network Predictor for Fast Channel Change on Set-Top-Boxes

Tomás Santos Azenhas Boavida Malcata

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Tiago Miguel Braga da Silva Dias
Prof. Nuno Filipe Valentim Roma

Examination Committee

Chairperson: Prof. Pedro Filipe Zeferino Aidos Tomás
Supervisor: Prof. Tiago Miguel Braga da Silva Dias
Member of the Committee: Prof. António Manuel Raminhos Cordeiro Grilo

November 2022

Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank to professors, Nuno Roma, Tiago Dias and Nuno Sebastião for all the knowledge transfer, discussions, guidance and support that they provided me from the moment I started this dissertation until now.

I would like to thank all my family, my parents, my sister, and my brother for all the support during my dissertation, and also throughout my whole degree studies because they made this journey easier. I would also like to thank to all my friends who I have met during the degree. Finally, a special thank you to Mariana who always helped me.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/CEC/50021/2020 and PTDC/EEI-HAC/30485/2017.

Abstract

The Television (TV) channel change delay is measured from the moment a user presses a button to change a channel until that channel becomes visible. With the launching of digital television, not only the channel quality improved but also the amount of available channels increased, which resulted in a great increase in the channel change delay to about 2 s [1]. This thesis proposes a predictive system to assist in predicting the channels users will see next and, thus, reduce the channel change delay.

The proposed solution consists of a Recurrent Neural Network (RNN) that processes data concerning the channel changes history from a user to predict the next channels to be displayed. This RNN was designed to be implemented in a Set-top-Box (STB) during training and inference, after refining the best hyperparameter combination for the RNN that best matches the constraints imposed by a STB implementation, the proposed RNN requires only 401 KB of disk space with its library included and 2.4 MB of Random Access Memory (RAM). The devised model was tested using four embedded systems with configurations very similar to a real STB, and the obtained experimental results show an accuracy of the RNN predictor of 50.2 % for a single channel prediction and 67.7 % for five channels predicted simultaneously. Such results correspond to, on average, 0.26 hours saved per user, during two months.

Keywords

Digital Television; Channel Change Delay; Predictive System; Machine Learning; Recurrent Neural Network; Embedded Systems

Resumo

O tempo de transição entre canais é medido desde que o utilizador carrega para selecionar o novo canal de televisão até que o utilizador começa a ver esse canal no seu equipamento. Com o aparecimento da televisão digital, a qualidade visual e a quantidade de canais recebidos aumentou. Contudo, estas mudanças também impactaram o tempo de mudança de canal. Este aumentou para 2 s, o que é considerado excesivo para se conseguir uma boa experiência de utilização. Nesta tese é proposto um sistema de previsão para utilização em Set-top-Boxes (STBs) com o objetivo de prever o canal que o utilizador verá a seguir e, dessa forma, reduzir o tempo de transição entre canais.

Este trabalho incluiu o desenvovimento de uma Recurrent Neural Network (RNN) com base nos registos de mudança de canal do utilizador para prever o canal seguinte. Esta RNN foi construída de modo a funcionar sempre na Set-top-Box (STB), tanto quando o modelo está a prever novos canais como quando está a treinar. Após encontrar a melhor combinação de hiperparâmetros, verificou-se que a sua implementação requer apenas 401 KB de espaço em disco e um máximo de 2.4 MB de Random Access Memory (RAM). O modelo foi implementado em quatro sistemas embebidos com características semelhantes a uma STB. Os resultados mostram uma exatidão de 50.2 % para a previsão de um só canal e 67.7 % para a previsão de 5 canais. Estes valores correspondem, em média, a uma redução de 0.13 horas mensais por utilizador no tempo perdido entre mudanças de canais.

Palavras Chave

Televisão Digital; Atraso na Mudança de Canal; Sistema de Predição; Aprendizagem Automática; Rede Neuronal Recorrente; Sistemas Embebidos

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Thesis objectives and restrictions . . . 4

1.3 Thesis outline . . . 5

2 State of the Art **7**

2.1 Current solutions for mitigating channel change delay . . . 7

2.2 Adjacent group join-leave method . . . 8

2.3 Surfing behaviour preference . . . 10

2.4 Approaches using other users' information . . . 11

2.5 Machine learning approach . . . 12

3 Neural networks overview **15**

3.1 Neural network history . . . 16

3.2 Important concepts . . . 16

3.2.1 Perceptron . . . 16

3.2.2 Activation functions . . . 17

3.2.3 Deep learning . . . 19

3.2.4 Recurrent neural network . . . 20

3.2.5 Backpropagation and training . . . 21

3.3 Neural network types and usages . . . 23

3.3.1 Convolutional neural networks . . . 23

3.3.2 Vanilla recurrent neural network . . . 23

3.3.3 Long short-term memory networks . . . 24

3.3.4 Gated recurrent unit network . . . 25

3.4 Frameworks for neural networks . . . 26

4 Proposed solution **29**

4.1 Data . . . 30

4.1.1 Data collection . . . 31

4.1.2 Data processing . . . 31
4.2 Neural network approach . . . 33
4.2.1 Framework . . . 33
4.2.2 Inputs . . . 33
4.2.3 Hyperparameters . . . 34
4.3 Channel mapping . . . 36
4.4 Model output . . . 37
4.5 Training . . . 38
4.6 Model evaluation . . . 40
4.7 Model Implementation . . . 40
4.7.1 Model generation . . . 41
4.7.2 Training . . . 42

5 Datasets and refinement 45

5.1 Data analysis . . . 45
5.1.1 Synthetic data . . . 46
5.1.2 Real users' data . . . 47
5.1.3 Best 30 and 270 users . . . 48
5.2 Hyperparameter selection . . . 49
5.2.1 Dropout rate . . . 50
5.2.2 Maximum root mean square error . . . 52
5.2.3 Layers . . . 53
5.2.4 Learning rate . . . 54
5.2.5 Network type . . . 55
5.2.6 Unroll length . . . 56
5.2.7 Weeks used to train . . . 57
5.2.8 Other hyperparameters and decisions . . . 58
5.2.9 Summary . . . 59

6 Results and analysis 61

6.1 User simulation . . . 61
6.2 Individual results . . . 62
6.2.1 Artificial user . . . 63
6.2.2 Best user . . . 64
6.2.3 Wort user . . . 65
6.3 General results . . . 66
6.4 Validation on embedded devices . . . 68

6.5 Overall results and comparison with other models 69

7 Conclusion and Future Work **73**

7.1 Conclusion 73

7.2 Future work 74

Bibliography **75**

A Code of Project **81**

B User's prediction graphics. **83**

B.1 Artificial User 83

B.2 Best User 86

B.3 Worst User 88

List of Figures

1.1 Channel change delay phases. Inspired in [2]. 2
1.2 Time used for channel change by each phase. Inspired in [2]. 3

3.1 Perceptron schema. 17
3.2 Example of a feedforward Neural Network (NN) with one input layer, three hidden layers and one output layer. 19
3.3 Recurrent Neural Network (RNN) with one layer and with only one neuron. This RNN is then unfolded to an equivalent feedforward NN. Removed from [3] 20
3.4 An Long Short Term Memory (LSTM) cell, as shown in [4]. 24
3.5 A Gated Recurrent Unit (GRU) cell, as shown in [4]. 25

4.1 Process diagram of how a new channel command is processed. 30

5.1 Dispersion graphic of the real users by number of different channels watched and number of channel changes. 47
5.2 Users dispersion of the best 30 users group. 48
5.3 Results for the variation of the dropout rate. 51
5.4 Results for the variation of the minimum Root Mean Square Error (RMSE) 53
5.5 Results for the variation of the number of layers. 54
5.6 Results for the variation of the learning rate. 55
5.7 Results for the variation of the RNN type. 56
5.8 Results for the variation of the unroll length. 57
5.9 Results for the variation of the number of weeks used to train. 58

6.1 Heat map regarding the accuracy for one channel, number of channels watched and number of channel changes. 66
6.2 Normal curves representing the dispersion of the accuracy and correspondent channel change delay for all the users. 67

B.1 Error loss plot during training for the artificial user. 83
B.2 Channel changes performed by the artificial user on the first week. 84
B.3 Channel changes by the artificial user and model prediction for the second week. 84
B.4 Channel changes by the artificial user and model prediction for the third week. 84
B.5 Channel changes by the artificial user and model prediction for the fourth week. 84
B.6 Channel changes by the artificial user and model prediction for the fifth week. 84
B.7 Channel changes by the artificial user and model prediction for the sixth week. 85
B.8 Channel changes by the artificial user and model prediction for the seventh week. 85
B.9 Channel changes by the artificial user and model prediction for the eighth week. 85
B.10 Error loss plot during training for the best user. 86
B.11 Channel changes performed by the best user on the first week. 86
B.12 Channel changes by the best user and model prediction for the second week. 86
B.13 Channel changes by the best user and model prediction for the third week. 87
B.14 Channel changes by the best user and model prediction for the fourth week. 87
B.15 Channel changes by the best user and model prediction for the fifth week. 87
B.16 Channel changes by the best user and model prediction for the sixth week. 87
B.17 Channel changes by the best user and model prediction for the seventh week. 87
B.18 Error loss plot during training for the worst user. 88
B.19 Channel changes performed by the worst user on the first week. 88
B.20 Channel changes by the worst user and model prediction for the second week. 88
B.21 Channel changes by the worst user and model prediction for the third week. 89
B.22 Channel changes by the worst user and model prediction for the fourth week. 89
B.23 Channel changes by the worst user and model prediction for the fifth week. 89
B.24 Channel changes by the worst user and model prediction for the sixth week. 89
B.25 Channel changes by the worst user and model prediction for the seventh week. 89
B.26 Channel changes by the worst user and model prediction for the eighth week. 90

List of Tables

2.1 Results for the implementation of the up-and-down plus previous channel predictive model on this thesis dataset. 9
2.2 Performance results obtained in [5]. 12

4.1 Example of data saved in the logs for a specific user. 31
4.2 Example of data after being processed. Fields marked with * are represented as one hot encoding, which was suppressed for readability. 31
4.3 Example of the output of the model for an user with just five channels and the respective two most likely classes to be watched next. Each class corresponds to a channel since it has been mapped. 38

5.1 Channel schedule for the artificial user. 46
5.2 Final hyperparameter values. 59

6.1 Accuracies obtained for the artificial user per week and train times measured in the Raspberry Pi 3. 63
6.2 Accuracies obtained for the best user per week and train times measured in the Raspberry Pi 3. 65
6.3 Accuracies obtained for the worst user per week and train times measured in the Raspberry Pi 3. 65
6.4 Training times in different embedded devices. 69
6.5 Accuracy results comparison between the model proposed in this thesis and the ones defined in the state of art. 70
6.6 Overall results. 71

List of Algorithms

4.1 Handling new channel occurrences before model training. 37

6.1 Simulation for one user. 62

Listings

4.1 Model generation code. 42
A.1 Code used to train the RNN 81

Acronyms

AM	Amplitude Modulation
API	Application Program Interface
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
CNNs	Convolutional Neural Networks
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DVB	Digital Video Broadcasting
EPG	Electronic Programme Guide
FM	Frequency Modulation
GOP	Group of Pictures
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
GRUs	Gated Recurrent Units
HDD	Hard Disk Drive
HD	High Definition
IPTV	Internet Protocol Television
IP	Internet Protocol
LMT	Logistic Model Tree
LSTM	Long Short Term Memory
LSTMs	Long Short Term Memory
MLP	Multilayer Perceptron

ML	Machine Learning
MPEG	Moving Picture Expert Group
MPEG	Moving Picture Experts Group
NN	Neural Network
NNs	Neural Networks
OS	Operating System
PVR	Personal Video Recorder
QoE	Quality Of Experience
RAM	Random Access Memory
RGB	Red-Green-Blue
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
RNNs	Recurrent Neural Networks
SMP	Semi-Markov Process
SSE	Sum of Squared Errors
STB	Set-top-Box
STBs	Set-top-Boxes
TV	Television
UDP	User Datagram Protocol

1

Introduction

1.1 Motivation

After several tentatives and experiences, Television (TV) started to be gradually adopted as a new medium of entertainment in 1925. At that time, TV systems were already based on electronics and employed analogue technology, the original TV technology. When the TV started to become commercialised and more used around the globe, it was primarily used analogue TV. In analogue TV, all the channels are provided using different Amplitude Modulation (AM) and Frequency Modulation (FM) signals, AM for the image and FM for the sound at the same time. Consequently, if a user wants to change to another channel, the TV set is required to tune the corresponding AM and FM frequencies. Using this technology, the delay time for changing a channel, which is known as zapping time, was around 200 ms [6], which is considered by users as an "instantaneous" channel switch [7].

The digital TV, such as Digital Video Broadcasting (DVB) and Internet Protocol Television (IPTV), latter appeared to replace the analogue TV. It has the main advantage of providing the standard definition channels already available in analogue TV but with superior quality both in terms of image and sound and even provides High Definition (HD) channels. Furthermore, it can integrate with the internet and the

user can use more functionalities than simply watching the channel, such as an Electronic Programme Guide (EPG) or a Personal Video Recorder (PVR), in addition, it allows easy integration with Internet Protocol (IP) based services. However, with the implementation of such functionalities and the change of the content delivery method, the zapping time has increased substantially from a delay of 200 ms [6] to a delay between 0.9 s and 70 s in some IPTV configurations [7] or 2.6 s in DVB [8]. Yet, based on the user experience, an acceptable Quality Of Experience (QoE) needs to have the zapping time lower than 430 ms to be deemed acceptable by the users [9]. Note that this delay time for channel switching is measured from the moment the user sends a new channel request by pressing the control remote until the time the user starts watching the new channel on the TV.

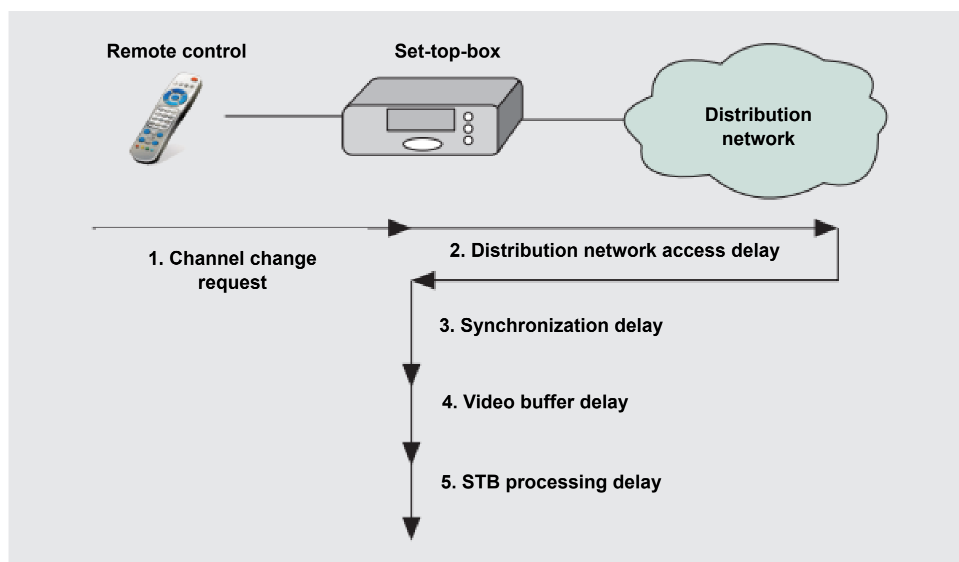


Figure 1.1: Channel change delay phases. Inspired in [2].

This operation can be divided into five different moments as depicted in Figure 1.1, which starts from the moment the user sends the remote request for a channel change to the moment the user starts watching the new channel on the TV.

Figure 1.1 also depicts the five most important contributions to the channel change delay. The channel change request delay corresponds to the time between the user sending the request and the Set-top-Box (STB) receiving it. The distribution access delay corresponds to the time difference from the moment that the STB queries the server through the network to start receiving the new channels and stop receiving the old ones until that response arrives.

The synchronisation delay corresponds to the time required to obtain the first frame to be displayed from the received bitstream and it varies according to the encoding format. In the most common video coding schemes, such as Moving Picture Experts Group (MPEG) version 2 and 4, three types of video frames are used to encode the video: I, B and P-frames. I-frames include all the data required to decode

and display a new frame. Conversely, B and P-frames depend on the information from other frames to be decoded, since the encoding process exploits the similarities in previous frames to improve the coding efficiency. Such sets of dependent frames are grouped together in Group of Pictures (GOP), which are usually composed of an initial I-frame followed by intercalated B and P-frames. To start displaying a new channel, the system needs to reach an I-frame of a GOP. The synchronisation delay is the time required to retrieve such frame [2].

For transporting data it is commonly used User Datagram Protocol (UDP) which has the advantage of being faster, allowing multicast and reducing latency. However, this transportation mechanism is not reliant and some datagrams can be lost. To complement the package losses it is commonly used a buffer to repair them and retransmit those datagrams. These buffers need to be filled before the stream is ready, which creates the fourth delay phase, the buffering delay.

Finally, the STB processing delay corresponds to the time required by the STB and the TV to be ready to display the new channel. Obviously, this delay depends on the device's processing speed.

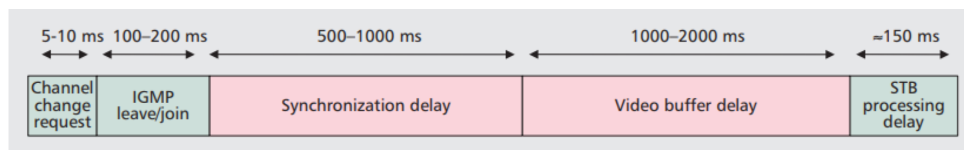


Figure 1.2: Time used for channel change by each phase. Inspired in [2].

Figure 1.2 presents typical times for each delay component in IPTV. As can be seen, the synchronisation delay and the buffering delay are the most relevant parts of the channel change delay on an IPTV architecture. This is why the usual delay time for a channel change is about 2 s [1]. It is also important to note that if the STB is already receiving the desired channels, the synchronisation and the buffer filling were already performed, making the delay time go to virtually 0 s [1].

DVB systems (Satellite/Cable/Terrestrial) also suffer from the same type of delays, which result in zapping times of about 2.6 s [8]. In DVB, the channel change request time is similar to IPTV and the tuning time is also not relevant portion to the whole process [8]. Conversely, the synchronisation and buffering delay also have a significant impact on the overall time [1].

In both architectures (DVB, IPTV), fetching several channels simultaneously may reduce the zapping time to 0 s [1], but requires more bandwidth, which may not be feasible. Moreover, on a DVB schema, the maximum amount of channels possible to request is also influenced by the number of decoders and tuners available. Typically. a STB includes between two and four decoders/tuners.

1.2 Thesis objectives and restrictions

The main goal of this research work was to investigate a Machine Learning (ML) based channel prediction model to improve the zapping time in digital TV. The requirements for such a model are the following: provide good predictions for the TV channel a user is most likely to be watching once the user changes channel; easily adapt to the characteristics of different users; operate independently of the TV configuration and for different technologies (e.g. DVB and IPTV); suitable for being implemented in computational and memory constrained devices like a STB.

Designing a ML that can have its whole lifecycle inside the STB that a user already has at his/her home was a key requirement for this work since this approach allows not only to have an independent model for each user but also to ease the integration of this solution in current TV architectures. Accordingly, training and inference should be carried out in this environment, where the logs of the channel history are expected to be stored (and currently are). Yet, the computational power and the memory (both Random Access Memory (RAM) and disk space) available in Set-top-Boxes (STBs) are known to be quite modest.

STBs have installed an embedded Linux operating system [10], and its memory specifications vary according to the offered functionalities.The primary purpose of a STB is decoding the MPEG video frames, loading the buffers for packet loss recovery, communicating with the content provider, and displaying the channels for the user. Moreover, it is responsible for secondary functions such as displaying the EPG, allowing video recording and video-on-demand services (for example, streaming platforms). Therefore, nowadays, STB are being developed with more memory available, not only in terms of flash and RAM, but also for the Hard Disk Drive (HDD) so that the user can store more video content and access the channels faster. Nonetheless, is not public how much memory is available on the disk or RAM after the applications are launched. However, some argue that such values will be around 10 MB each. These values can change for different STB models throughout the day, especially for the RAM, since the RAM can be higher when the user is not watching TV. For example, the DCX3510-M HD Dual Tuner DVR Set-Top [11] specialised in HD video and storing content has 512 MB Dynamic Random Access Memory (DRAM), 64 MB of flash and a 500 GB HDD.

In order to simulate a STB it is commonly used a Raspberry Pi [12] [13] [14]. In this work, a Raspberry Pi 3 Model B Rev 1.2 with 1 GB of RAM and a quad-core ARM Cortex-A53 Central Processing Unit (CPU) operating at 1.2 GHz (ARMv8) was used to simulate a STB. Other embedded systems were also used to do performance tests.

1.3 Thesis outline

This thesis is organised into eight chapters. In Chapter 1, the zapping problem is introduced and the main goals for this research work are presented. The Chapter 2 discusses the existing studies and solutions to solve the zapping delay problem are presented, from the simple up and down model to the more complex ML approaches. Chapter 3 briefly reviews the fundaments of Neural Network (NN) and the most important concepts for this work. Also, the most prominent related work is discussed. Chapter 4 describes the idea underlying the proposed model and Section 4.7 describes the implementation of such model using the C programming language and the Kann framework. The dataset considered in this work is presented in Chapter 5, along with the adopted criteria to choose the hyperparameters for the devised Recurrent Neural Network (RNN). Chapter 6 presents the considered experimental setup, including the simulation environment based on several embedded systems, and discusses the obtained results in terms of accuracy and performance for the general dataset and some specific users. Finally, Chapter 7 presents the conclusions of this research work and draws some ideas for future work.

2

State of the Art

This Chapter mainly explains the current solutions and studies to mitigate the zapping delay. It focuses primarily on the schemes that mitigate the zapping delay by creating a predictive heuristic to find the channel the user will watch. It starts with the most common and straightforward prediction model and ends with the most complex heuristics using ML based schemes.

2.1 Current solutions for mitigating channel change delay

In digital TV, two approaches are commonly used to reduce the zapping delay problem and improve the QoE for the user. The first one consists in optimising the system architecture, the connection to the servers, and the involved protocols, therefore, reducing the zapping time [5]. The other approach consists in predicting the channel that is most likely to be watching when selecting a new channel and performing all the necessary operations supporting such channel change with an almost irrelevant delay while the user is still watching the previous channel.

On the architectural side, the improvements can be within two levels. On the network level, to create conditions for the STB buffers to be filled faster, which implies the stream decoding to be faster. It

may require more bandwidth usage during the transmission or it can be accomplished using companion streams which also require more bandwidth and STB resources to use them. On the decoding side, to reduce the time needed to start displaying a new channel, it is possible to create more random access points which can be accomplished either by adding extra I-frames in the GOP or by reducing the GOP size and therefore having more I-frames in the GOP. Having more I-frames allows the decoding to be faster since it reaches an I-frame quicker. [15] [16] Even though these proposals have been able to reduce the zapping time by around 60 % with only the cost of some extra bandwidth, there is a problem with this solution, the change in the architecture (to send the companion streams or extra I-frames) would imply changes in the provider systems, further, the existing STB would have to be prepared to used those extra resources.

The predictive approach has the advantage that the heuristic developed to find the channel the user will see next is not dependent on the architecture of the TV provider (either if it is IPTV, DVB or other). By predicting the channel correctly it is possible to fill in the buffers and do their synchronisation in the background [1], which greatly reduces the buffering delay, as shown in Figure 1.2. In fact, this approach allows reducing the delay to below 0.43 s which is not perceptible by the users [7]. If the method fails to predict the channel, the user will experience an average delay. The main drawback of this solution concerns the bandwidth consumption and resources needed, which increases with the number of predicted channels. This research work considers the predictive method to reduce the zapping delay in digital TV.

2.2 Adjacent group join-leave method

The first predictive model for shortening the zapping delay was proposed in 2004 [17]. It created a scheme that allowed the IPTV system to receive more than one channel (3 channels in this case) simultaneously, effectively reducing the zapping time. In this scheme, once the user changes to a new channel, the STB queries the server for the next and previous channel in the channel grid. Therefore, if the user is doing a linear zapping (especially using the arrow key on the TV remote control), he/she will not experience any delay since the channel will already be ready to be displayed. It should be noted that as new channels are asked to the server, the ones that are being received and become at a distance greater than one channel from the displayed channel are stopped from being received by the STB to save bandwidth. Since some TV remote controllers also had a button to select the last seen channel, some models also received the previously displayed channel. Accordingly, some combinations were studied between these possible scenarios to save bandwidth.

Although this scheme was presented without an accuracy evaluation [17], it was later evaluated using data from 25500 users for 150 different channels [2]. It was observed that 55 % of the channel changes

were to adjacent channels. Furthermore, it was concluded that a user usually does not change the channel number range very drastically, since in 80 % of the cases the user does not change the channel to a distance greater than six channels.

Another study presented in [1] addressed for how long the adjacent channels should be fetched. It is shown that if the two adjacent channels are always brought, the percentage of channel changes without delay is 55 %. However, if the channel is only fetched during one minute after the channel change, the user will still have 44 % of the channel change requests without delay time. With this mechanism, the trade-off between accuracy and zapping delay and bandwidth consumption is performed. It is also possible to see that the accuracy (and the decreasing of the zapping delay) can be increased by fetching more than just the closest neighbour channels, but the bandwidth consumption also increases.

On the data used in this project, described in Chapter 6, the up and down model plus the previous channel was performed, and its results can be seen in Table 2.1. It is visible that the results were not the same as observed in [1] [2]. However, the dataset is different. With these results, it is possible to compare the results on the same dataset with the results that this project will present.

Table 2.1: Results for the implementation of the up-and-down plus previous channel predictive model on this thesis dataset.

Number of predicted Channels	1			2			3
Type of channels Fetched	Up	Down	Previous	Up + Down	Up + Previous	Down + Previous	Up + down + previous
Accuracies (%)	12.2	7.7	11.3	19.9	23.5	19.0	31.2

A significant advantage of this approach is the simplicity of its implementation and not requiring memory to store the model data or to execute it. This approach also does not need any training or data to work. Furthermore, it does not need to be recompiled if a new channel appears. The drawback is that it requires that the user's primary interface with the STB is the remote control's up and down buttons, which is not always true. In addition, the constant data requested to the server for the current channel and the adjacent ones consume a significant amount of bandwidth, especially if more than the adjoining neighbour channels are considered. Another disadvantage is not considering any user behaviour pattern since it does not take into account the day of the week or the time of the day.

The main takeaway from this approach is the importance of knowing the channel being displayed to better predict the next channel to be displayed since with only the current channel as input to the model is possible to successfully predict in 55 % of the cases the channel that the user will see next.

2.3 Surfing behaviour preference

A method essentially based on the user's surfing preference was first proposed in [18]. In this model, the user preferences are also considered, and the user should store them in a favourite channel list, accessed by the favourite button. This method assumes that a user can switch channels using four different surfing formats, the up and down button, the previous channels button, the favourite button and the numbers. It also assumes that a user tends to have a surfing pattern where the user repeats the buttons.

The next channel prediction by the STB is made according to the last button the user used to change a channel. If the user uses the up or down button, the STB will also require the adjacent up or down channel, respectively, of the current channel. If the user uses the previous button, the last channel will be fetched, and the next favourite channel (a list of channels defined previously by the user) will be fetched for the favourite button. If the user presses a number button, the preferred channel from the favourite channel list will be predicted.

The proposed model considers all the surfing behaviours and was shown to provide high prediction accuracy. However, the data used to assess it was not real data. In fact, it was generated by a simulation that may not reflect an exact channel change pattern. Furthermore, this approach requires a specific type of STB with an EPG installed since it also needs to store the favourite channels for the user. As discussed in Section 2.2, the use of the previous channel information and how did the user change the channel reinforces the premise that the last channel visited by the user strongly influences the channel that will be seen next. However, it will only work if the user always uses the same remote control button, which in most cases does not occur.

A variation of this solution was explored in [19] which considers how many channels users usually switch in a row until they settle for a specific channel and which channels users see in such a transitional period. For dealing with the number of channels watched in a row, a purely mathematical Semi-Markov Process (SMP) was created. It used the same categories connected to the remote control to find which channels the user will see, as in [18]. The SMP states are used while the user is changing channels. The states increase while the user changes more channels sequentially until he/she watches a channel for a sufficiently long period to reset the model to its initial state. Then, the number of fetched channels is classified into two groups, the ones that were brought in the background while viewing a channel and the ones that were brought while surfing to find a channel to see.

This model has the main drawback of being purely theoretical and not considering the existence of outliers. Furthermore, it always tries to fetch five channels when the user watches a channel which has a high bandwidth cost. Moreover, it considers that the user patterns do not change and that the user's favourite channels are constantly updated. Finally, the probabilities used by the model for the percentage of times that a user uses each channel switching button or the usual number of channels

that the user changes until reaching the final destination may become outdated or vary from region to region and from user to user.

Other research on this methodology [20] starts by fetching the neighbour channels. Then, as soon as the user starts pressing a channel number in the remote control, the fetching of the neighbour channels is stopped until the new channel number is settled and sent to the STB. The model exploits the latency of this operation to predict the next channel to be displayed. When the user presses a number, a pool of channels is created with the channels that start with that digit. Once the user presses the second key, this pool becomes even narrower. For each iteration, it is chosen the maximum number of channels to the available bandwidth and bandwidth consumption per channel, and that amount of channels is fetched. This model shows a promising advance in terms of zapping delay on average. However, it requires a large amount of bandwidth to work with one or two keys pressed (since with two keys, excluding the current and the adjacent channels, it has a pool of 11 channels).

These researches suffer from the lack of practical results since they were performed using simulated data to create the channel change history. They also require many channels to be fetched simultaneously, which is not feasible for most STB. Furthermore, it requires a large amount of user information such as the remote control type and the preferred channels that need to be configured. The final results show that the models perform worse than the up and down mode, although the channels are to be fetched for a less amount of time.

2.4 Approaches using other users' information

The popularity of channels among the general population was also studied as another heuristic for fetching new channels for a given user. In [21] and [22], it is shown that the channels can be represented in a Zipf-like distribution grouped by popularity. It was also shown that this is sustainable by separating the data at different times of the day and other geographic locations. Also, a k-mean cluster algorithm was employed to divide the population between the subset of channel types (by categories such as sport, cinema, kids, ...) and create subsets of the most popular channels. This was later used in [23] to create a predictive channel system based on recording the TV channels by their popularity.

Even though there were promising results in [21] [23], it was later performed another experience [1] where the seven most popular channels were always fetched which showed low accuracy results. This evaluation was performed using the same dataset as in the up and down model [2], and it only had 15 % accuracy. Furthermore, if the two neighbour channels were fetched, it would have 45 % of accuracy. If these two channels were added to the seven most popular as the model prediction, the accuracy would only increase to 51 %. This modest increase of 6 % with the extra effort of having fetched seven extra channels shows that this approach is not very efficient and, therefore, was not pursued in this research.

Table 2.2: Performance results obtained in [5].

Method	Accuracy [%]	Kappa statistic	Tree size	Leaves
Random Forest	29.48	0.2786	NA	NA
LMT	32.79	0.3117	155	78
J48 (default C,M)	34.58	0.3298	663	332
J48 (default C,M)	35.76	0.3406	207	104
J48 (default C,M)	37.39	0.3580	263	132
RepTree	34.14	0.3350	235	NA
Random Committee	36.32	0.3457	211	NA
AdaBoostM1 (J48, default C, M)	35.20	0.3364	669	335
ADABoostM1 (J48, optimized C, M)	37.39	0.3580	263	132
Multilayer Perceptron (MLP)	6.89	NA	NA	NA

2.5 Machine learning approach

A more complex approach for the predictive problem was proposed in [5]. This research work suggests that a purely ML based approach is possible to be implemented with data from the channel history of the users. Accordingly, a set of ML models were created, trained and analysed using synthetic data based on [21]. These models prediction was intended to be used together with the up and down model. Therefore, the STB in every channel change required the next and previous channel along with the channel predicted by the ML model. The accuracies of the models alone (without the up and down model) are presented in Table 2.2.

In the presented model, the channel change log is saved once a channel change is requested. These logs are used for training the model. The model is updated once it is considered outdated or changes to the channel order are performed (and therefore the channel classes become unusable). Once the model needs to be updated, the old one is discarded, and a new one is created with the history log stored. For training the models a ML framework named Weka [24] was used externally to the STB.

The ML models experimented with were mostly tree-based models. First, a random forest classification model was used. This classification model is based on creating several trees, each using different data for the training. The data for the training of each tree is chosen by using Bagging and randomising. From the primary dataset, random samples are extracted (with a replacement on the original dataset) to a new smaller dataset. Then, each tree will have its small bag of datasets to be trained. For inferring, each tree produces an output and the most common class is chosen as the output of the random forest. This technique avoids overfitting in small datasets. In this case, trees were a classification model with one class per channel as the output class, and the input was only information from the channel log. It is not explained how the data treatment was performed. The absolute accuracy was 29 %.

J48 is an implementation of the C4.5 algorithm in the Weka software. This algorithm creates a

classification tree from a specific dataset. A classification tree or decision is a classification model that has a tree structure, wherein at each node of the tree, a decision regarding the input (such as time of the day or month, for example) is taken until it reaches the leaves. Each leaf represents an output class in a classification tree. The Weka implementation also allows the pruning of the tree to reduce the tree dimension and to have the model less overfitted to the training dataset. The objects per leaf and confidence factor are used as hyperparameters and, as seen in Table 2.2, the best combination produced an accuracy of 37.39 %. Another method used to create the tree was the REPTree, which provided a 37.39 % accuracy.

Logistic Model Tree (LMT) is also a tree classifier that combines the tree classification model with the logistic regression. Regarding the tree-based model booster, it was the Random Committee, a pruning mechanism added on top of the created tree to reduce its dimension. Also, AdaBoost, a booster used in training, gives more relevance to weak learners and punishes misclassified samples.

The results presented in [5] are shown in Table 2.2. Such results outperformed the adjacent model for all tree-based models. The main drawback of this solution is that tree-based models require reconstruction from scratch to train the model again. The training process can take a long time, depending on the number of channels in history. Furthermore, the memory requirements can be overwhelming for the STB if the training is done in the STB itself. Conversely, having a central server training the models for each user not only reduces the STB storage requirements but also enables using high level software such as Weka. However, the computational power needed for a central server to train all the models would be tremendous. It should be noted that the computational cost of having this model is not shown in [5] and that the experience was performed with a simulated dataset, which does not consider possible outliers.

The MLP, which is a simple NN, was the only non-tree-based method used in [5]. However, it performed very poorly, providing an accuracy of only 6.69 %. This may be because the considered dataset was fabricated. Furthermore, it is not known how the data is fed in the MLP, so it is impossible to take many more conclusions. However, the MLP and the tree-based models are non-recurrent, so unlike the up and down model and some other ML models do not take into consideration the history of channels when predicting an output (which was shown very promising).

On a different note, a NN based model was created on [25] for a recommender system using a RNN, more specifically, a Long Short Term Memory (LSTM). The main difference between recommender systems and the predictive model is that the purpose of the model is not to decrease the delay but to assist the user in choosing the next channel. The recommender system provides the user with the hot (popular) and cold (unpopular) channels that the user is likely to see. However, the predicted channels are not fetched by the STB. Similar to the previous models, the information used for this recommendation system is the channel zapping history, so the dataset should be very similar to previous experiences (but

in this case, from real users). Furthermore, this LSTM was intended to provide live recommendations, i.e. while the user is watching and changing channels, such as most of the previously described predictive models.

The benefit of having a many-to-one RNN and the influence of its unroll length size is reinforced [25], which shows the importance of predicting the next channel based on the ones previously watched. At last, the final results were encouraging, since this recommendation system, provided an accuracy of 25.5 % when the chosen channel was in the top one and 80.86 % when it was in the top 5 of popularity. Even though these values can not be mapped literally into a predictive system accuracy since the model is only active for sessions where the gap between channel changes is less than 20 s, the solution (with few changes) can be adapted into a predictive next channel model and implemented for each user.

3

Neural networks overview

A NN is a machine learning technique based on the human brain and, consequently, its components are named neurons [26]. Like the human brain, a NN can acquire knowledge by learning with experience, storing that information, and using it with new data to create new predictions based on what it learned [27]. This technology has solved many prediction problems since NN with at least one layer can approximate any mathematical function [28].

According to [26], the main advantage of NN is that its output function does not need to be linear and it has fault tolerance since the network can work even if a neuron or more are unavailable. Neural Networks (NNs) are also adaptable to new data and can change their behaviour after the training. Furthermore, NN respond to evidence since they adapt to the best pattern for the input/output relationship. Finally, an input always corresponds to the same output if the neural network is not trained in the in-between.

In this research work, it is going to be used a NN based predictive model. In this chapter the history of the NN is discussed, the most important concepts that will be used along the research, the several forms and types of networks and their usages and at last, the most suitable frameworks to develop NN based models.

3.1 Neural network history

The first attempt on trying to model the human brain was proposed by McCulloch and Pits in [29]. It supposes that the human brain has the neuron as the fundamental unit. Furthermore, it considers that humans perceive time in a discrete way, having the synapses (when the neuron actuates) as the fundamental unit of the perceived time. It also states that each neuron has a binary value at each time step. The neurons are connected to form a graph, and each neuron's boolean value depends on the number of ones and zeros it receives from prior connected neurons. It is possible to design most logic within the proposed format.

In 1958, Frank Rosenblatt models the perceptron [30]. The perceptron is the node base for a NN, which, based on McCulloch and Pits, gets activated if the input sum exceeds a defined threshold. This little unity is still used nowadays as a network, and several perceptrons are connected. Furthermore, Rosenblatt also made progress on how these networks should learn. This is why Rosenblatt is called the "father of the NN". After Rosenblatt's success, new models were created. In 1986, the MLP was proposed and the discovery of backpropagation to train its models with more layers and neurons occurred [31].

Although most of the studies were done in the past century, this technique is currently considered state of the art in many problems. It has been implemented for solving a lot of challenging issues, such as image classification [32], stock forecasting [33], recommendation systems [34] and even image, text or music generation [35].

3.2 Important concepts

In this section, some of the most important concepts regarding NNs that will be referenced throughout the research are explained. Those concepts mostly regard the possible NN formats and some important components of the NN architectures.

3.2.1 Perceptron

The perceptron is a NN built only one neuron, only one layer, without any recurrence and a step function for the activation as depicted in Figure 3.1. It is possible to use a perceptron to make a classification between two classes. The output of a perceptron, y, will be zero or one, depending on the class. This methodology alone has many limitations. For example, even though it can learn the behaviour of an OR or an AND gate, a perceptron alone can not be trained to reproduce patterns such as an XOR gate. This is because a perceptron alone cannot learn class separation with classes that are not linearly separable [36].

Figure 3.1 depicts a perceptron, which in this particular case implements a NN with only one neuron and a step activation function. The inputs to the perceptron are the x_i values representing any commensurable thing, and its output is y. The input showing the number 1 is the bias, which is the constant that sets the threshold for the class separation.

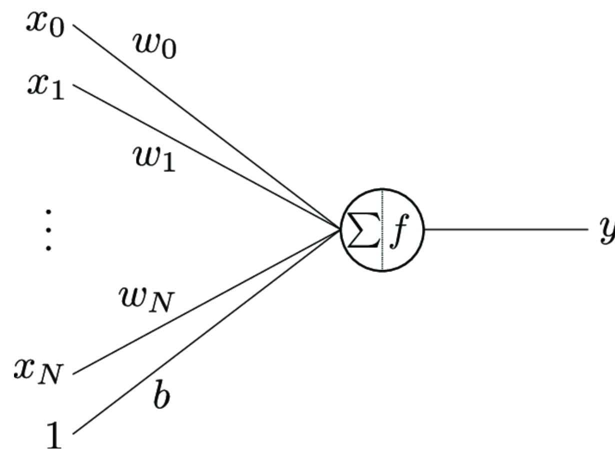


Figure 3.1: Perceptron schema.

Figure 3.1 also shows the trainable parameters for the perceptron. Those values, represented by w and b, are the values that determine the classification into one of the two possible classes. It is required to find the parameters that optimise the performance of the perceptron on the training dataset. To get such classification, the weights, are multiplied by the corresponding input and summed up, as shown in Equation (3.1). Then, the class value is obtained using Equation (3.2), which corresponds to an activation function for a step function. The weights are responsible for the line format that separates the two classes, while the bias is responsible for adjusting the threshold position.

$$\Sigma = \sum_{i=0}^{n} x_i * w_i + b \tag{3.1}$$

$$y = f(\Sigma) = \begin{cases} 0, \Sigma < 0 \\ 1, otherwise \end{cases} \tag{3.2}$$

3.2.2 Activation functions

Although perceptrons can be used to build a NN, the neural network's neurons do not always need to be a perceptron, whose activation function is the step function shown in Equation (3.2). Consequently, it is possible to have different activation functions associated with a neuron, besides the step function. Moreover, it is possible to have different functions from neuron to neuron within the same network. The main drawback of using a step function in a neuron is that its derivative is zero for all differentiable points. This creates a significant problem if a backpropagation algorithm is used to train the network. Since it

is impossible to update the network weights and stabilise them with a zero derivative since the second term of Equation (3.12) will always be zero.

Relu, *leaky relu*, *sigmoid* and *tanh*, defined by Equation (3.3), Equation (3.4), Equation (3.5) and Equation (3.6) respectively, are used in hidden layers and on output layers when the output can be more than one class. These functions, as well as the step function shown in Equation (3.2), only access a single neuron and decide if it should be active or not. This decision is independent of other output classes of neurons in the same layer. These functions serve the same purpose as the strict step function, however, its derivative is never zero, and therefore these functions can be used with backpropagation algorithms.

$$relu(x) = max(0, x) \tag{3.3}$$

$$leakyRelu(x) = \begin{cases} 0.01, x < 0 \\ x, \geq 0 \end{cases} \tag{3.4}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{3.5}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.6}$$

The relu function defined in Equation (3.3) outputs the zero value when the neuron is deactivated (value less than zero). However, if the neuron is active, it will have a linear function as output. These properties have the advantage of giving importance to a neuron with a value higher than one, which means that, unlike the step function that only states if the neuron is active or not, the relu also has information about how active the neuron is. The main drawback is that the derivative is zero for negative values, similar to the step function, which does not allow training. However, this drawback can be overcome by using the leaky relu function defined in Equation (3.4), which has a value very close to zero for negative values but does not have a constant derivative. This activation function is mainly used on Convolutional Neural Networks (CNNs).

The sigmoid function defined in Equation (3.5) is similar to the step function since for values lower than zero the output will be closer to zero, and for values higher than zero the value will be more relative to 1. The output value represents the likelihood of each binary class being the correct one. However, it has the main advantage of being differentiable at all points, and its derivative is not constant. Therefore, this activation function works well with training methods using the gradient descent algorithm [37].

The tanh function in Equation (3.6) is also differentiable for all values. However, unlike the other functions, the highest value this function can have is closer to 1, and the lowest is closer to -1. This

difference makes it possible to converge to the optimal solution using some gradient descent-based training method [37].

For multi-class classifiers, it is crucial to have information about how likely each class is to be the correct output in the output layers. In such situations, the softmax function described in Equation (3.7) is used as the activation function. This activation function considers the value of the neuron along with the others within the same layer (usually in the output layer) it outputs for each neuron the probability of being the correct class. All the outputs of the neurons on the output layer will sum up to one. These properties enable choosing the most likely class and ranking the classes by ordering the probabilities of each one being chosen.

$$softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.7}$$

3.2.3 Deep learning

Deep learning techniques use several layers of neurons interconnected in chain. Furthermore, there may be more than one neuron per layer and a model with a graph format is created with all the neurons combined. This technique is used due to the flexibility that these models have. For example, if more than one layer is used, it is possible not only to represent an XOR gate but also any other mathematical function [28]. Figure 3.2 depicts a NN feedforward with three hidden layers.

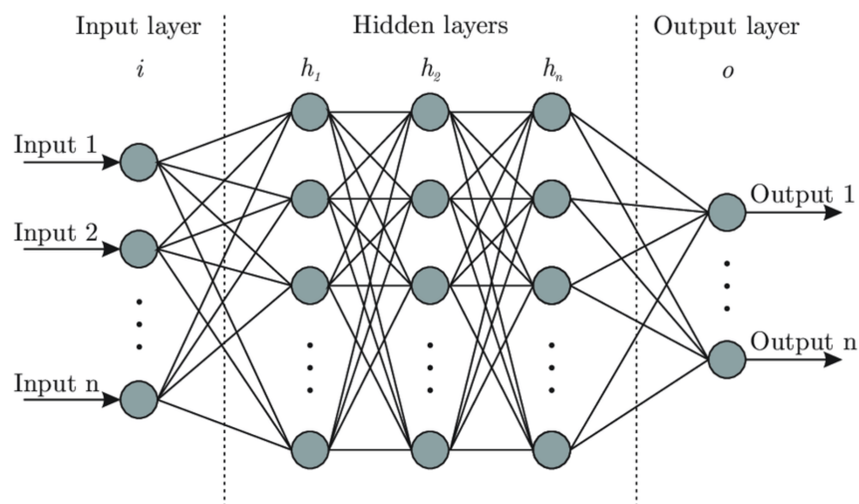


Figure 3.2: Example of a feedforward NN with one input layer, three hidden layers and one output layer.

A NN can have multiple architectures. It may vary the number of layers, the number of neurons in each layer, which neurons are connected to each neuron on the next layer, which activation layer is used and which type of neuron is used. These values are decided prior and cannot change after the network is created. Those variables are called hyperparameters.

Conversely, trainable parameters need to be updated for each training. These values consist of the weight for each connection on the graph of a NN. These weights are represented by w_i, j where *i* is the neuron where the edge started and *j* where it ends. The weights represent the importance of a particular entry on the next neuron choice. Hence, higher values correspond to more important contributions while lower values to more irrelevant ones. A weight of zero means no contribution.

Each NN has several neurons and, in each one, Equation (3.1) and Equation (3.2) are going to be applied (if the step function is used as an activation function, otherwise Equation (3.2) should be replaced). Furthermore, each neuron's output (y) is used as an input for neurons in the next layer connected to it. The fact that there are more neurons per layer creates more possible combinations for classification. For example, in Figure 3.2 the fact that there are n neurons in layer h_1 creates n different step functions (if Equation (3.2) is used as the activation function). These n steps can be further combined by the weights that connect h_1 and h_2 to create any step function after h_2. This will necessarily be a step function after the first layer, but when the different steps are combined within the hidden layers, it is possible to create any function.

When the values finish being propagated from the input layers, passing by the hidden layers and reaching the output layer, the output of the model is created using a special activation function that takes each of the output neurons into consideration. Usually, the output of the value is the neuron in the output layer with a greater value.

3.2.4 Recurrent neural network

The NN presented in Figure 3.2 is a feedforward NN, in which the output only depends on the current input. Recurrent Neural Networks (RNNs) are another type of NNs that take into account the order of the inputs and their influence on the predicted values. An example of a RNN is presented in Figure 3.3.

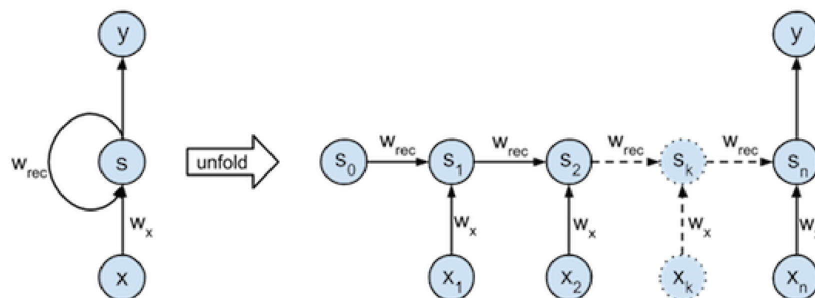


Figure 3.3: RNN with one layer and with only one neuron. This RNN is then unfolded to an equivalent feedforward NN. Removed from [3]

According to [38], there are five architectural types of a RNN. In one-to-one RNN, the output only depends on one input. Hence, this type of RNN is a feedforward NN, it is possible to use neurons

usually used for RNN in this configuration such as LSTM and Gated Recurrent Unit (GRU). One-to-many RNN have a single input that leads to multiple outputs. It is used, for example, to generate text from a single image. Many-to-one RNN is a configuration with many inputs and a single output. This architecture is mainly used for forecasting, such as forecasting stocks from historical data to a specific moment or into sentiment analysis from a text. The last architecture type is the many-to-many RNN, which has two variants: i) several outputs are generated when many inputs are first considered, such as text generation from the introductory text; ii) a new output is generated every time an input is given, considering the current and previous inputs. A video in each second labelled is an example of this type of RNN.

As shown in Figure 3.3, every RNN can be unrolled into an equivalent NN. This process has special importance when training these networks since it is the initial procedure of the Backpropagation Through Time (BPTT).

3.2.5 Backpropagation and training

Training a NN, either with or without recurrence, has two main phases. The first consists in determining the risk which means how well does the model behave in a particular labelled dataset. After assessing the correctness of the behaviours, the second part of the training consists in updating the trainable parameters of the NN to improve the correctness of its behaviour or, in other words, reducing the risk function value. This operation can be performed several times with the same dataset and each iteration is an epoch.

According to [39], the empirical risk represents the mean error for a group of samples, for example, the training dataset. This function is defined by Equation (3.8), where N is the number of elements in the dataset, L is the loss function, y is the correct label and $\hat{y}$ is the predicted label. The risk value ($\mathcal{R}$), therefore, is the mean of the loss function value for each entry in the dataset. The loss function represents how well a specific $\hat{y}$ was expected by comparing it to the real y. Several loss functions can be used to calculate the risk, the most common the Sum of Squared Errors (SSE), defined by Equation (3.9), the cross entropy error, determined by Equation (3.10), and the Root Mean Square Error (RMSE) defined by Equation (3.11). In each of the equations, the value of the loss is calculated for each element in the dataset, where n represents the amount of classes considered in the model.

$$\mathcal{R} = \frac{1}{N} \sum_{k=1}^{N} L\left(y^{(k)}, \hat{y}^{(k)}\right) \tag{3.8}$$

$$L_{SSE}(y, \hat{y}) = \sum_{i=0}^{n} \left(y_i - \hat{y}_i\right)^2 \tag{3.9}$$

$$L_{CEM}(y, \hat{y}) = -\sum_{i=0}^{n} y_i \log \hat{y}_i \tag{3.10}$$

$$L_{RMSE}(y, \hat{y}) = \sqrt{\frac{\sum_{i=0}^{n} (y_i - \hat{y}_i)^2}{n}} \tag{3.11}$$

To minimise Equation (3.8) the backpropagation algorithm can be employed. This algorithm updates the weights to find the best combination of weights. Such optimisation is realised in two phases: feedforward and backpropagation. In the feedforward phase, an input is selected from the training set, and the corresponding output is calculated. After this, the loss function is calculated, comparing the predicted result with each class expected output. In the backpropagation phase, the errors are propagated from the output layer using Equation (3.12), and the weights are updated. In Equation (3.12), w_t corresponds to the weight value before the backpropagation is performed, w_{t+1} is the value of a particular weight after being updated, α is the learning rate defined and $\frac{\partial L}{\partial w_t}$ is the influence that the weight has on the loss function L.

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t} \tag{3.12}$$

BPTT [40] is very similar to the backpropagation algorithm. However, for networks with recurrency, it is necessary to repeat this operation throughout the time. To do that, and as shown in Figure 3.3, an RNN can be unfolded into a feedforward network. After this process, the BPTT is performed as a normal backpropagation. However, some weights are shared in several iterations within the same input. These weights are the ones on recurrent layers. Usually, these weights are assigned with several values, and at the end, the final value will be the average of all given values within the same backward passage. It is possible to use shared weights not to allocate the same weight repeatedly when the unroll is performed.

Backpropagation and BPTT are the most used methods for training NN and RNN. Using the chain rule and automatic differentiation [41] makes it possible for tensor-based frameworks to quickly implement backpropagation training without calculating the complete derivatives of the function composed by all operations together. Furthermore, this method is considered very fast and can be used in almost all types of networks.

However, backpropagation also has some drawback, Since the error and the gradients originate from the output layer to the input layer, the propagated gradients may get smaller and smaller while passing from layer to layer. This problem is aggravated in RNN since these layers are unfolded and, therefore, the gradients pass by it more times. Conversely, it is also possible to have gradient explosions, where the gradient increases from layer to layer [42]. In such cases, the weights are either almost not updated or they are updated very much, getting huge values. Another problem is that the training is very sensitive to the data being used, especially in the case of outliers that significantly influence the weights update.

3.3 Neural network types and usages

As discussed in the previous sections, there are several types of NN, such as the perceptron, the MLP and the RNN. Nevertheless, it is possible to have more types of NN by varying the network topologies and connections between neurons and creating different kinds of neurons, besides the perceptron.

3.3.1 Convolutional neural networks

Convolutional Neural Network (CNN) is a type of feed-forward NN mainly used for image classification. This type of networks has been gaining popularity due to the high accuracy it provides and how they analyse images. It was firstly proposed for detecting numeric digits from handwritten images. One of the most famous CNN is the AlexNet, developed initially for the ImageNet contest [32]. This particular CNN has 1000 different classes as output and provided an accuracy rate between 17.5 % and 37.5 % on the contest.

These networks combine a deep learning approach with kernel convolutions. The input is usually an image and is represented by an array with three dimensions (Width×Height×Depth), where the depth means the Red-Green-Blue (RGB) intensity for the colour of each pixel. This configuration makes it possible to describe every pixel in an image. There are three types of layers in a CNN, the convolutional layer, the pooling layer and the dense layer. There are filters with specific dimensions on the convolutional layers, represented as matrices. These filters perform convolutions with the input on those neurons to detect particular patterns. After the convolution, a new matrix is created, and an output number is recovered from doing the dot product of the result matrix. Next, neurons will use this new information matrix by compiling these dot products and performing new filters. This way, it is possible to analyse the image by parts, having each filter responsible for each piece. The pooling layer is responsible for reducing the matrices' size. This technique is commonly performed by max-pooling [43]. The last layer is a fully connected layer that associates the matrix with the classes defined as the output. The final layer usually uses softmax as an activation function to determine the probability for each class.

In general, CNNs are created by combining all these layers. Nonetheless, each layer can be configured in several different ways. For example, by changing the number of neurons per layer, the filter dimensions and the order or combination of the different layer types. These networks have been widely used, especially with image or video inputs. They provide excellent accuracy and have proven to be reliable solutions. However, they are mostly designed for data that do not depend on previous states.

3.3.2 Vanilla recurrent neural network

A vanilla RNN, or simple RNN is an RNN in which the neuron mathematical function is given by Equation (3.13), where $W_{h,x}$ are the weights for new entries, $W_{h,h}$ are the weights for recurrent entries, x_t is

input at time t, h_{t-1} is the output of the neuron for time $t-1$, and the bias is given by b_h. Therefore, the neuron does not have any mechanism to avoid gradient explosions nor vanishing, which are common in RNNs.

$$h_t = \tanh\left(W_{h,x}x_t + W_{h,h}h_{t-1} + b_h\right) \tag{3.13}$$

This network can be used for any input/output relationship and, similarly to other RNN types, it is mainly used for time series data. As an example, this type of network has already been used for identifying music patterns [44]. However, it suffers from being affected by gradient descent problems. On the other hand, it has the advantage of being simpler since it has only one equation and no additional training parameters, which makes it quicker to train an epoch.

3.3.3 Long short-term memory networks

The LSTM network was created to overcome the problems of vanishing, and exploding gradient [45]. The LSTM neuron is presented in Figure 3.4, and the equations that support it are defined by Equation (3.14), Equation (3.15), Equation (3.16), Equation (3.17).

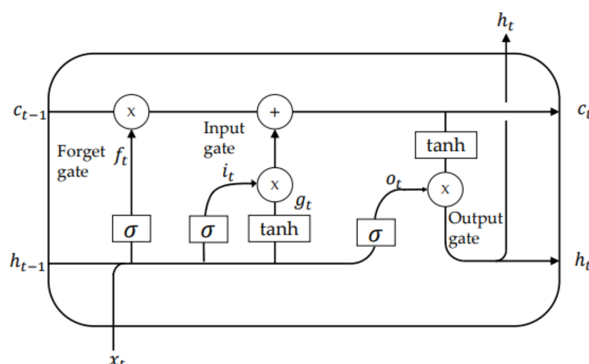


Figure 3.4: An LSTM cell, as shown in [4].

$$i_t = \sigma\left(W_{i,x}x_t + W_{i,h}h_{t-1} + b_i\right) \tag{3.14}$$

$$g_t = \tanh\left(W_{g,x}x_t + W_{g,h}h_{t-1} + b_g\right) \tag{3.15}$$

$$f_t = \sigma\left(W_{f,x}x_t + W_{f,h}h_{t-1} + b_f\right) \tag{3.16}$$

$$o_t = \sigma\left(W_{o,x}x_t + W_{o,h}h_{t-1} + b_0\right) \tag{3.17}$$

The LSTM stores two different types of states, the hidden state at the time (h_t), and the neuron state at the time (c_t). These states are defined using three different gates: the input, the forget, and the output

gates.Equation (3.14) and Equation (3.15) correspond to the input gate [46], which is responsible for considering the previous state, h_{t-1}, and the current input x_t. The forget gate is given by Equation (3.16) and is responsible for measuring how much information from the previous state c_{t-1} should be kept and how much should be removed. Finally, the output gate presents the output for the current neuron by considering the values from the forget and input gate combined with the cell state, as shown in Equation (3.17).

The LSTM network creates a large number of trainable parameters, which increases the training time. However, it may influence the network's performance once it resolves the gradient problems that RNN and deep NN have. LSTM networks are mainly used for time series data and for generating information such as speech emotion recognition [47] or language modelling [48].

3.3.4 Gated recurrent unit network

The GRU network first appeared in 2014 [49]. The GRU is very similar to the LSTM since it has the same purpose: having a RNN without the vanishing or exploding gradient problem. This type of neuron has proven to provide similar accuracy and sometimes even better accuracy than the LSTM, although being more simple and with less trainable parameters [50]. A GRU neuron is shown in Figure 3.5 and the main equations that define its behaviour are Equation (3.18), Equation (3.19), and Equation (3.20).

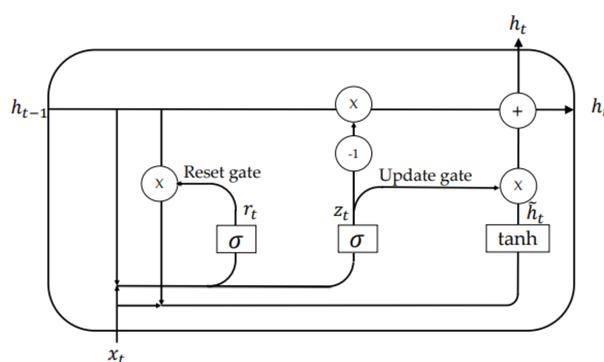


Figure 3.5: A GRU cell, as shown in [4].

$$z_t = \sigma\left(W_{z,x}x_t + W_{z,h}h_{t-1} + b_z\right) \tag{3.18}$$

$$r_t = \sigma\left(W_{r,x}x_t + W_{r,h}h_{t-1} + b_r\right) \tag{3.19}$$

$$\hat{h}_t = \tanh\left(W_{h,x}x_t + W_{h,h}\left(r_t \circ h_{t-1}\right) + b_h\right) \tag{3.20}$$

GRU cells only have two gates: the reset gate and the update gate. The update gate is essential since it decides how much information should be forgotten and how much new information should be

memorised. The behaviour for this gate is defined by Equation (3.18), where $W_{z,h}$ corresponds to the weights for the connection to the previous state h_{t-1}, $W_{z,x}$ are the weights connecting the gate to the input x_t, and b_z is the bias inserted on the gate. The reset gate is also responsible for forgetting previous state information and filtering new ones. The main difference between the reset and the update gates is that the update gate is responsible for long-term memory, while the reset gate is responsible for short-term memory. The behaviour of the reset gate is defined by Equation (3.19).

Equation (3.20) defines the behaviour of the output state, where $\circ$ denotes the Hadamard product. Unlike the other gates, this activation function is the hyperbolic tangent.

Gated Recurrent Units (GRUs), like other RNNs, are suitable for time series data. The fact that they overcome the gradient problems makes them ideal for RNN with long-term recurrence orders. Also, it has the advantage of having fewer parameters, making the training quicker and reducing the memory requirements for the model implementation. However, since GRUs are more recent than the LSTM, they are not as used as Long Short Term Memory (LSTMs). Still, it has been used for a natural language processing task to translate texts from English to French [49]. Nowadays, its applications are more comprehensive. This network is also used to forecast modellings, such as creating a recommender system for several suggestion types such as music and movies [51].

3.4 Frameworks for neural networks

When choosing a deep learning framework, several different factors must be taken into account, such as the available types of networks, the optimisations supported by the framework, or understanding how the training is done, for example, if it can be done by using CPU and Graphics Processing Unit (GPU) in a distributed manner to save time. Other factors include the involved programming language, popularity and availability of an active supporting community.

Nowadays, TensorFlow [52] is the most common framework for implementing deep learning NNs [53], it is based on tensors for representing the graph. It provides most network types already implemented, options for parallel training, a vast community and a very well documented Application Program Interface (API). It has an API in Python with a very efficient implementation in C++. It also has a vast API to describe the network, understand the training and several optimisations already implemented. In terms of community, it is one of the largest, with multiple examples of how to use this framework. In the same way, frameworks like Pytorch [54], Caffe [55] and MxNet [56] also have similar benefits as Tensorflow. Although MxNet is faster, the overall advantages are very similar, and both frameworks are largely used for academic works and production models.

For this particular project, where the RNN must be implemented in a low resource environment (STB), it is fundamental to assess the memory and disk requirements of the framework, as well as

its dependencies and capability to interact with an embedded device, since not only the NN must be deployed in the STB but also used to infer on the next channel and to be trained in this environment.

Even though the previously referred frameworks are the state of the art for deep learning, they suffer from an extensive library dependency. For example, TensorFlow requires 1.1 GB of Hard Disk memory to be installed. The same occurs with the rest of the presented frameworks. Therefore, such frameworks are unsuitable for a model that needs to run in a low resource environment.

Light frameworks for NN are still emerging since it is challenging to have all possible optimisations and types of networks available in short memory space. Several of these frameworks have been presented in the past years for microcontrollers, especially for mobile applications in smartphones and Operating System (OS) capable of using NN. There are mainly image and speech recognition systems implemented on smartphones, so having the NN installed on the device reduces the latency. Some examples are plant recognition applications.

TensorFlow Lite is a solution that can fit in 16 KB and can be used for inference. The model needs to be created and trained in TensorFlow and converted, which is inefficient if the NN is being trained regularly. The on-device training feature, which was a requirement for this project, was only added on May 2022, which now allows complementing the built in model with training performed inside the device. Unfortunately, this feature was only later implemented, so this framework was not used in this work.

Tiny-dnn [57] is another low memory library that provides training and inference for NN. It is available as an open-source project. However, it only works on feed-forward NN, not providing an API for RNN.

Kann [58] is a framework that uses very little memory on the hard disk (124 KB). It provides functionalities for feedforward NN and RNN, such as GRU, LSTM and vanilla RNN. This framework makes use of the CPU and was not devised for scaling large networks. However, it has similar training times as TensorFlow for small networks. Kann is an open-source project with a small community and with a well-described API. Some functions, such as training an RNN, require the user to create his/her routine by integrating the already available training API. This lightweight framework, fully built in C, was already used in other research works [59] and was also chosen for this project.

4

Proposed solution

The main goal of this research work was to devise a NN model for the prediction of the channel a user would like to watch when doing TV zapping and suitable for implementations in STB. Figure 4.1 depicts how such model can be integrated in a STB. The NN model has the responsibility of predicting the next channel to be displayed, while the remaining processes of the STB responsibility. How the STB performs the log save, the channel request and the stop request operations presented in Figure 4.1 is not known by the NN model and, in most cases, should be dependent of the STB system, and the service provider architecture. Nonetheless, the NN model implementation and execution should be independent of such operations.

In the proposed approach, the STB operation should be the following: once a channel change is requested, such request should be immediately saved in the logs using, for example, the format described in Section 4.1. Then, two different flows can occur. If the NN model was not able to successfully predict the channel change, the STB will have to request the new channel to the provider, which will result in a channel change delay. On the contrary, if the channel change was successfully predicted, the STB request to the provider was already performed and the channel change occurs with a negligible delay.

In both cases, the model will then infer which channel should be requested next, using the list of

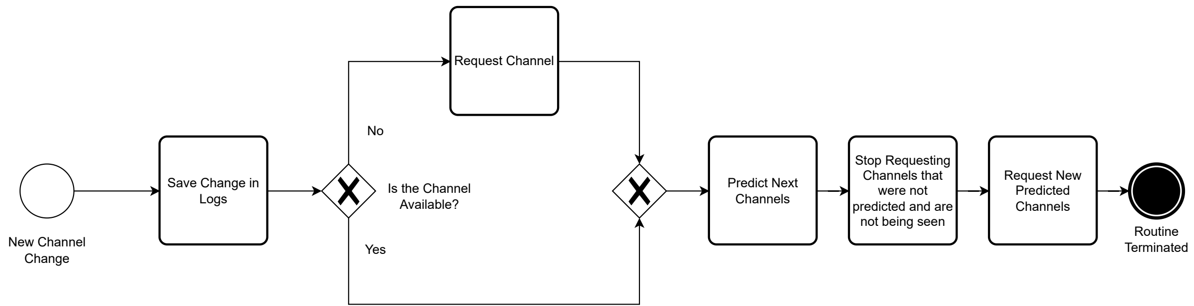


Figure 4.1: Process diagram of how a new channel command is processed.

the last displayed channels as input. After this inference process, the channels that are not going to be displayed and that the model did not predict are no longer required by the STB, which sends another request to the service provider to stop receiving them. Finally, the STB issues another request to the service provider in order to start receiving the newly predicted channels.

The proposed NN model is based on a RNN and its output can be either the most probable channel or an ordered list of the most probable channels to be displayed at a given moment. In the following sections, the architecture of the proposed model is detailed.

4.1 Data

One key factor to select a NN model is the type and amount of data to be processed. To predict TV channel changes, the simplest models consider only data regarding the channel number and the time at which the channel change occurred. Yet, there are other models that consider additional user data. For example, in [18] the model also considers general information, such as the most popular channels for all the users or even the surfing behaviour. Even more complex models take into account personal information regarding the user's favourite programs, their preferred TV genres and some demographic information on the user.

In our approach, these options were not considered for several reasons. Firstly, the simplicity of using only the logs regarding channel changes makes the model open to be integrated with different STB models and several existent and future television service schemas. Secondly, using information from several users creates the need to have a central server handling the data. Because our model should work for several different users operating the same STB and, therefore, it is not feasible to take into account information regarding the preferred TV genres of all the users. Finally, models using common information among users have shown not to perform very well, as discussed in section 2.4.

In the model that was devised, the data that is used for channel prediction is obtained from the channel change logs stored in the STB, which typically includes only the registry of the new channel

and the time at which the change occurred. Furthermore, the model only uses information regarding the STB where the model is running, so the information and logs are not shared among several users.

4.1.1 Data collection

The data considered by the proposed model, which must be stored in the STB, reflects the channel changes that the user has performed. For keeping a channel change record, the STB logs just need to store the time at which the channel change was requested and the requested channel number. Table 4.1 exemplifies such log for a STB user.. The information shown is the one existing before processing the data.

Table 4.1: Example of data saved in the logs for a specific user.

Date	Time	Channel Position
2021-01-02	09:29:40.000	312
2021-01-02	10:30:13.000	48
2021-01-02	10:53:25.000	313
2021-01-02	12:27:16.000	75
2021-01-02	12:28:20.000	93

This minimal data storage approach can then be enriched in runtime by generating another type of information, such as previous channel, day of the week or day of the year and to be further normalised. This solution avoids privacy problems regarding personal data, diminishes the amount of data stored inside the STB and avoids using data that may not be available in every TV or STB schema for the model to work.

4.1.2 Data processing

The data collected by the STBs and stored in their logs is not suitable for direct application in a NN model. Hence, such data needs to be processed to be used in a ML model. Table 4.2 shows the result for the preprocessing (i.e. treatment and enrichment) of the data presented in Table 4.1, where the several fields that are represented can be used as input for the proposed RNN.

Table 4.2: Example of data after being processed. Fields marked with * are represented as one hot encoding, which was suppressed for readability.

Input				Output
Day Of the Year	Time of the Day	Week Day *	Current Channel *	Next Channel *
0.0054795	0.6202355	6	312	48
0.0054795	0.3818098	6	48	313
0.0054795	0.2864549	6	313	75
0.0054795	-0.1186928	6	75	93
0.0054795	-0.1186928	6	93	75

As it is well known, the NN input values need to be normalised, with values between -1 and 1, in order to reduce the propagation of errors [60]. If data is not normalised, the backpropagation that uses the gradient descent method will have drastic variations for large input numbers. To solve this problem, the date field in Table 4.1 was divided into two fields in Table 4.2, Day of the Year and Week Day. Day of the Year represents a day, from 1 to 365. After getting this value, the day is normalised by being divided by 365. Week Day represents the day of the week, from 0 to 6. This input was considered due to the fact that TV programs are usually periodical and broadcast on the same day of the week. Weekdays are not continuous, so each value from 0 to 6 (representing a weekday) is treated as a class. Considering this, the day of the week is represented using one-hot encoding. The input is transformed into seven entries, where the number one is placed in the index corresponding to the weekday and all the other entries are filled with the number zero.

Similarly, the Time field in Table 4.1 is represented as the Time of the Day input in Table 4.2 as Time of the Day. For these data to be fed to the model, firstly, it is calculated the second of the day that is correspondent to the time at which the channel was changed (generating a value between 0 and 86400), creating a continuous variable that represents the time. This data also needs to be normalised, and the process can be done by simply dividing the second of the day correspondent to the Time by the total seconds of a day. However, to avoid a discontinuous point from 23:59:59 on a day to 00:00:00 on the following day, it was used Equation (4.1) to normalise the Time of the Day and maintain the time continuity [61].

$$TimeOfTheDay = \sin\left(\frac{2 * \pi * SecondOfTheDay}{60 * 60 * 24}\right) \tag{4.1}$$

In Table 4.1 there is only one entry per channel change. However, such data can be used to discriminate the channel the user is currently watching (Current Channel) and the one he/she will see next, as shown in Table 4.2. This value is the same as the current channel in the subsequent row. In our model, a channel is represented by a number from 1 to 1000. However, it is a discrete value. Having this in consideration, channels are treated as classes. To process channel classes, the input field needs to be one-hot encoded. Accordingly, for each table entry, all the channels are marked with a value of 0 except for the class corresponding to the channel that is being represented, which will have the value of 1. This is performed not only for the Current Channel but also for the Next Channel since the classes match.

In the proposed model, the expected output is the channel that the user will watch next, while the input is the fields previously described. Regarding the time entry, it corresponds to the current time since the model can update the output every second (or at any defined rate). Nevertheless, the previous channel only changes if the user indeed changes the channel.

4.2 Neural network approach

In this work, a RNN-based model was adopted for several different reasons. Firstly, because after the training, the inference procedure is very fast. Secondly, because NNs provide high tolerance to failure even if some input is incomplete. The capability of a NN changing its behaviour was also taken into consideration, which is a very important feature when dealing with data collected in real time. On the predicting channel changes problem, it is well known that users quite often change their favourite channels or start having different schedules. A NN adapts its weights to this change while training without having to recreate the model from scratch. Furthermore, it creates the opportunity to train the model every week with some new data, without having to dispose and recreate the model every time. Also, having a NN with recurrency allows having the model output depending on more than one of its inputs. In this work, a RNN with a many-to-one configuration was adopted, so that the channel prediction can take into consideration not only the TV channel the user is currently watching but also a set of channels previously displayed.

In the context of this problem, the main advantage is that if the user is, for example, in channel one and moves to channel two, on a feedforward NN this would always have the same output for the channel after the two (if there are no other input variables), regardless if it was previously on channel one or on a different channel. A RNN also depends on the channel that was once being seen. In this case, channel one is also taken into consideration for inferring the next possible channel.

4.2.1 Framework

The framework that was chosen to implement the proposed RNN model was the Kann library [58]. This decision was the result of a careful analysis of the pros and cons of several frameworks, as discussed in Section 3.4. To this matter, the storage and memory requirements of the platform are two critical factors, due to the typical specifications of STBs. Another important factor is computational efficiency. The Kann library only uses 132 KB of the filesystem and it is implemented using the C programming language requiring only the standard C library, which is available in most Unix systems such as the ones used in STB. Furthermore, the Kann library is as efficient as other libraries for small NN [58] such as the one devised in this work.

4.2.2 Inputs

The data available for the model to predict the next channel is in the format presented in Section 4.1, after the parameters are normalised and the classes expanded in the one-hot encoding format. A row in Table 4.2 represents a channel change, and it has sufficient information for creating a prediction.

Since the implemented model consists of a many-to-one RNN, many inputs influence the unique model output. Therefore, the input to the RNN is a vector of such table rows representing the history of the last channels watched. The size of this vector needs to be configured, so it is a hyperparameter called unroll length. As a result, the prediction depends not only on the last channel watched but also on the sequence of the last channels visualised by the user.

4.2.3 Hyperparameters

The RNN architecture is not defined a priori but results from what the model is being used for and the type of data. Such architectures are defined before the training phase. Therefore, they do not change during the training. The hyperparameters are the values that differentiate the architectures such as the number of layers, and neurons per layer. Using the same base model, the RNN, several reasonable solutions can be proposed that are capable of achieving similar accuracy and similar efficiencies. The values of the hyperparameters need to be adjusted using some actual data. They should be changed to provide good accuracy for the model and respect the restrictions defined in Section 1.2. Once the hyperparameters are fixed, they will not change from user to user or during each training.

Conversely, the parameters, also called weights of the RNN, are changed after each training, defining the interior of the RNN. These values vary from user to user and in each training iteration. The change of these parameters only affects the model's accuracy for each particular user. Therefore they need to be different for each user, creating a distinct and independent network per user.

In the devised RNN model, the hyperparameters to be discovered are the number of predicted channels that should be stored, the neurons type, the number of layers, the number of neurons per layer, the learning rate, the dropout rate, the unroll length, the number of weeks used to train, and the epochs used for training the network.

Since the model used is a RNN, the neuron type needs to be decided between vanilla RNN, GRU, or LSTM. Has seen before on Section 3.3, the vanilla option is the simplest and only has an activation function. It provides more simplicity and fewer parameters to be trained. However, it does not take into consideration gradient problems. The LSTM and GRU are more complex neurons that avoid gradient explosions. On the other hand, they require more trainable parameters, increasing the training time. Since it is expectable that the final RNN is not very deep in terms of layers, a gradient explosion-proof neuron may not be required.

The number of neurons per layer depends from layer to layer. This value can not change for hidden and output layers, being one per input class for the input layer and one per output class in the output layer. However, having all the hidden layers with the same number of neurons is not necessary, although it is a common practice. There is no rule that states the number of neurons per hidden layer, even though it should be in the same order as the neurons in the input layer. It is also known that it is required at

least two neurons and a hidden layer to have a NN that generalises if a large enough dataset for training is provided [62].

Increasing the number of hidden layers that the network has diminishes the possibility of occurring underfitting. Furthermore, it increases the number of available neurons and weights for modelling the system. It is possible to create any non-linear function with more than one layer. On the other hand, if the data is linearly separable, it is possible to use just one hidden layer. Increasing the number of layers also increases the complexity of the model, which may originate in overfitting to the training dataset.

An increased number of neurons per layer and number of layers affects the total number of parameters that the model has. An increased number of parameters for a network makes the training time higher and requires more allocated memory to have the RNN inferring on the upcoming channels and to train it.

To train a model it is necessary to define its learning rate α. This value is required since the training of a RNN uses the gradient descent algorithm to find the optimal value for the weight's configuration, which uses Equation (3.12) to update the weights by propagating an α fraction of the error. This value usually varies from 0.01 to 0.1. Increasing the learning rate makes the model's weights react quicker to error changes; if they advance very fast, they may diverge from the actual value and undo some of the training previously done [63]. On the other hand, a minimal learning rate may lead to the training error being trapped in a local minimum [64].

As seen in Figure 3.3, a RNN can be unfolded into a feedforward NN. This operation must be performed to train the RNN using BPTT. The value of the unrolled length needs to be an integer higher or equal to one, and it determines how far in time the unroll should go and, therefore, how long should be the history of inputs that influence a single output. The higher this hyperparameter, the more significant the number of previously watched channels will be considered when predicting the next channel. In other words, in a many-to-one RNN, the unroll length represents the many inputs that influence the single output. A small number will make the network dependent on only a few inputs. However, if the unrolled length is a large number, it may be possible that the channel sequence never repeats, making it difficult for the network to train accurately.

By increasing the unroll length, the training time for an epoch increases proportionally, since the unrolled network will have a proportional increase of parameters. In terms of memory, the extra parameters created for the unrolled network are not allocated since Kann uses a shared weight optimisation for it, therefore, the used memory will not change.

The number of channels that the RNN should be able to predict does not affect the number of parameters of the network, nor its training since the network always trains for the correct output. However, increasing the number of predicted channels also increases the model's accuracy. However, as seen in Chapter 1, augmenting the number of predicted channels also increases the model costs, since the

STB must have that amount of channels ready at the same time.

The model is usually trained with a particular dataset and the network weights are updated using Equation (3.12), which is computed per each entry in the dataset. However, it is possible to use the dataset more than once to train the model. The number of times the dataset is used for the same training is called epochs. Since the number of epochs is proportional to the number of times the dataset is iterated, the training time also increases proportionally to the amount of epochs. Having a large number of epochs decreases the training dataset's error. However, a small error on the training dataset may lead to the RNN losing its generalisation properties and becoming overfitted. On the other hand, a meagre number may cause the RNN not to have enough iterations to adjust its weights. The correct value of epochs should be that the error on the training dataset stops decreasing or at a threshold where the error on the training dataset is good enough.

In the proposed model, training is performed only on a weekly base. Nevertheless, the network can be trained using data from the previous week or from several previous weeks to improve its accuracy. The number of weeks used for creating the training dataset impacts the STB memory requirements since the data needs to be stored in logs for more time and the training time also increases due to the higher amount of data that is used for training.

The dropout rate should be a value between 0.5 and 0.8. It is a regularisation method, first proposed in [65] to reduce overfitting. It makes a percentage of the nodes corresponding to the dropout rate disabled. The neurons that are disabled are chosen randomly. The dropout is only active during training to not lose values while inferring. This hyperparameter does not directly affect the training time or the memory needed by the model to be operating either during training or inference.

4.3 Channel mapping

Nowadays, the total of available channels in a TV provider is very large and can sum up to more than a thousand. As discussed in Section 4.1, channels are treated as classes and the model output should be one class corresponding to a channel. Hence, there might be a problem with a large number of possible outcomes for the RNN, which might be a problem due to the lack of enough data to cover all the classes when training the model. Furthermore, a user does not make use of all the channels. In fact, as it is discussed in Section 5.1, most of the users use less than five percent of the available channels (around 50 channels [25]). Another drawback of such a huge amount of classes is the proportional increasing size of the RNN and therefore the extra time needed to train and process it.

To reduce the number of classes, a new channel representation was created with a maximum dimension of `maxChannels`. Consequently, each mapped channel will have a correspondent in the set of available channels. However, the opposite is not true since the mapping will have a smaller dimension.

A similar approach was also pursued in [25].

The main drawback of this implementation is that the model will never predict channels that are not mapped in the reduced dimension. However, even without the mapping, such a new channel would not be used to train the RNN and therefore it would not be possible to predict them. All the displayed channels are stored in the logs, and when the model is retrained all the entries will be considered and may be mapped since this mapping will have a maximum size.

Algorithm 4.1 shows how the need to create a new mapping is assessed, where the `listChannels` is the mapping list containing the mapping between the dimension with all existent channels and the dimension with the channels that the model will acquaint. The `getMostSeenChannels` creates a new mapping of channels, and `addNewChannels` adds the new channels to the existing mapping.

Furthermore, since RNNs are static in terms of input and output format, for the RNN to take into consideration the mapping changes that may have occurred, the model needs to be disposed and reconstructed so that the classes can correspond to the most recently displayed channels.

Algorithm 4.1: Handling new channel occurrences before model training.

```
if getNewChannels(mappingList ml) != NULL then
    previousListChannels ← listChannels;
    /* checks if total different channels is higher than maxChannels */
    if maxChannels > getTotalChannels() + len(listChannels) then
        listChannels ← getMostSeenChannels(maxChannels, ml, listChannels);
        if previousListChannels == listChannels then
            return;
    else
        listChannels ← addNewChannels(listChannels, ml);
    /* deletes the old model and creates a new one with different mapping and/or a
       different number of channel classes */
    deleteModel(model);
    model ← generateNewModel(listChannels);
```

By adding this mapping, it creates the drawback that the not mapped channels will not be used by the RNN either as input or output. However, since these channels would be new to the RNN, it would not have trained with samples that include those channels, so the accuracy is not affected. Once the model is rebuilt, it is trained with the data respecting the last defined weeks to train.

4.4 Model output

The desired output for the RNN is the channel the user will watch when changing from the currently displayed channel. The RNN has an output layer with one neuron per class, which represents a mapped channel. For each output class, once inferring, the RNN will provide a value that, after using the softmax

activation function shown in Equation (3.7). It will be transformed into the probability of each channel of being the desired output for the current network inputs, the output value should be one for the class corresponding to the desired channel and zero for the others. Furthermore, the neuron's output classes will respect the map previously described, and therefore the channel results need to be mapped to the original grid number.

Since the output is given in terms of probability per class, it is possible to rank the several output classes by probability for a given input. Therefore, even though the RNN is trained to predict just one channel, the model output is a list of channels ordered by their probabilities. For each STB configuration, the service provider needs to decide the size of such a list by trading off two parameters: the prediction accuracy versus the cost of having more channels ready. In Table 4.3, it is possible to see five consecutive outputs (having the probability for each channel) for a model with only five different channels. Furthermore, the STB has resources to have two channels available, so the first and second-ranked channels are selected, which is possible to see on the two rightmost columns of Table 4.3.

Table 4.3: Example of the output of the model for an user with just five channels and the respective two most likely classes to be watched next. Each class corresponds to a channel since it has been mapped.

Output class 0	**Output class 1**	**Output class 2**	**Output class 3**	**Output class 4**
0.658356	0.071425	0.265766	0.003691	0.000762
0.002328	0.230009	0.087564	0.679639	0.000459
0.087986	0.000627	0.003119	0.242452	0.665816
0.69812	0.093273	0.206309	0.001909	0.000388
0.092649	0.000403	0.695448	0.209513	0.001988

1st rank class	**2nd rank class**
0	2
3	1
4	3
0	2
2	3

Even though the model provides output for all the mapped classes, the model is trained to provide output for only one class. Therefore, changing the number of predicted channels does not affect the model architecture since the weights, the outputs and the training will remain the same. By increasing the number of channels in the output list, the accuracy can only increase or maintain since the channels existent on a smaller list will always be present on a bigger one.

4.5 Training

To train a RNN, data with a correct mapping between the network inputs and the desired output is necessary.

The creation of the mapped dataset occurs during the model inference, using the logs previously described, therefore, the model will always be trained with data already used during inference. This dataset can comprehend the history of channel changes of several previous weeks.

Even though the model is trained weekly, it is not necessary to recreate the model from scratch on each training. If it is the first time the RNN is being trained, the model will be generated with the correct

hyperparameters, input and output size, and the weights of the RNN will be set randomly. However, suppose it is not the first time the model has been trained. In that case, the model will have the parameters already set from the previous training, making it faster to achieve a low training error, as defined by Equation (3.8), since the RNN will already have some knowledge stored in its weights. The exception in this procedure results from the creation of a new class for a new channel before training. In such cases, a new RNN prepared to consider this class will be generated with the initial weight values randomly set.

The RNN is trained using a derivation of the backpropagation technique described in Section 3.2.5 named BPTT, which is an adaptation of the backpropagation for networks with recurrence. BPTT expands the RNN through time, creating an equivalent feedforward network. However, since the RNN can not be unrolled through infinity, it is necessary to define the recurrence order (defined as a hyperparameter named unroll length). As an implementation note, since the same network is being unrolled, several weights are repeated through several time orders. When the network is collapsed, all the weights orders will be collapsed into their original place in the RNN. To save RAM space with extra memory allocations for the additional weights during the training phase, it is possible to have those weights shared, avoiding duplication of resources. This unroll length also can not be very large since the training time would increase proportionally.

The gradient descent algorithm is used to perform the backpropagation or BPTT. This algorithm uses the iterative process defined in Equation (3.12) to update the weights. However, finding the gradients of the used functions is not simple, and with multiple layers, the gradients become more and more complex. Kann and other tensor-based libraries for deep learning, such as TensorFlow or Pytorch, use automatic differentiation [41] to overcome this problem By passing the training data in feedforward and a backwards mode and keeping track of the intermediate values, it is possible to use the differentiation chain rule only to calculate simple derivatives.

During the BPTT, the model is being trained with the training dataset, and as seen before, the same data can be used in several epochs. So, it is necessary to create stop conditions for the model not to train indefinitely and not starve the STB resources. For example, the training may cease if, after an epoch, one of the stop conditions is achieved. The first stop condition should be the error value, Equation (3.8) which should be low enough for the RNN to be considered trained but not low enough to be overfitted to the training dataset. The second stop condition should be the number of epochs in which the weights are being updated since some datasets will not be able to converge for the minimum error threshold. However, the datasets' size between users varies, so the training time per epoch changes. Thus, as a third stop condition, it is necessary to have a maximum amount of time for the training and therefore limit the training within what is acceptable for the STB to have the extra resources in use.

4.6 Model evaluation

To assess that the model is performing well, a high amount of channels must be correctly predicted so that the user can benefit from it. The accuracy is given by dividing the total amount of correctly predicted channels by the total amount of predicted channels. The accuracy in percentage for a particular user is presented in Equation (4.2). However, there are a large number of possible NNs that can provide a good accuracy e.g. if the output list size is the same as the total of channels, the model will have 100 % accuracy.

$$accuracy(\%) = \frac{\#Correct\ Predictions}{\#Predictions} * 100 \tag{4.2}$$

A successful evaluation of the model requires not only that the model provides good accuracy but also that it complies with the constraints defined in Section 1.2 regarding the STB resources to receive extra channels, disk space and RAM so that it can be as less of an impediment for the model to be implemented in a STB. The model comprehends two main phases, training and inference, and these restrictions need to be checked for both phases, even though the training phase is the most cost-demanding in terms of time and memory.

For the developed RNN model to comply with the requirements, it needs to have a low amount of trainable parameters since the parameters increase the memory allocated by the model during inference and increase the training computation cost, not only regarding time but also regarding RAM to have those parameters allocated in memory. Furthermore, the total amount of predicted channels should be as low as possible.

The considered framework to implement the model is also required to use a small amount of disk space, not only for the framework itself but also for the required dependencies, without disregarding an optimal enough solution that does not increase the training time. The developed model along with the framework should comply with the requirements and they should take into consideration that there may be STBs with lower capabilities.

4.7 Model Implementation

The created model is supposed to be continuously operating in the STB, either doing inference or training. When doing inference, which is most of the time, channel change requests cause the RNN to predict the channel that should be displayed. When doing training, the model uses all the collected zapping information to perform the training. Hence, the model can not predict new channels. Therefore, the training phase should be executed when users are not watching TV. To minimise the downtime of the channel prediction functionality and reduce the training cost, network training does not need to be

performed daily. For example, it can be done once a week and still exploit the regularity of the program grid.

The implementation of the model comprehends the definition of the architecture of the RNN, how it is developed, and the logic necessary to have the network operational, which consists of the data processing to feed the RNN and train it and the triggering of the train plus its logic. As discussed in Section 1.2, this process involves stringent requirements. To comply with such requirements, the model can be implemented using the C programming language. It allows direct memory management, and easy portability to different OS.

The NN component of the devised model was developed using the Kann library [58] [59], which is a lightweight open-source library fully developed in C as a C API and some documentation. Furthermore, the Kann framework does not require many library dependencies, making the built program lighter. To develop this model, Kann provides the core of the NNs, but it is needed to create a specific training routine for the RNN and to integrate the layers in the model generation.

4.7.1 Model generation

Using Kann, a RNN can be created with the `model_gen` function presented in Listing 4.1 designed specifically for the new channel prediction problem. This function returns a structure representing a Kann RNN and requires as input the hyperparameters that model the architecture, such as the number of inputs and outputs for the RNN, number of hidden layers, number of neurons per hidden layer, neurons types and the dropout rate.

The NN model is defined by a `kann_t struct` that contains a list of `kad_node_t nodes`, the gradients for the connections between nodes in the created graph and some auxiliary variables. Each `kad_node_t` node represents a mathematical operation in the NN, and a complex neuron such as LSTM is constructed by connecting several `kad_node_t` structures. Each of these structures contain the internal gradients for the node, the dimensions, the weights, the child nodes representing the nodes connected to it and some other utility variables. The first layer of the NN is generated using the `kann_layer_inputs(n_in)` function, which returns the layer responsible for receiving the inputs. The following layers are added sequentially as a child to the previous layer, having a different function according to the type of hidden layer used (LSTM, GRU or vanilla RNN). A dropout layer is subsequently added, responsible for performing the dropout regularisation. Afterwords, the selection of the output is conducted since a RNN can have a sequence of outputs (if there are many outputs). The output is a sequence of vectors each one with the probability of each channel being chosen in a specific time with as many vectors as the inputs of the RNN. However, only one of such outputs is required in the devised many-to-one model. For this, a selection is used to choose the last element from the output vector as the output, since it is the one that corresponds to the recurrence order. The previous operation creates

the output with the correct activation function, defined by KANN_C_CEM that represents a softmax function, using the kann_layer_cost function and generating an NN with all the layers by using the kann_new function.

Listing 4.1: Model generation code.

```
kann_t *model_gen(int n_in, int n_out, int n_hidden,
                  int n_layers, int type, float dropoutRate) {
    kad_node_t *t;
    int rnn_flag = KANN_RNN_VAR_H0;
    t = kann_layer_input(n_in);
    for(int i= 0; i < n_layers; i++) {
        if(type == 1) t = kann_layer_lstm(t, n_hidden, rnn_flag);
        else if (type == 2) t = kann_layer_gru(t, n_hidden, rnn_flag);
        else t = kann_layer_rnn(t, n_hidden, rnn_flag);
        t = kann_layer_dropout(t,dropoutRate);
    }
    t = kad_select(1, &t, 0);
    kann_t *ann = kann_new(kann_layer_cost(t, n_out, KANN_C_CEM), 0);
}
```

4.7.2 Training

To train the RNN a custom function was developed based on the train function available in Kann. A new training function had to be created because the training of a RNN is different from a straightforward NN, which is the type of training available in Kann. To train a RNN it is necessary to define which values are considered input and their respective labels. In addition, the network must be unrolled. The implemented training function is present in Listing A.1 of Appendix A.

The training process has two main parts: setup and training. The setup part starts by acquiring all the necessary information to conduct the training procedure, i.e. the number of inputs and outputs of the RNN and the amount of trainable variables that the model has. Then, it allocates all the memory required to implement the training procedure: the matrix x that will save the input values for each iteration of the training with each row of the input size and the length equal to the unrolled length; and the matrix y that holds the output values, with each row of the dimension of the output size, although since it is a many-to-one RNN, this matrix only has one row; the vector r that stores the values of each parameter to assist in the backpropagation. Finally, the setup part creates a new NN by unrolling the initial RNN through time and turning it into an equivalent feedforward NN. This new network is used for the training and has the parameters pointing to the RNN parameters. Therefore, updating the values on the unrolled networks updates the values on the original one.

In the action part, the model weights are updated as a result of the network training. The data is stored in a Vector structure named data containing input and output values for the dataset used

to train. Since the RNN has many inputs, the exact channel change will be repeated in several input vectors per position. To avoid repeated information, these values are stored only once, and the logic to choose the correct values to populate `x` and `y` is performed on the training function. Such operation, consists in converting the channels from the numerical form the to one-hot encoded format, according to the mapped classes, as described in Section 4.3, and the data treatment described in Section 4.1.2. Regarding the cost function, it is computed based on Equation (3.9) for the given output and expected output value. `n_cerr` stores the error based on the accuracy (if the model showed the correct result), and `cost` holds the RMSE for the current data, where it is expected to have all classes with a zero value except the correct one. The `cost` is propagated through the NN by using the `kann_RMS_prop` function. At the end of the cycle, the stop conditions are checked to determine if the training should continue for another epoch. This process is repeated once per epoch. The last step of the training procedure consists in releasing all the allocated memory.

The space complexity of the developed training algorithm is $\mathcal{O}(x + y + r)$, where x is the input size, y is the output size, and r is the number of parameters. In terms of time complexity, there is one weight update per epoch. Therefore, the model is proportional to the number of epochs, although each weight update involves two operations per parameter, one for the feedforward and one for the backpropagation. The backpropagation is performed in terms of matrix multiplication. However, the time complexity remains proportional to the number of parameters. Furthermore, unrolling the network u_len times into a virtual network creates a proportional amount of parameters to u_len. Finally, each backpropagation in each epoch is performed once per data in the training dataset, giving a time complexity of $\mathcal{O}(e \times p \times u_len \times x)$ where e is the number of epochs, p is the number of parameters, and x is the size of the dataset.

5

Datasets and refinement

In this chapter, the dataset used to evaluate the designed model is presented. Three different datasets were used: i) an artificial user as a proof of concept for the model; ii) a dataset of 300 real users; iii) a subset of 30 users. At last, the process of defining the hyperparameters is presented and the final combination of hyperparameters for the model is defined.

5.1 Data analysis

The data used when evaluating the RNN model is divided into two main groups: the synthetic data and the real users' data. The synthetic data was created to have a proof of concept on the RNN and, therefore, to verify if it would perform well for some users with a known pattern. The real data is used to have real case scenarios, and it is used in the first part to adjust the model hyperparameters and further test and analyse real users' performance.

5.1.1 Synthetic data

The synthetic data was produced manually to infer if the proposed solution performs well on a user with a clear pattern to the human eye and, therefore, as well to an RNN. If the model performs poorly on this user dataset, it would also perform poorly on a real dataset. This user was created with a minimal set of channels (nine different channels) and a consistent weekly pattern. However, some weeks the pattern has slight differences with channels that are not watched and channels that are momentarily changed in order to simulate common changes of channels that a user may perform. Furthermore, the starting time of watch channels is not strict and has a variation of several minutes (which is not always the same), since a real user will never execute his/hers routines at the exact same time. There are also some momentarily zapping sequences where a user changes to different channels, and the sequence ends on the channels the user initially saw to simulate common zapping sequences.

This artificial user follows a pattern that can be summarised in the schedule represented in Table 5.1, where for each time slot, the channel that the user is watching is expressed in the table by its number. The empty channel blocks on the schedule represent when the user is not usually watching TV. This dataset can also be seen in Figure B.2. However, since the channels are mapped, and some irregularities exist, as previously explained, there are some differences from Table 5.1.

Table 5.1: Channel schedule for the artificial user.

Hour	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
9h00						2	
10h00	2	2	2	2	2		
11h00						3	
12h00							
13h00						1	1
14h00							
15h00						4	
16h00	6	6	6	6	6		
17h00							
18h00	5						5
19h00			8				
20h00	1	1	1	1	1	1	1
21h00				7	7	5	
22h00						9	4
23h00					8	8	9

The data represents a two month user channel history and has nine different channels mapped. It has a total of 277 channel change entries. To make the scenario close to a real user, there are a few days and entries skipped, plus some entries that were not planned according to the schedule, including zapping events while a user is supposedly watching a channel. Furthermore, if a user usually starts watching a channel at a determined time, the channel changing time is not fixed, it has a random variation time (if the channel change is not skipped to simulate some user sudden event that did not

allow him/her to watch TV at that time) of 30 minutes earlier or later.

With this scenario, it is expected that the accuracy will not reach a value of 100 % since there are some outliers and new patterns. However, since the core pattern is constant, it is expected that the accuracy of the system to be very high. It obtained an accuracy of 83.5 % for the final model as shown in Table 6.6.

5.1.2 Real users' data

The data for real users corresponds to the channel change requests performed by 300 users of the same TV provider. The whole dataset has 262360 channel changes and represents the history of the channel changes from 01-01-2021 to 28-02-2021.

Of the entire dataset, it is possible to separate the different users, creating a separate dataset with the channel change history for each of them. By performing an initial analysis of the data, it is possible to observe that the users are very different from each other, and the two main differentiating factors are the number of channels a user sees and the number of channel changes the user makes, which is expressed in Figure 5.1.

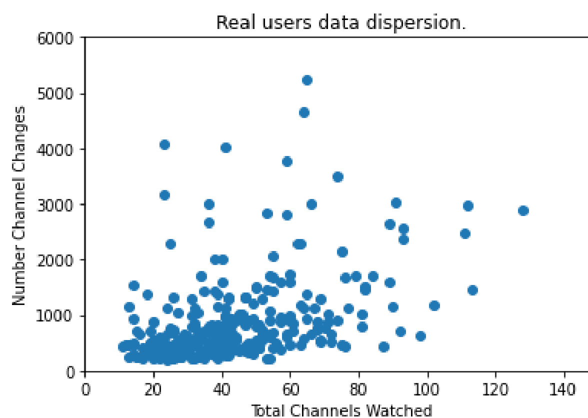


Figure 5.1: Dispersion graphic of the real users by number of different channels watched and number of channel changes.

The two main factors, the number of channel changes and the number of different channels watched, were used as initial indicators of how well a behaviour could be predicted. Suppose the number of channels watched is very similar to the number of samples, empirically, these users should perform poorly since the routine of changing the channels does not have enough repetitions. On the other hand, a user with a number of channel change requests superior to the number of channels watched may be easier to predict since there should be more patterns and more routine, and the same channels are repeatedly watched. However, other factors also may influence the accuracy of the model, such as the number of samples alone (if it is shallow, makes it hard for a model to train) and the dispersion of the data

through the weeks (if the data is all condensed in a week, it is not possible for a model to be accurate).

5.1.3 Best 30 and 270 users

The best 30 users dataset is a subset of all of the real users' data. It consists of the users that will likely perform better according to the algorithm described in this section. Since the group with all the users contains a large number of TV users (300), some will most likely have wildly unpredictable patterns, and a RNN may never be able to learn and detect a pattern. Also, to reduce the time needed to experiment with different models, creating a smaller subset of meaningful users was preferred. This subset is used for finding the best combination of hyperparameters that will be more suitable for a RNN used to predict new channels. Another significant advantage of creating this narrower dataset is that it is possible to see if the model for which the hyperparameters were determined based on only 30 users generalises when using it in users that were not used to find the best combination of hyperparameters.

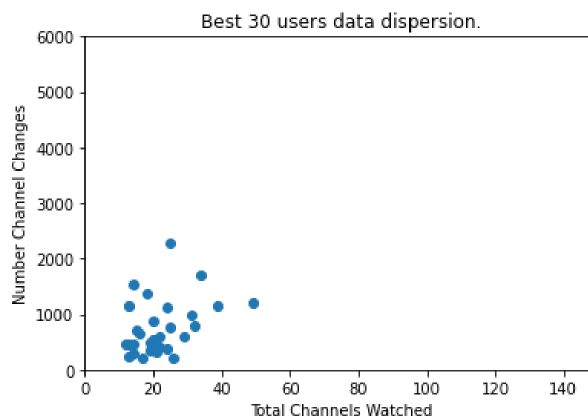


Figure 5.2: Users dispersion of the best 30 users group.

For creating this subset, it was used 100 different RNN models by randomly varying the hyperparameters inside the defined ranges in Chapter 4. All the users would be assigned the same randomly chosen RNN configuration (from the 100 combinations), and the five users that would perform worst would be dropped from the subset. To analyse the performance of a user with a specific RNN architecture for this algorithm was only taken into consideration the accuracy.

This process is repeated with different RNN models until only 30 users are left. These 30 users form the best 30 users dataset. Since this algorithm has several random factors (the randomness used to choose the model hyperparameters and the order in which the models were picked), it does not always produce the same output. The best 30 users are presented in Figure 5.2.

As it is possible to see in Figure 5.2, users with an extensive range of different channels watched were dropped, mainly because there were not enough entries to create a pattern. In general, the users that were not dropped have better accuracy, and therefore, the indicators for how the hyperparameter is

performing are more accurate since it is more likely that these users will be able to have a good RNN that fits them.

The remaining users compose the 270 users dataset. This dataset was required since the RNN model will have its hyperparameters adapted for the best 30 users. Therefore, it is expected that the accuracy increases for them. The main purpose of this last dataset is to test the RNN for users that it was not adapted to and at last to verify the generalisation properties of the developed RNN model.

5.2 Hyperparameter selection

Having the model defined, and knowing that the inner model will be an RNN, it is possible to have various networks that present different results, not only on the number of correctly predicted channels but also on its performance taking into consideration the restrictions defined in Section 1.2. These various RNNs are obtained by changing the hyperparameters. To assess the performance of a particular RNN and how it is affected by the hyperparameter combination, the model's training time, the number of parameters and the model's accuracy will be considered.

The tested hyperparameters should already satisfy the STB restrictions defined in Section 1.2 or be immediately put aside since they will not be able to work in the desired environment. Among the various considered assessment elements, accuracy should be taken first into consideration. However, supposing that different models may have similar results for accuracy, the best model will be the one with fewer parameters on the model (it is worth noting that the parameters are an average of all models since, even though the hyperparameters are the same, it depends on the number of channels considered in the mapping as defined in Algorithm 4.1), and if these are also identical, then the model with less training time.

The algorithm for finding the best hyperparameters starts with having an initial guess on the hyperparameter values and each hyperparameter should have a range of values and a change step. On the first iteration, a model is created for each hyperparameter combination and each of these modes is assessed with the user's data.

After the first iteration, for each hyperparameter, it is checked if the majority of the users perform better for it. If for a particular hyperparameter there was an option with a better performance, that value will be changed to the best hyperparameter value and a new iteration is performed. This process is repeated until a stable combination of hyperparameters is found, where all the hyperparameters are already with the optimal value and should not be changed. Due to the possibility of this process not producing a stable solution in a finite time using an automated process, a manual selection process was used, which takes into consideration the actual meaning of each hyperparameter and the understanding of what is being changed and which values are acceptable.

As illustrated, the used algorithm works interactively and may not provide the optimal solution since the model cost is not only the accuracy (and the RMSE) but also the model complexity measured in the number of trainable parameters (which has an impact on the memory used) and the time used for training. Furthermore, after performing the variation of a hyperparameter, it was calculated the number of users that have better accuracy for each hyperparameter value.

This parameter was used since the average accuracy may be affected by users with very good accuracy (or bad) and with this parameter, it is possible to see the result for each user individually. For the first iteration, the hyperparameter values were chosen in the range of the standard values for each specific hyperparameter, described in Section 4.2.3.

It is essential to notice that this model does not have a defined dataset but one dataset per user. Therefore, some combinations of hyperparameters may work better for some users and not for others. This is why it is taken into consideration not only the average accuracy of all users for each hyperparameter but also the number of users that perform better for that particular hyperparameter. Also, since there are few data entries per week, it was not used the validation dataset to stop the training (by dividing the training dataset), but only used the test dataset (corresponding to the following week) to evaluate how the model will perform after being trained. In sum, there were four decision factors for evaluating each hyperparameter: the number of parameters, the average accuracy, the number of users that performed better and the training time per 100 training samples.

The results and the times measured were all obtained on a personal computer in order to have the results faster in a test environment before it is tested in the production environment. The times measured will hold their proportion when trained on a STB (if a model takes more time than others on a personal computer, it will also take more time if both run on a slower device).

5.2.1 Dropout rate

The dropout rate is a hyperparameter that does not affect the inference in the RNN. It only affects the training. The dropout is responsible for freezing some neurons (a percentage equivalent to the dropout rate) during training, which counteracts the overfitting of the RNN. Since it only freezes some randomly selected neurons per layer, it does not affect the memory used or the training time of the network.

The insertion of dropout increases the training time since it increases the dimension of the matrices used during training by adding the dropout values. Consequently, the more extensive matrix needs more training time for the calculations performed in training. As seen in Listing 4.1, the dropout rate is a layer that is added. Moreover, as observed in Figure 5.3, changing the value of the dropout does not affect the time used in training, only the existence of dropout increases it.

The dropout works as a regularisation factor since preventing some of the neurons from being trained in each epoch makes the neurons not biased for the same input/output, making the neural network less

susceptible to becoming overfitted. Reducing the overfiteness increases the model's generalisation capability since the model is less biased towards known data.

As seen in Figure 5.3, it does not affect the number of parameters and, as a consequence, also does not affect the RAM needed to allocate the model since the RNN will always be the same and have the same amount of neurons, with the only difference of the number being neurons that are active during the training.

The training time and the number of parameters are irrelevant to the choice since, since the dropout rate does not affect them, the final value chosen was 0.5, since, as seen in Figure 5.3, not only the average accuracy is higher, but there is a higher total of users that have an higher accuracy for that value. This is within the typical values for the dropout rate and is the value that not only has the most users that perform better on the test dataset (a dataset corresponding to a week after training) but also because the average accuracy on that is better. This value provides a good generalization for the RNN and, therefore, better performance on data not used for training the RNN.

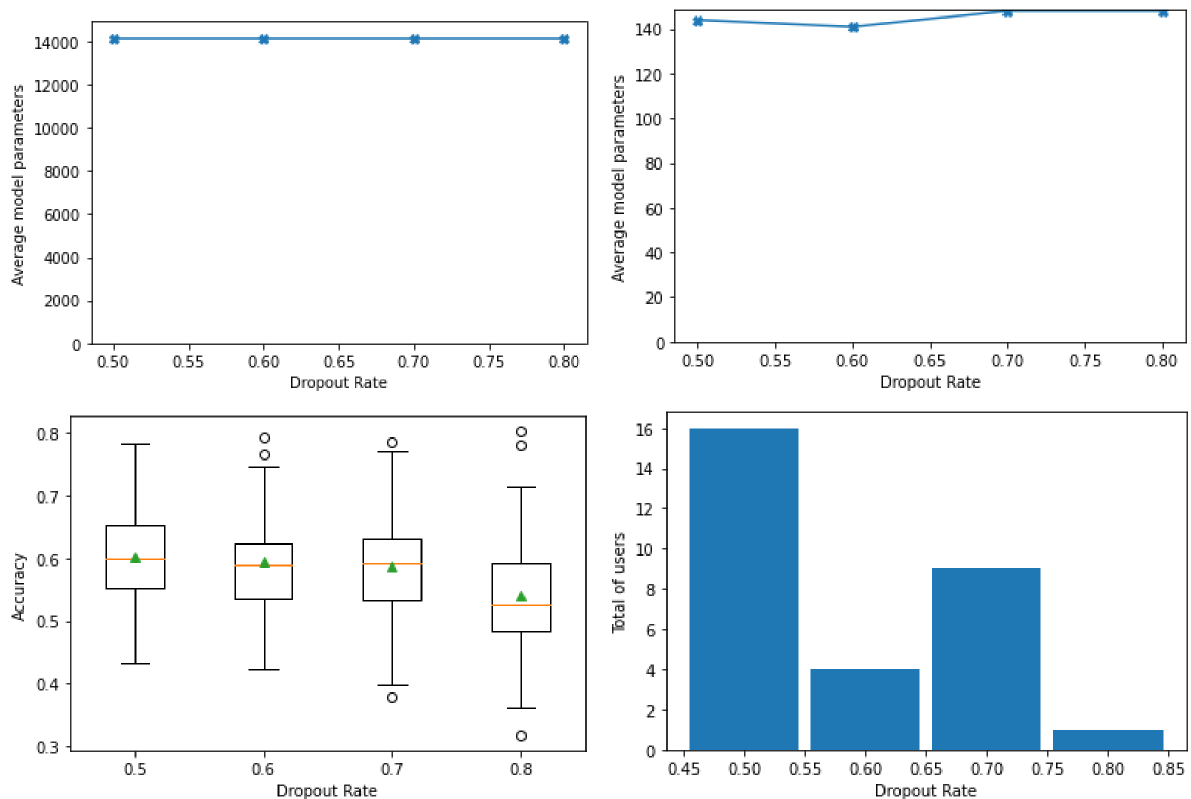


Figure 5.3: Results for the variation of the dropout rate.

5.2.2 Maximum root mean square error

As seen in the charts on Figure 5.4, increasing the minimum acceptable value for the risk on the training dataset decreases the average training time needed. This happens because the chosen RNN has two stop conditions: the maximum of 1500 epochs per train or if the risk of a particular epoch goes below the maximum RMSE threshold. Therefore, for the same RNN, increasing the maximum RMSE threshold gives the training an equal or lower duration. It is important to note that this value is multiplied by the unroll length in the Kann implementation, so a real RMSE and epoch cost (Equation (3.11)) would be calculated by dividing this value by the unroll length (four).

If the final risk of the RNN is very low, it means that the trained RNN will perform very well on the training dataset. However, that may also mean that the model will lose some generalisation properties and perform worse on the test dataset than a model trained with a higher risk/RMSE. Since the lower threshold for the RMSE is just a stop condition for the RNN training, it does not affect the model's parameters and, therefore, the memory needed to allocate the model does not change by changing this hyperparameter.

The maximum risk value that provides the best accuracy for most users is 0.5 (using RMSE as a loss function). It is also the value that provides a lower time for the model to be trained since it is the higher RMSE value.

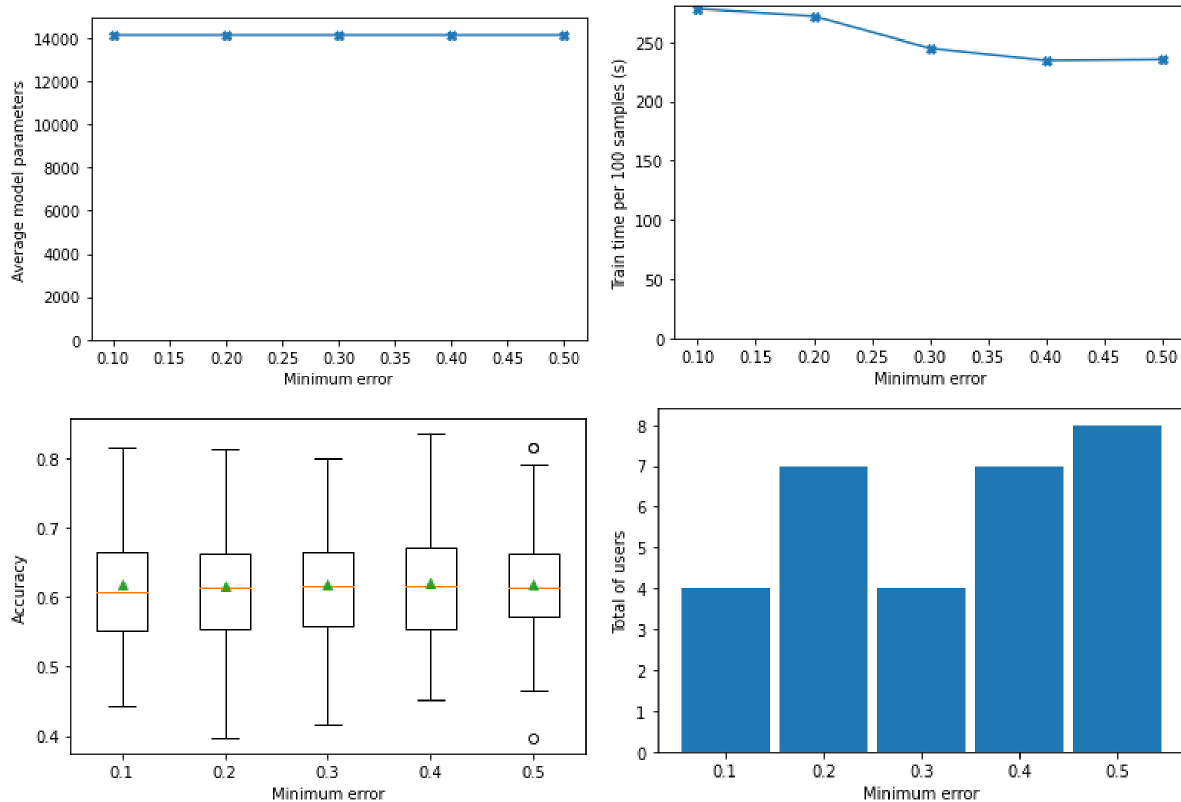


Figure 5.4: Results for the variation of the minimum RMSE

5.2.3 Layers

By analysing the Figure 5.5, having the same amount of neurons, increasing the number of layers creates a proportional increase of the number of neurons and, as a consequence, the number of parameters that the model will have. As a result of this increase of parameters, the RAM will also increase since it requires extra memory to store additional parameters. Another consequence is the increase of the average training time since performing the same amount of epochs with more layers and, therefore, more parameters, takes more time, at least for the worst case scenario (having to train 1500 epochs). The time needed for inference should also increase, yet, this value can be discarded since it is very small because it only executes once.

As seen before, a single hidden layer with multiple neurons can replicate any mathematical function using an RNN [28]. However, it does not state how easy it is to train a model with few layers. As seen in Figure 5.5, most users perform better for a model with one hidden layer, and since this is the configuration with less train time needed and fewer parameters, it was the one chosen.

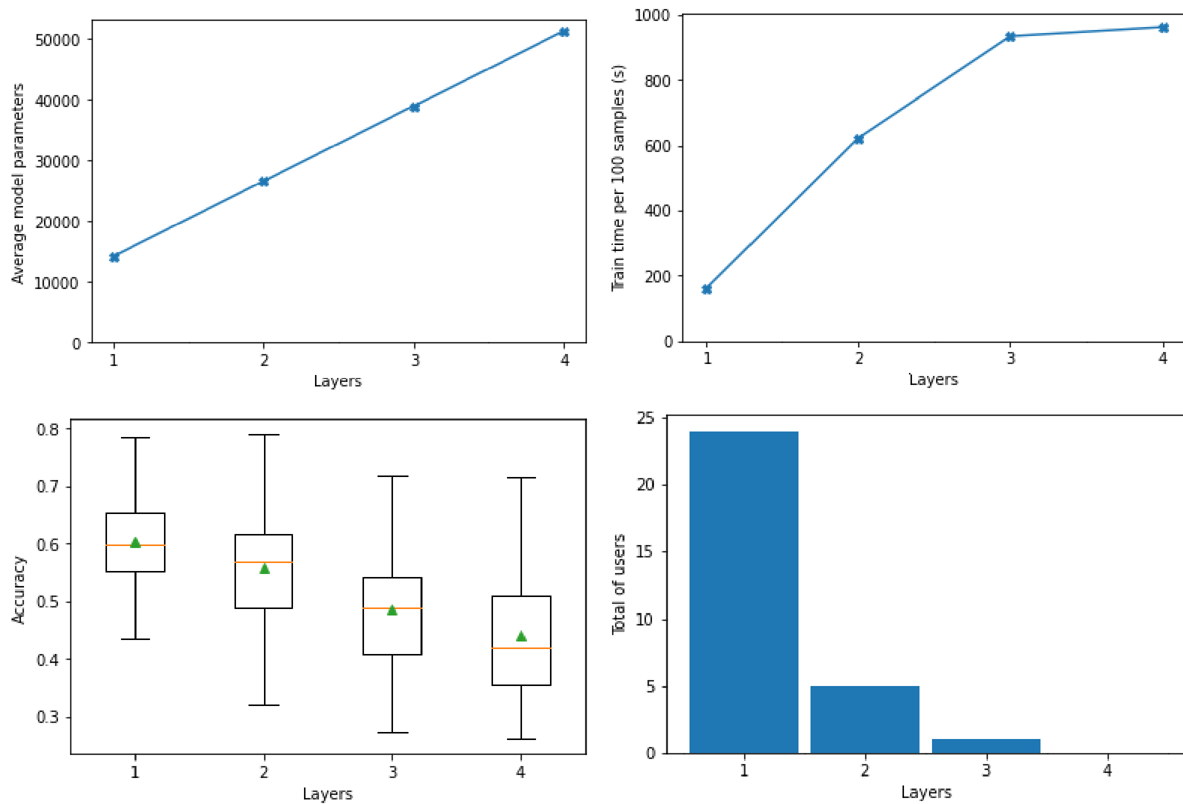


Figure 5.5: Results for the variation of the number of layers.

5.2.4 Learning rate

Learning rate is a condition used while training a NN when using the gradient descent, as shown on Equation (3.12). This value only affects how fast the model should change, and consequently, it does not affect the number of parameters the model has, as seen in Figure 5.6. Furthermore, it does not influence the training time if the same amount of epochs are performed since it is just an equation variable. Thus, the same amount of operations per epoch will be executed.

However, the train time seen in Figure 5.6 changes due to how fast the model converges and if the error hits the maximum RMSE risk or not. It is also important to notice that if the value is very high, the model will update very fast from epoch to epoch. However, it may never reach the minimum error where the RNN has better results and it may even create a gradient explosion. On the other hand, if the value is very low, closer to 0, the model may get trapped on a local minimum since it does not have a rate high enough to leave it.

Based on the obtained results, the chosen value for the learning rate is 0.01 since according to Figure 5.6, all the factors point towards it: the average accuracy is better for this value, the average train time is lower which shows that the model converges in average in a minor amount of epochs, and it is also one of the values where most users have better accuracy.

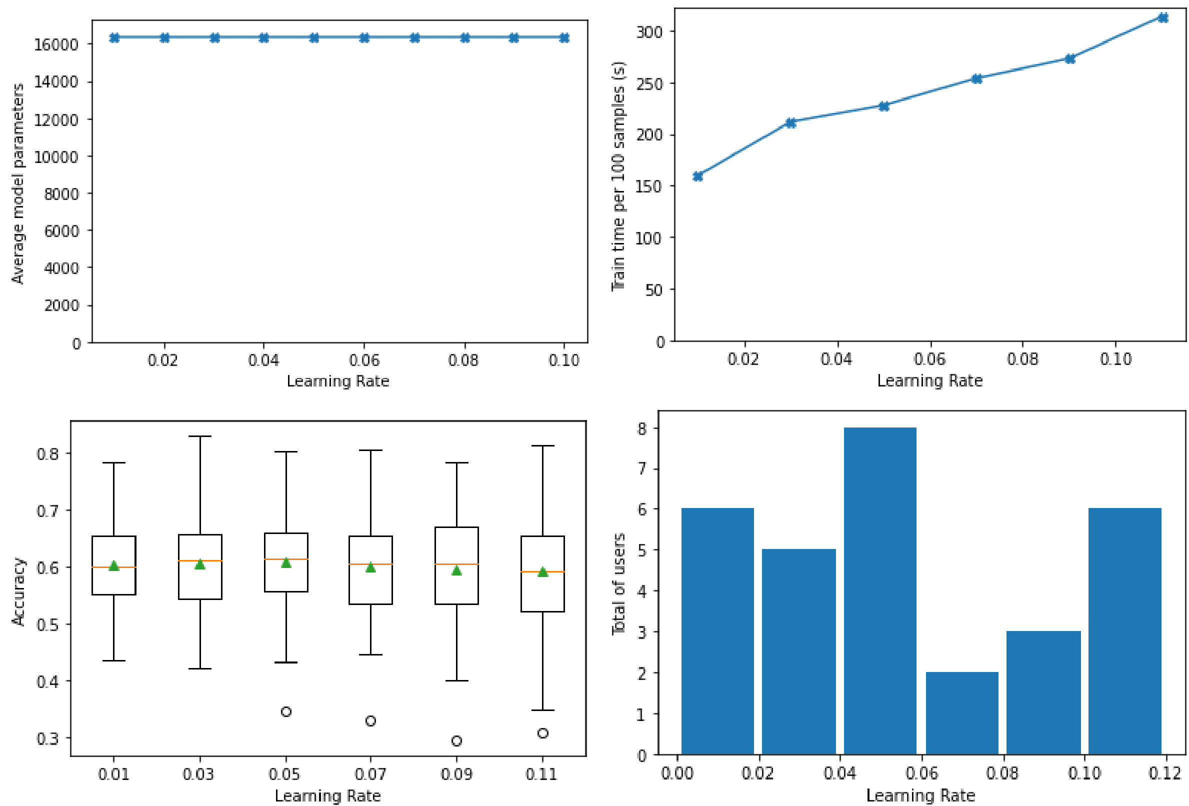


Figure 5.6: Results for the variation of the learning rate.

5.2.5 Network type

The network type has a significant effect, especially on the training time and the number of parameters. As seen in Section 3.3 the LSTM is the most complex cell, the GRU intermediate and the vanilla RNN the less complex one. Therefore, while training, it is expected that the LSTM takes more time since there are more parameters and, consequently, more multiplications during BPTT.

Theoretically, the LSTM is the one that takes more time, and experimentally it is supported by Figure 5.7 when comparing it with the vanilla RNN. The GRU should be faster, yet, it takes more time since it may be more challenging to converge and consequently need more epochs for the training. However, the extra complexity of the LSTM cells reduces the risk of the model having a gradient explosion (or a gradient vanishment) which may explain why the LSTM performs better for most of the users and why the average accuracy is slightly better. Having this in consideration, and in the fact that the extra time to train the model is in the same order, the chosen value for this hyperparameter was the LSTM cells.

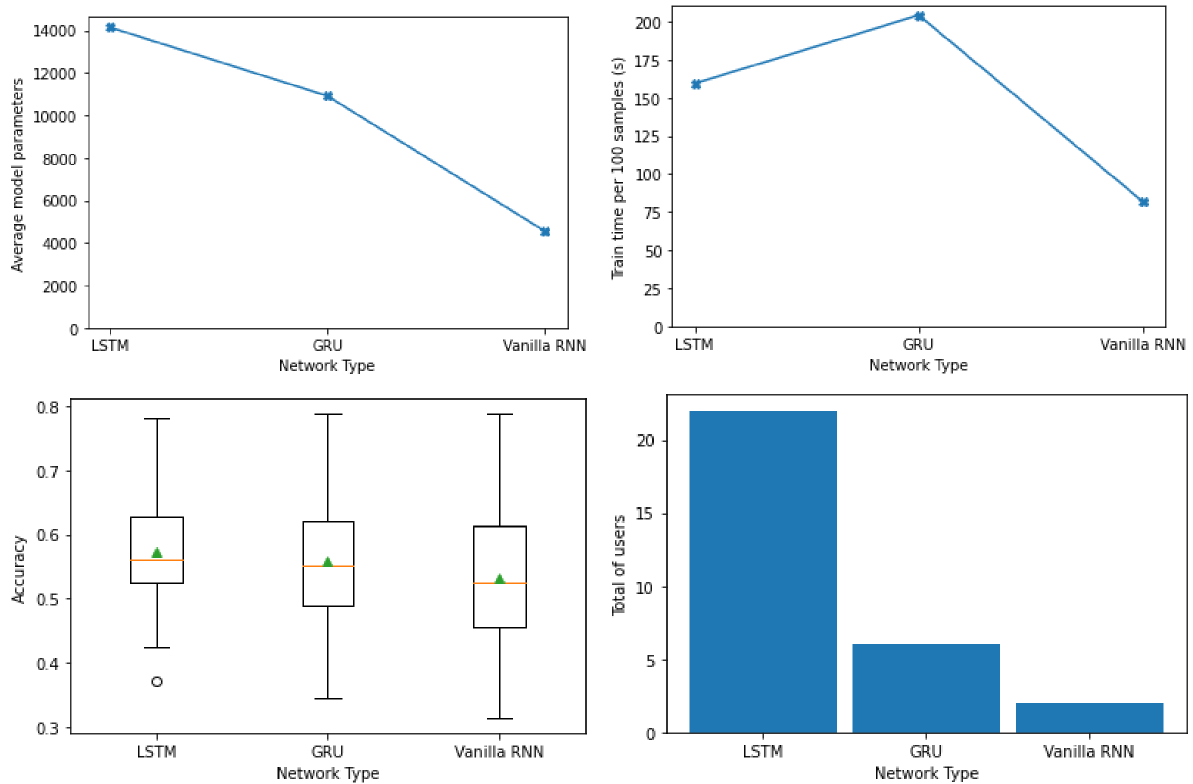


Figure 5.7: Results for the variation of the RNN type.

5.2.6 Unroll length

The unroll length determines the length of the sequence of inputs that will influence the output. In this case, it defines the number of channel changes that will influence the next channel to be watched by a given user. Empirically, the number should be high enough to have some sequence of channels but not very high since the patterns may not repeat. Because the library where the RNN is implemented uses optimised parameter sharing for training, the number of parameters, hence the memory used, does not change with the unroll length. However, even though these extra parameters are not allocated in the system memory, the training time per epoch will increase proportionally to the unroll length.

As seen in Figure 5.8 the parameters do not change if the unroll length increases and the train time tends to increase proportionally. However, unlike what was theoretically predicted, the train time is smaller with an unroll length of four than with two. This is because an epoch takes longer, but in opposition, the model converges quicker and therefore needs fewer epochs to train, resulting in a faster train with a unroll length of four than with two.

With this unexpected reduction of the train time and since it is with this value that most users perform better, the unroll length chosen was with the value of four. Therefore, the model will predict based on the last four channels seen by the user.

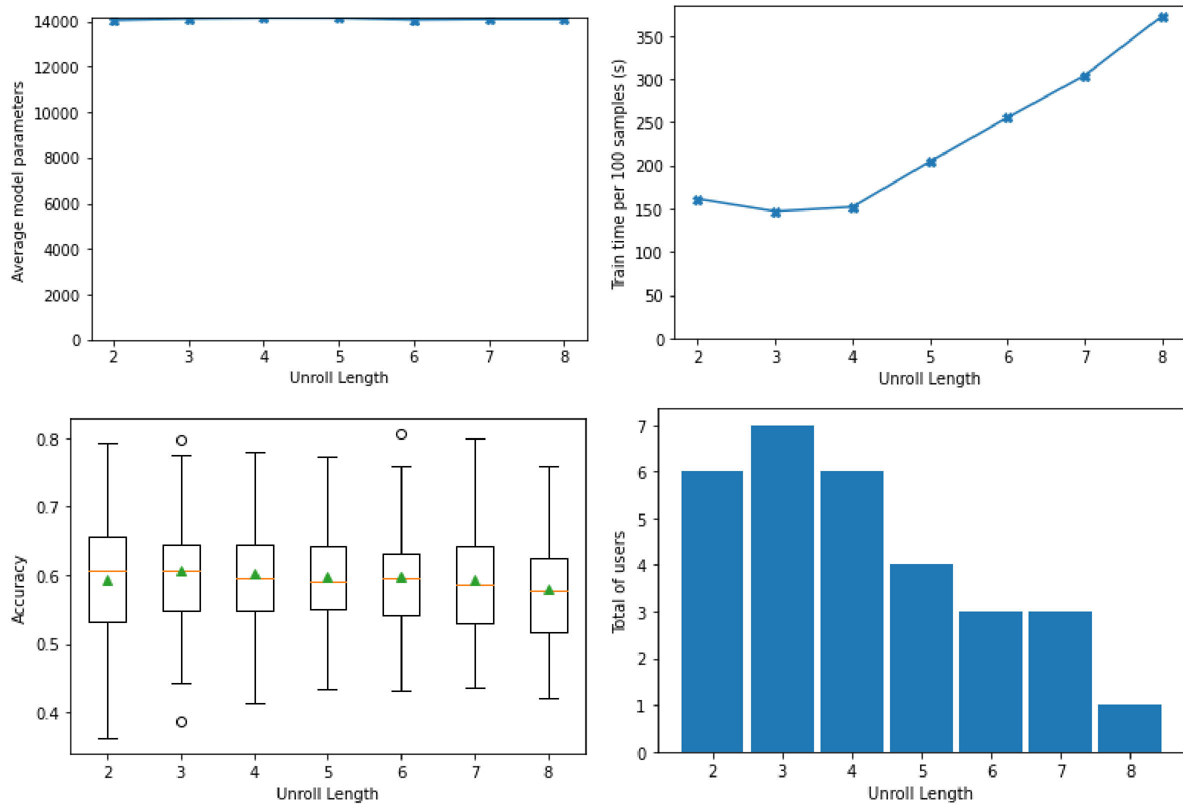


Figure 5.8: Results for the variation of the unroll length.

5.2.7 Weeks used to train

The number of weeks used for training does not influence the RNN model, its training mechanism or its hyperparameters. However, it has a critical role in defining the training dataset because it is not viable to store all channel change history in the STB since its initial set up and use it all to train the RNN. Because it does not change the model, the number of parameters will remain the same, regardless of the number of weeks used. Regarding the train, it is expected that the train time per number of samples keeps the same. However, for the same amount of new data acquired in a particular week, as seen in Figure 5.9, it is expected that the train time increases proportionally.

Having a low number of weeks makes the train dataset smaller and each channel change entry is only used to train once (for a value of one week). However, a large number creates a larger train dataset with more variety, however, some entries will be used repeatedly, which may cause some overfit to these values. It is theoretically expected that a good value is a balance between repeating the same data (which may be outdated) and having a large and diverse enough dataset.

As seen in Figure 5.9, the train time increases with the number of weeks, and so does the accuracy. The accuracy may be explained since sometimes the model needs to be rebuilt if new channels appear, therefore, this repetition of channels in training does not occur as often as theoretically expected. With

this possibility of the model being rebuilt and better performance for a majority of users in consideration, it was chosen five weeks as a value for this hyperparameter. Higher values for the hyperparameter are not considered in Figure 5.9 since the train time would increase to values that may significantly impact the STB usability.

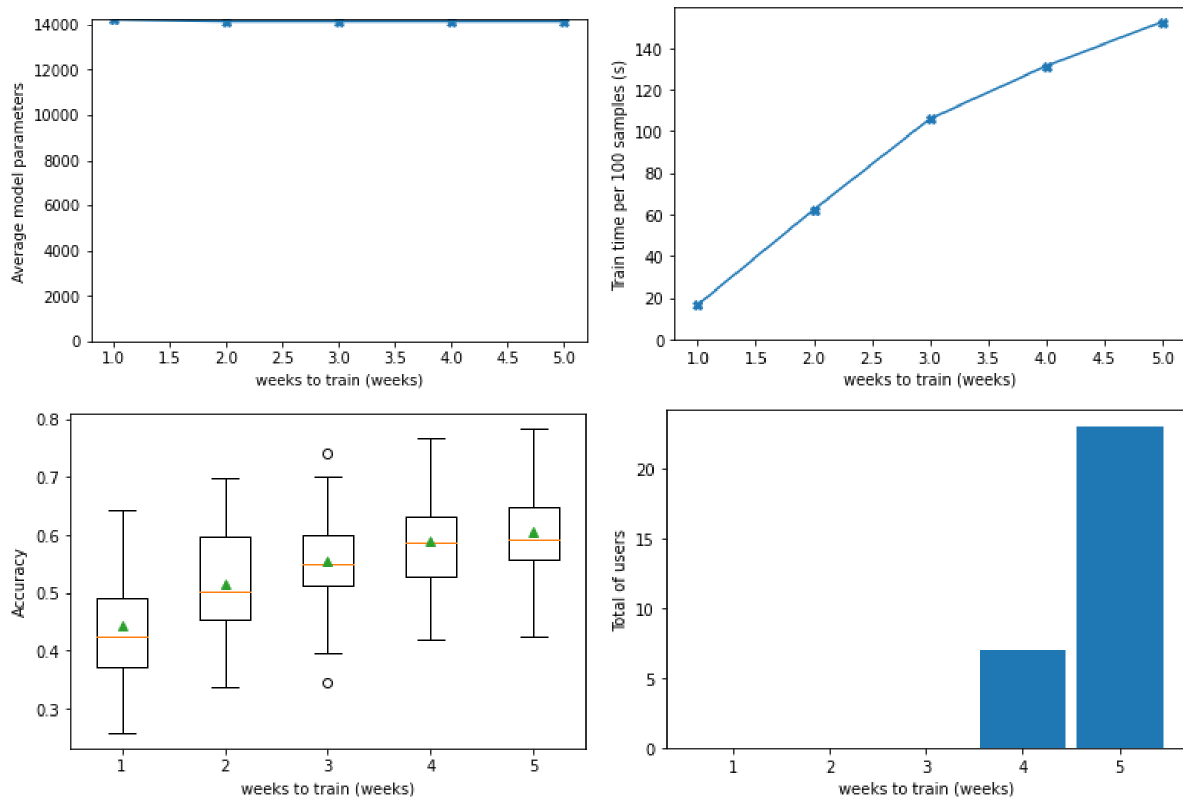


Figure 5.9: Results for the variation of the number of weeks used to train.

5.2.8 Other hyperparameters and decisions

For the full functioning of the model in terms of training, inference and data collection over several weeks, some other small decisions needed to be taken without having a rigorous analysis like the previous hyperparameters.

Regarding training, the RMSE minimum value works well if the RNN converges and the time taken to train is low. However, it is possible to have the training not converge, and in a different case, the model may take too much time to train. If the model is not converging, training for extra epochs keeps on overfitting the model. Therefore, a new train stop condition of a maximum of 1500 epochs was added, a value not too big and where the error does not decrease much more if the minimum RMSE was not yet extinguished. For the case where an epoch takes a long time to train, and even 1500 epochs have a long duration, it was created a maximum time of 1.5 hours for the model to train, after it, the model stops

Table 5.2: Final hyperparameter values.

Hyperparameter	Value
ulen	4
max_train_rmse	0.5
weeksToTrain	5
n_layers	1
networkType	LSTM
learningRate	0.01
dropoutRate	0.5

training once the current epoch is finished. This option was taken instead of truncating the dataset since a model training in more than 1.5 hours is rare and a worst case scenario. Besides, truncating would probably lead to ignoring data, despite the model converging faster. Furthermore, it is better to train with a more extensive dataset than with a smaller one for a larger number of epochs.

The wall clock time at which the training is to be performed also needs to be selected to a time that does not significantly affect the STB usability (e.g. 1:30 AM on Sunday since it is when fewer people are watching TV [22]). This value can be changed to a different time without affecting the model since it is trained weekly.

For the RNN architecture, the only hyperparameters that were not assessed were the maximum number of channels that a model will consider when using the channel mapping and the number of neurons in the hidden layers. The maximum amount of channels chosen was 50 since, by looking into Figure 5.1, not many users have more than this number of channels, and therefore the model is rebuilt fewer times, which is also supported by [25]. Furthermore, it is a value that does not consume much memory. Regarding the number of neurons in the hidden layer, this value is correlated to the number of inputs and, therefore, the size of the mapping. To have a good proportion of neurons in the input, hidden, and output layers the number of neurons in the hidden layer was set to the same as in the input layer. Having this value adaptable to each user creates more flexibility for the RNN since it is not a hard-coded value that is common to all users (e.g: a user that watches only five channels or, in opposition, a user that watches fifty different channels).

5.2.9 Summary

After considering the analysis for each hyperparameter, the final results, where the values got stable, for each analysed hyperparameter is presented in Table 5.2.

6

Results and analysis

For measuring the results of applying this model to the acquired data explained in Chapter 5, it was used the final hyperparameters configurations presented in Table 5.2 as it is the last stable version and the one that would be chosen to deploy this model. The results are divided into two parts. Firstly, the model was applied to three specifically chosen users to understand how it works in favourable and unfavourable scenarios. For these users, the model was performed entirely on the Raspberry Pi 3 to simulate a STB. The second part contains the analysis of the results of the users divided by groups where the benefit and costs are analysed. Furthermore, the restrictions presented in Section 1.2 are validated and analysed at what point this implementation benefits the end user.

6.1 User simulation

For developing and testing the proposed solution and evaluating it, it is necessary to perform the model for all users. To assess one user and to be as realistic as possible. The provided data contains the history of all the channel change operations for a set of STBs. The data is then separated into a log file per user with the data for one user. Then, it is possible to create a scenario where several weeks

correspond to the training dataset and only a week is used for the test dataset. The week of data used for testing will be incorporated into the training dataset on the next iteration. Therefore the test data is used for training, but not the opposite since these datasets are created chronologically.

In a non-simulation environment, the user will be changing the channel via a TV remote for the testing, so the data will not be available immediately. However, for the simulation, the data is already available. The training dataset used will be created for each week with the history of channel changes and the test dataset is created with the channel changes of the subsequent week.

Algorithm 6.1: Simulation for one user.

```
data ← loadData(file, user);
testData ← null; trainData ← null; rnn ← null; mappingChannel ← null ;
for week ← 0; week < getTotalWeeks(data)-1; week++ do
    trainData ← getNextWeekData(data, week - hyperparameter.weeksToTrain, week);
    testData ← getNextWeekData(data, week,week + 1);
    if rnn == null OR isNewMappingNeeded(rnn, testData, maxMappingSize then
        mappingChannel ← getNewMappingChannels(data,week,maxMappingSize);
        rnn ← model_gen();
    train(trainData,rnn, hyperparameters);
    applyAndPrintResults(testData,rnn,hyperparameters.ulen);
    freeVector(trainData);
    freeVector(testData);
freeVector(data);
```

On Algorithm 6.1, an auxiliary structure of type `Vector` was created containing the list of channels seen by the user with the respective time information. After, a loop will occur where the training and test datasets are allocated and freed in each iteration. An iteration in this loop represents a week of visualisations for the user. The training dataset is created with data from the previous weeks (the number of weeks used in the train is a hyperparameter), and the test dataset is made with the subsequent week, simulating a real usage experience. Then the model is trained with the training dataset and the inference is executed by using `applyAndPrint()` function with the test dataset, the accuracy values are also retrieved. The cycle will perform for the number of weeks of data that the user has minus one since, on the last one, there will not be a correspondent test week to evaluate the model.

6.2 Individual results

The users considered to be interpreted separately are the artificial user, created to verify if the model works, one of the users with better accuracy, with data that can be visualised was used to see an actual case where the model works well. One of the worst users, also with data that is possible to visualise, was used to represent when the model does not work well. The results can be seen in Table 6.6. For each one of these users, it is plotted the training curves and a representation of the first channel predicted by

the model and the respective real channel for each week on Appendix B in addition to a table, in each subsection, with the summed accuracy results and training times for each week per user.

6.2.1 Artificial user

As seen before, the artificial user, described in Section 5.1.1, is a user whose channel changes are fabricated to have a straightforward pattern. This user was created to have a proof of concept for the model's functioning. The user only has nine different channels, and 272 channel changes occurred during eight weeks. The accuracy results are presented in Table 6.1 and the results' visualisation is presented on Figure B.2 to Figure B.9.

On Figure B.1, it is possible to notice that the model always converges to the minimum RMSE defined and that the pace and number of epochs required are very high and low, respectively, only needing around 20 epochs to converge. This is because of the number of channels the user watches and, therefore, the number of classes of the model is meagre. Furthermore, few samples are processed, making the training datasets small. Adding the fact that the pattern is very repeatable creates a model that trains very fast, and as seen in Table 6.1, the training time needed is almost irrelevant. It is also possible to notice that the model has very high accuracy.

As seen in Table 6.1, the second week has the lowest accuracy, only 43% if only one channel is fetched. This lower result can be explained by looking into the data displayed on Figure B.2 and Figure B.3. In the first week, the user did not see channels 8 and 9. Therefore, the model was not able to predict it. Furthermore, the first week lacks the first days of the week, so the model cannot learn some patterns. It is possible to see on Figure B.4 that the model learned the patterns for channels 8 and 9, which were missing before, on the third week. After this week, the data has very few changes. On week 5 in Figure B.6, the accuracy is almost perfect since all the occurrences were very similar to ones found in the previous week, failing only once when a user watched a channel that was not common for that time, and the model predicted the usual channel.

Table 6.1: Accuracies obtained for the artificial user per week and train times measured in the Raspberry Pi 3.

Week	1 Channel (%)	2 Channels (%)	3 Channels (%)	4 Channels (%)	5 Channels (%)	Train Time (s)
2	42.86	54.29	62.86	74.29	74.29	0.14
3	74.29	91.43	94.29	97.14	100.00	0.32
4	72.22	77.78	83.33	91.67	94.44	0.38
5	87.50	90.62	93.75	93.75	96.88	0.44
6	96.97	100.00	100.00	100.00	100.00	0.07
7	85.71	88.57	91.43	94.29	94.29	0.00
8	76.47	84.31	92.16	96.08	96.08	0.47

6.2.2 Best user

For the best user, it was chosen a user with a good performance and where it is possible to visualise the data. This user does not have the last week, and therefore its data is separated into seven different weeks. The channel changes can be seen from Figure B.11 to Figure B.17. The user sees a total of 33 different channels, and the user changes channels 510 times during the seven weeks.

As it is possible to see in Figure B.10, the model converges to the minimum defined RMSE before the 1500 epochs in every training iteration and the error where the model starts training each week decreases from week to week. This is because the data does not change its pattern suddenly, so the information from previous weeks is not wasted, making the training need fewer epochs. On the same note, it is possible to see that except for the last week, the accuracy shown on Table 6.2 increases each week. Regarding the training time presented in Table 6.2 it is noticeable that the time increases as the weeks pass since the training dataset increases, culminating in a peak of 2 minutes. However, in weeks six and seven, the training time diminishes again since the error where it starts training is very low. This happens since the pattern does not change much in these last weeks, so the model behaves as it was already trained.

For the second week, shown in Figure B.12 the model performs with an accuracy of 43.18 % as seen in Table 6.2. This is primarily because on Sunday and Monday (2021-01-10 and 2021-01-11), the user sees a large number of channels in the same sequence as in the first week (Figure B.11). For the rest of the week, the user also sees new channels which are not possible for the model to predict since it was not trained with them. Furthermore, it is noticed that the user on this particular week did not see any channel for two days. On Saturday of this week (2021-01-16), the model tries to predict the same pattern observed on Figure B.11, especially notable on channel 22. However, the user saw different channels causing the model to fail.

In the initial part of the third week (Figure B.13) and the rest of the weeks, the model performs very well, reaching 97.3 % in the fifth week and a 100 % accuracy in the sixth week with the model fetching only one channel. Notably, these weeks differ from the first weeks since the data is much more sparse, having much fewer channels that the user does not regularly watch, which creates noise for the model.

It is possible to notice that the significant part of the correct predictions is contained on days the user watched the same channel on the same day of the week on the previous week, the same channel at the same time on another weekday or the same sequence of channels. For the same reason, primarily pointing to week 3, the model failed by predicting similar patterns that occurred in previous weeks that did not happen on that one.

For the last week, the model had an accuracy of 63.64 %. This value is lower than the previous weeks and is a combination of the user starting to watch new channels and the week having very little data.

Table 6.2: Accuracies obtained for the best user per week and train times measured in the Raspberry Pi 3.

Week	1 Channel (%)	2 Channels (%)	3 Channels (%)	4 Channels (%)	5 Channels (%)	Train Time (s)
2	43.18	51.14	63.64	69.32	71.59	26.29
3	71.43	75.51	77.55	77.55	81.63	59.48
4	87.23	87.23	89.36	91.49	93.62	77.87
5	97.30	97.30	97.30	97.30	97.30	127.22
6	100.00	100.00	100.00	100.00	100.00	141.46
7	63.64	63.64	63.64	63.64	63.64	0.00

6.2.3 Wort user

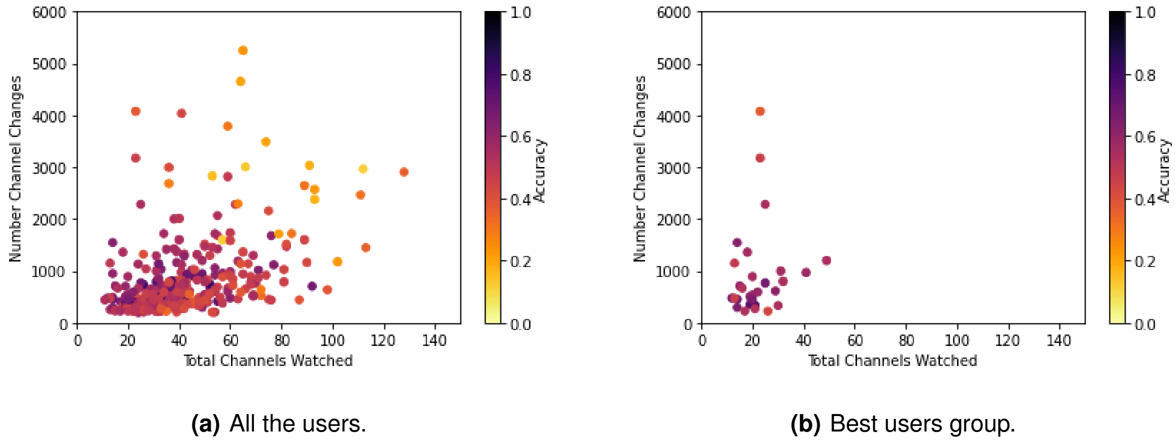
For the worst user example was chosen a user that watched 111 different channels and changed channels 2465 times within five weeks. This user has the best result in the seventh week with a 15.42 % accuracy if only one channel is fetched and 39.13 % accuracy if five channels are fetched. The remaining accuracies are shown in Table 6.3.

At first glance, by looking into Figure B.18, it is possible to see that the training does not converge to the minimum RMSE defined and, therefore, that the training always uses 1500 epochs or the train stops because it goes over one hour and a half. Furthermore, the training error has almost not decreased in the last weeks. This is primarily because the user does not always change the same channels. For example, it is possible to see that in the second week Figure B.20, the user saw 40 channels not included previously and some extra 20 in the third week Figure B.21. This behaviour saturates the 50 channel cap for the model and, most relevant, create a large number of patterns that are not regular, and therefore the model cannot learn.

As seen in Figure B.18, the model does not converge for this user never, and the RMSE does not decrease steadily as in Figure B.10. The training time for the first week is swift since the dataset is minimal, but the train reaches one hour and a half for weeks 2, 3 and 8. For the other weeks, the train stops because it performs the 1500 epochs, but the train time is also considered since it can not converge.

Table 6.3: Accuracies obtained for the worst user per week and train times measured in the Raspberry Pi 3.

Week	1 Channel (%)	2 Channels (%)	3 Channels (%)	4 Channels (%)	5 Channels (%)	Train Time (s)
2	1.84	3.68	4.41	6.62	8.82	4.46
3	7.24	10.58	15.88	19.22	20.89	2951.86
4	9.17	15.21	20.58	24.83	29.53	5400.33
5	12.15	19.44	24.65	28.47	31.25	5405.19
6	15.86	23.26	28.75	33.19	38.05	5411.30
7	11.92	18.91	24.35	27.72	31.35	5413.39
8	15.42	26.48	32.41	35.57	39.13	5404.50



(a) All the users.

(b) Best users group.

Figure 6.1: Heat map regarding the accuracy for one channel, number of channels watched and number of channel changes.

6.3 General results

After evaluating the niche of users individually, it is crucial to assert how a more significant number of users perform. The summed results are presented on Table 6.6, and it is possible to see the accuracy results with only one channel fetched according to the dataset size and the number of channels seen for all the available users represented in Figure 6.1(a) and for the best users group in Figure 6.1(b). The average accuracy is 50.2 % for one channel fetched, and it increases to 67.7 % if five channels are brought. However, it also increases the bandwidth consumption and the costs of receiving multiple channels for the STB. The best 30 users group changes from 57.3% to 83.8 %, and the bandwidth consumption changes the same way.

By comparing Figure 6.1(b) and Figure 6.1(a), it is possible to see that the users that perform better and therefore are included in the best 30 users group (this group does not have the best 30 final users since it was created previously to the hyperparameters decision) are users that do not see a large number of different channels. This is evident on Figure 6.1(a) since most users that watch more than 80 channels have poor accuracy. However, there are a few outliers, and this can be explained by the number of times the user sees each channel, if a user sees ten channels regularly and the other 70 only once, the model will mispredict those 70 channels but can predict the regular 10, increasing the accuracy. It is also noticed that users with many channel changes (higher than 2500) tend not to perform very well. This can be explained since the scope of the dataset only has two months; therefore, 2500 channel changes within this time frame correspond to an average of 41 channel changes per day and 312 per week. This number is very high, and there is a significant possibility that the user is constantly changing patterns and creating noise in the model.

Considering the average accuracies for each user, it is possible to get to know how beneficial it is

for a user to have this model implemented. As seen before, when the channel is already available for the user to see, it takes nearly 0 s to change the channel. However, if the channel is not ready yet, the change takes upon 2 s [1]. Therefore, it is possible to calculate the average channel change delay for a particular user using Equation (6.1).

$$ChangeDelay_{average} = (1 - accuracy_{average}) * delay_{notAvailableChannel} \tag{6.1}$$

On Figure 6.2 it is presented the normal curves for the average accuracies dispersion for the group that contains all users and created the correspondent average delay time by using Equation (6.1). The average delay time only considers when the model starts working (second week). A normal distribution was created based on the accuracies if it is being fetched from one to five channels, and the respective mean and standard deviation are presented in the figure label based on accuracy. A second axis was also created with the correspondent average channel change delay. As expected, once the number of channels increases, the mean accuracy also increases; therefore, the average channel change delay reduces. Regarding the standard deviation, its values do not change much (it surrounds 0.12), which means that the dispersion of the accuracies from the mean does not differ much with changing the number of channels.

Because the average accuracy for one channel fetched is 0.502, the average delay time for changing channels will drop from 2 s to 0.996 s. For the case where it fetches five channels, it falls to 0.646 s since the accuracy is 0.677. By correlating the information from all the datasets (number of users, number of channel changes and number of channel changes for each user), it is possible to affirm that this model would save (with only one channel fetched) on average 0.26 hours for each user and a total of 78.6 hours for the 300 users together, during the two months that the dataset cover. Regarding the user that saves more time (combining its accuracy and number of channel changes), it will save 1.37 hours, and on the opposite note, the user that saves less time would only save 2.02 minutes.

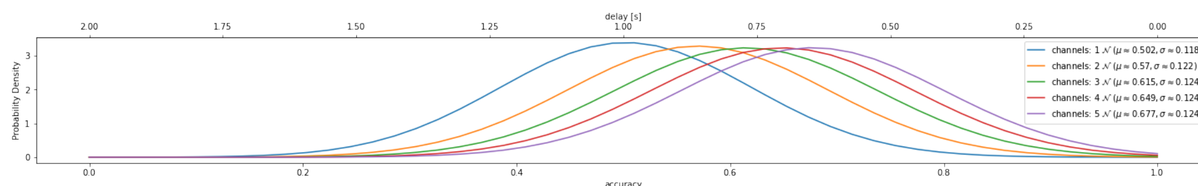


Figure 6.2: Normal curves representing the dispersion of the accuracy and correspondent channel change delay for all the users.

6.4 Validation on embedded devices

Apart from the results on Table 6.6, all the experiments have the common property of the disk space used to have the source code and the executable ready to run. The source code (with the Kann library) and the executable require 401 KB which can easily fit in a low-resource environment. Furthermore, it is only needed to have the dependent libraries, which are standard libraries for a Unix environment, such as math.h, string.h, stdlib.h, stdio.h and assert.h. Plus, it is necessary to have space for the logs for each user. However, the space for these logs was not measured since they are already being stored, so there is no need for extra space.

On the RAM side, the maximum used is 2.4 MB. This value represents the memory needed to allocate the RNN and the training dataset for the training part. The peak RAM value is low enough for a STB. This value (and the other columns from the table apart from the accuracy) is independent of the number of channels that the model predicts since the model only runs once, returning an ordered by accuracy list of channels and from that is chosen the number of channels wanted.

Furthermore, the model was also tested in a Raspberry Pi 3 with an ARM Cortex-A53. It was also used during these performance tests on three other embedded devices: a Raspberry Pi 4 model B with a quad core ARM Cortex-A72 CPU operating at 1.5 GHz and an ARM processor with a Cortex-A9 dual core embedded in a FPGA Zynq. It was also used a Raspberry Pi 1 with an ARM1176JZF-S, single core, with 512 MB RAM operating at 700 MHz. The last devices tested were a personal computer with a i7-5500U CPU operating at 2.4 GHZ with a limitation of 50 % on the CPU usage and a Raspberry Pi 2 model B with a quad-core ARM Cortex-A7 operating at 900 MHZ.

However, it was only possible to test for the three individual users, a user with the same average time as the average time of the all users group and another for the best 30 users group. Regarding the RAM and disk space, it does not change if the model performs in an ARM Cortex-A53 or a i7-5500U. However, due to processor limitations, the training time will increase. On Table 6.4 the training times are exposed for the average training time on the previously refereed devices. Regarding the inference time, the time taken to predict one channel is almost none and therefore it can be discarded.

It is possible to observe on Table 6.4 that the times are slower on ARM Cortex-A53. However, the models (apart from the Worst User) do not tend to saturate at the one hour and half stop condition. Therefore the number of epochs trained was similar to the ones on the i7-5500U, and the accuracies sustained. However, for the all users group, the average train time is 3457 s, so 57 min. This means most users do not require an entire hour and a half to complete the training. This is another validation that the model can also train in good time in this low restrictions environment.

Regarding the other embedded systems, the ARM Cortex-A72 provided performance results in the same order as the i7-5500U, being capable of not saturating the hour and a half limit on the worst user scenario for 6 weeks. This embedded system has a higher computer power than the rest, so it is

Table 6.4: Training times in different embedded devices.

	Average train time (s)				
	i7-5500U	**ARM Cortex-A72**	**ARM Cortex-A53**	**ARM Cortex-A9**	**ARM Cortex-A7**
Best user	17	14	72	133	303
Worst user	2240	1705	4281	4501	4608
Artificial user	0.18	0.20	0.19	0.30	0.91
All users	441	627	1465	2758	3888
Best 30 users	238	310	506	546	1614

expected to be faster. At last, the ARM Cortex-A53 embedded saturated for the worst user as expected, but it provided an average of 2758 s for all the users, so, 45 minutes which is a very acceptable value.

In sum, the restrictions defined in Section 1.2 are passed for all the groups presented in Table 6.6, and it was possible to execute the models in different embedded systems with restrictions very similar to a STB. Leaving to decide the balance between the desired accuracy and the time saved for a channel simultaneously changes the cost of fetching the corresponding number of channels.

6.5 Overall results and comparison with other models

On Table 6.6, it is possible to see the average accuracies for each group of users from one to five predicted channels, the average number of parameters on the created network and other specific information. It is possible to see that for the 300 users, it obtained 50.2 % accuracy with only one channel being predicted and 67.7 % if five channels were predicted. This accuracy is measured by the average accuracies of each week for each user.

For the 270 users, the accuracy obtained was 49.3 % if only one channel is fetched and it increases to 65.7 % if five channels are considered. These values are smaller than the one obtained for the all users group since it does not include the users where the RNN was designed for when choosing the hyperparameters. This value shows that the model can generalise for users whom the RNN were not specifically designed for.

As expected for the best 30 users, the accuracy is considerably higher (57.3% and 83.8 % for one and five channels, respectively). This is because of the process of creating this set of users. Regarding the number of parameters, the worst user has the higher average of parameters, this happens since the number of parameters increases with the number of different channels seen by a user and, therefore, with the number of input and output classes, which also influence the number of neurons per hidden layer. These values are all connected to the number of channels seen (but it saturates at 50), so the worst user watches more than 50 channels and therefore saturates this number. It is also noticed that the higher the parameters, the worst the accuracy. This is not a rule, however, since, as seen in Figure 6.1(a) and Figure 6.1(b), the users with a more significant amount of different channels seen tend to perform

worst and therefore the users with more parameters on the model tend to perform worst.

Regarding the train and test size, it is possible to see a similar amount of samples to train in each (the test represents the new samples that will be added next week). The average of 141 is good for the model to train weekly, however, if it were supposed to train daily, it would not have a relevant dataset and most likely overfit the model.

On Table 6.5, it is possible to see how the model performs when comparing the state-of-the-art models presented in Chapter 2. It is important to notice that for the models presented in [5] and [25], the accuracy was calculated in a different dataset (the one present on each respective paper) therefore, it can not be directly compared. For the up and down model, the presented results are a summary of Table 2.1 but only considering the best combination of channels for one, two or three channels to be predicted. This last accuracy was obtained on the same dataset as this thesis model.

Table 6.5: Accuracy results comparison between the model proposed in this thesis and the ones defined in the state of art.

Number of Predicted Channels	Thesis Proposal (%)	Up and Down [17] (%)	J48 Default [5] (%)	Ada Boost [5] (%)	CL [25] (%)	SL [25] (%)
1	50.2	12.2	37.39	37.39	19.27	22.23
2	56.9	23.5	NA	NA	NA	NA
3	61.4	31.2	NA	NA	23.84	47.98
4	64.8	NA	NA	NA	NA	NA
5	67.7	NA	NA	NA	54.78	63.55

It is noticed that the usage of the implemented RNN for a low resource environment outperformed the up and down model and the tree-based models trained using an outside computer. It is noticed that the accuracy between the proposed model and the one presented in [25] is higher for one channel, although it is very similar for five channels. This is primarily due to the similarity of these two models (both are LSTM based). Furthermore, it shows that implementing the model on a low-resource environment using the Kann framework provides better or similar accuracies than the model trained without restrictions.

Table 6.6: Overall results.

Group		Results						
Users	Number of Fetched Channels	Number of Users	Accuracy (%)	Average Number of RNN parameters	Average Epochs	Average Train Dataset Size	Average Test Dataset Size	Maximum RAM used (MB)
All Users	1	300	50.2	28061	441	400	141	2.4
All Users	2	300	56.9	28061	441	400	141	2.4
All Users	3	300	61.4	28061	441	400	141	2.4
All Users	4	300	64.8	28061	441	400	141	2.4
All Users	5	300	67.7	28061	441	400	141	2.4
270 Users	1	270	49.3	29067	463	401	140	2.4
270 Users	2	270	55.3	29067	463	401	140	2.4
270 Users	3	270	59.8	29067	463	401	140	2.4
270 Users	4	270	63.0	29067	463	401	140	2.4
270 Users	5	270	65.7	29067	463	401	140	2.4
Best 30 Users	1	30	57.3	14141	238	390	150	1.8
Best 30 Users	2	30	69.0	14141	238	390	150	1.8
Best 30 Users	3	30	75.5	14141	238	390	150	1.8
Best 30 Users	4	30	80.2	14141	238	390	150	1.8
Best 30 Users	5	30	83.8	14141	238	390	150	1.8
Artificial User	1	1	83.5	2871	9	122	52	0.5
Artificial User	2	1	88.3	2871	9	122	52	0.5
Artificial User	3	1	93.1	2871	9	122	52	0.5
Artificial User	4	1	94.9	2871	9	122	52	0.5
Artificial User	5	1	95.2	2871	9	122	52	0.5
Worst User	1	1	10.7	49066	1288	1061	358	2
Worst User	2	1	17.0	49066	1288	1061	358	2
Worst User	3	1	21.9	49066	1288	1061	358	2
Worst User	4	1	25.5	49066	1288	1061	358	2
Worst User	5	1	29.0	49066	1288	1061	358	2
Best User	1	1	81.6	15783	82	381	82	1.6
Best User	2	1	83.7	15783	82	381	82	1.6
Best User	3	1	86.8	15783	82	381	82	1.6
Best User	4	1	88.3	15783	82	381	82	1.6
Best User	5	1	90.2	15783	82	381	82	1.6

7

Conclusion and Future Work

7.1 Conclusion

This thesis presented a predictive model based on a RNN suitable for implementations in STBs for the prediction of the channels a user most likely would like to watch at a given moment and, therefore, help reduce the channel zapping delay and improve the user experience while watching digital TV. This model was designed to have its lifecycle completely inside a STB, so it not only uses the logs of the channel changes for the RNN training in the STB but also works in real-time in the STB to predict the channel that the user would like to watch when going for a channel change.

When developing the proposed prediction model, special attention was given to the involved performance and memory requirements, owing to the strict resource-constrained nature of STBs. Accordingly, the devised mode was built using the Kann library [58], a lightweight C library built with the minimal dependencies possible. The resultant code implementing the RNN was also built using the C programming language, not only for efficiency reasons but also to be portable to other new low-resource environments. As a result, the final model implementation requires only 401 KB of disk space (with the Kann library included) and a peak RAM usage of 2.4 MB. Furthermore, this model was validated inside four embedded

systems, which their hardware configurations resemble a STB.

To test the model, three different embedded systems were used (with hardware configurations similar to standard STBs) and data from 300 different users with channel change logs acquired during two months were considered. Regarding the hyperparameters achieved, it was used a LSTM that predicts based on the last four watched channels by the user.

The obtained experimental results show that the average accuracy of the proposed model for predicting only one channel is 50.2 %. The prediction accuracy increases to 67.7 % if five channels are simultaneously predicted. However, predicting such extra channels also has a higher cost. Hence, it is the designer's responsibility to choose the NN configuration providing the desired trade-off between prediction accuracy and implementation cost. The prediction accuracy of the proposed model surpasses the results of the most common models (up and down) by 38 % for one channel and 30.2 % for three channels. It also outperforms the tree-based models [5] in 12.81 %. When compared to a more general LSTM model not implemented in a STB [25] the results provided by the proposed model are also better: 27.97 % for one channel and 4.15 % for five channels. Noticed that the up and down model was implemented on the same dataset, but the rest of the results were obtained on each research dataset.

In conclusion, the obtained experimental results show the benefits of using a RNN for predicting the next channel and the importance of having recurrency in this type of prediction system. In addition, they show that using the considered data the users would have saved, on average, 0.26 hours of time spent waiting for a channel to change.

7.2 Future work

For future work, the main suggestion is to implement the proposed model in a real STB being used by real users and assess not only the accuracy of the model in a real operation scenario but also its benefits for the QoE of the users.

Another interesting research direction would be to assess the actual cost of receiving an extra channel and to mitigate it, the new research should study the possibility of stopping receiving those channels at a given moment with its cost reduction and possible influence on the model accuracy.

Choosing the best moment to train the model is another relevant task for future work. Currently, the model is being trained once a week, at a moment when the user is less likely to be watching TV. However, it would be quite interesting to find the best training moment for each STB user. Furthermore, the model could also be extended to detect significant errors in its predictions and trigger the RNN retraining at the next not busy moment.

Bibliography

[1] F. M. Ramos, J. Crowcroft, R. J. Gibbens, P. Rodriguez, and I. H. White, "Reducing channel change delay in IPTV by predictive pre-joining of TV channels," *Signal Processing: Image Communication*, vol. 26, no. 7, 2011.

[2] F. M. Ramos, "Mitigating IPTV zapping delay," *IEEE Communications Magazine*, vol. 51, no. 8, 2013.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–444, 11 2015.

[4] S. C. Jo, Y. G. Jin, Y. T. Yoon, and H. C. Kim, "Methods for integrating extraterrestrial radiation into neural network models for day-ahead pv generation forecasting," *Energies*, vol. 14, no. 9, 2021.

[5] I. Basicevic, D. Kukolj, S. Ocovaj, G. Cmiljanovic, and N. Fimic, "A fast channel change technique based on channel prediction," *IEEE Transactions on Consumer Electronics*, vol. 64, no. 4, 2018.

[6] A. C. Begen, N. Glazebrook, and W. Ver Steeg, "A unified approach for repairing packet loss and accelerating channel changes in multicast IPTV," in *2009 6th IEEE Consumer Communications and Networking Conference, CCNC 2009*, 2009.

[7] Agilent Technologies, "White Paper: Ensure IPTV Quality of Experience," 2005.

[8] N. Fimic, I. Basicevic, and N. Teslic, "Reducing Channel Change Time by System Architecture Changes in DVB-S/C/T Set Top Boxes," *IEEE Transactions on Consumer Electronics*, vol. 65, no. 3, 2019.

[9] R. Kooij, K. Ahmed, and K. Brunnström, "Perceived quality of channel zapping," in *Proceedings of the 5th IASTED International Conference on Communication Systems and Networks, CSN 2006*, 2006.

[10] B. Hu, G. B. Wu, L. Pan, H. Ni, and M. Zhu, "An implementation of interactive set-top-box and its applications," in *International Conference on Signal Processing Proceedings, ICSP*, vol. 4, 2006.

[11] Motorola, “Motorola DCX-3510M Data Sheet.” [Online]. Available: http://www.gditechnology.com/manuals/Motorola-DCX-3510M-Data-Sheet.pdf

[12] W. Chicaiza, D. Villamarín, and G. Olmedo, “Hybrid DTT & IPTV set top box implementation with GINGA middleware based on low cost platforms,” in *Communications in Computer and Information Science*, vol. 1004, 2019.

[13] T. T. Adeliyi, O. O. Olugbara, and S. Parbanath, “Minimizing Zapping Delay Using Adaptive Channel Switching with Personalized Electronic Program Guide,” *International Journal of Digital Multimedia Broadcasting*, vol. 2021, 2021.

[14] L. C. Costa, C. Hira, M. G. De Biase, F. A. Soares, W. Carvalho, and M. K. Zuffo, “Cost-effective hybrid Ginga-NCL interactive set-top box,” in *Proceedings of the International Symposium on Consumer Electronics, ISCE*, 2016.

[15] U. Jennehag, T. Zhang, and S. Pettersson, “Improving transmission efficiency in H.264 based IPTV systems,” *IEEE Transactions on Broadcasting*, vol. 53, no. 1, 2007.

[16] H. Joo, H. Song, D. B. Lee, and I. Lee, “An effective IPTV channel control algorithm considering channel zapping time and network utilization,” *IEEE Transactions on Broadcasting*, vol. 54, no. 2, 2008.

[17] C. Cho, I. Han, Y. Jun, and H. Lee, “Improvement of channel zapping time in IPTV services using the adjacent groups join-leave method,” in *6th International Conference on Advanced Communication Technology: Broadband Convergence Network Infrastructure*, vol. 2, 2004.

[18] Y. Kim, J. K. Park, H. J. Choi, S. Lee, H. Park, J. Kim, Z. Lee, and K. Ko, “Reducing IPTV channel zapping time based on viewer’s surfing behavior and preference,” in *IEEE International Symposium on Broadband Multimedia Systems and Broadcasting 2008, Broadband Multimedia Symposium 2008, BMSB*, 2008.

[19] C. Y. Lee, C. K. Hong, and K. Y. Lee, “Reducing channel zapping time in IPTV based on user’s channel selection behaviors,” *IEEE Transactions on Broadcasting*, vol. 56, no. 3, 2010.

[20] J. H. Ryu, B. Lee, K. T. Kim, and H. Y. Youn, “Reduction of IPTV channel zapping time by utilizing the key input latency,” in *2014 IEEE 11th Consumer Communications and Networking Conference, CCNC 2014*, 2014.

[21] Q. Tongqing, G. Zihui, L. Seungjoon, W. Jia, Z. Qi, and X. Jun, “Modeling channel popularity dynamics in a large IPTV system,” in *SIGMETRICS/Performance’09 - Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, vol. 37, no. 1, 2009.

[22] M. Cha, P. Rodriguez, J. Crowcroft, S. Moon, and X. Amatriain, “Watching television over an IP network,” in *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, 2008.

[23] U. Oh, S. Lim, and H. Bahn, “Channel reordering and prefetching schemes for efficient iptv channel navigation,” *IEEE Transactions on Consumer Electronics*, vol. 56, no. 2, 2010.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The WEKA data mining software,” *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, 2009.

[25] C. Yang, S. Ren, Y. Liu, H. Cao, Q. Yuan, and G. Han, “Personalized Channel Recommendation Deep Learning from a Switch Sequence,” *IEEE Access*, vol. 6, 2018.

[26] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 3rd ed., 1999, vol. 13, no. 4.

[27] K. Gurney, *An introduction to neural networks*, 1st ed., 1997.

[28] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, 1989.

[29] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943.

[30] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychological Review*, vol. 65, no. 6, 1958.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, 1986.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, vol. 2, 2012.

[33] Z. Berradi and M. Lazaar, “Integration of Principal Component Analysis and Recurrent Neural Network to Forecast the Stock Price of Casablanca Stock Exchange,” in *Procedia Computer Science*, vol. 148, 2019.

[34] M. Sheikh Fathollahi and F. Razzazi, “Music similarity measurement and recommendation system using convolutional neural networks,” *International Journal of Multimedia Information Retrieval*, vol. 10, no. 1, 2021.

[35] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, “DRAW: A recurrent neural network for image generation,” in *32nd International Conference on Machine Learning, ICML 2015*, vol. 2, 2015.

[36] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[38] A. Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks," *Web Page*, 2015.

[39] Q. Wang, Y. Ma, K. Zhao, and Y. Tian, "A Comprehensive Survey of Loss Functions in Machine Learning," *Annals of Data Science*, 2020.

[40] P. J. Werbos, "Backpropagation Through Time: What It Does and How to Do It," *Proceedings of the IEEE*, vol. 78, no. 10, 1990.

[41] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," 2019.

[42] J. F. Kolen and S. C. Kremer, "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies," in *A Field Guide to Dynamical Recurrent Networks*, 2010.

[43] J. Nagi, F. Ducatelle, G. A. Di Caro, D. Cireşan, U. Meier, A. Giusti, F. Nagi, J. Schmidhuber, and L. M. Gambardella, "Max-pooling convolutional neural networks for vision-based hand gesture recognition," in *2011 IEEE International Conference on Signal and Image Processing Applications, ICSIPA 2011*, 2011.

[44] M. Mozer, R. Lippmann, J. Moody, and D. Touretsky, "Induction of multiscale temporal structure," *Advances in Neural Information Processing Systems 4*, 1992.

[45] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, 11 1997.

[46] G. Van Houdt, C. Mosquera, and G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, no. 8, 2020.

[47] Y. Xie, R. Liang, Z. Liang, and L. Zhao, "Attention-based dense LSTM for speech emotion recognition," *IEICE Transactions on Information and Systems*, vol. E102D, no. 7, 2019.

[48] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM neural networks for language modeling," in *13th Annual Conference of the International Speech Communication Association 2012, INTERSPEECH 2012*, vol. 1, 2012.

[49] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, 2014.

[50] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,” 12 2014.

[51] T. Donkers, B. Loepp, and J. Ziegler, “Sequential user-based recurrent neural network recommendations,” in *RecSys 2017 - Proceedings of the 11th ACM Conference on Recommender Systems*, 2017.

[52] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: A system for large-scale machine learning,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, 2016.

[53] R. Elshawi, A. Wahab, A. Barnawi, and S. Sakr, “DLBench: a comprehensive experimental evaluation of deep learning frameworks,” *Cluster Computing*, vol. 24, no. 3, 2021.

[54] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[55] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” in *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia*, 2014.

[56] T. Chen, M. Li, U. W. Cmu, Y. Li, M. Lin, N. Wang, M. Wang, B. Xu, C. Zhang, Z. Zhang, and U. Alberta, “MXNet : A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems arXiv : 1512 . 01274v1 [cs . DC] 3 Dec 2015,” *Emerald Group Publishing Limited*, vol. 36, no. 2, 2015.

[57] Taiga Nomi, “tinny-dnn.” [Online]. Available: https://github.com/tiny-dnn/tiny-dnn

[58] H. Li, “kann: A lightweight C library for artificial neural networks.” [Online]. Available: https://github.com/attractivechaos/kann

[59] ——, “Identifying centromeric satellites with DNA-brnn,” *Bioinformatics*, vol. 35, no. 21, 2019.

[60] J. Sola and J. Sevilla, “Importance of input data normalization for the application of neural networks to complex industrial problems,” *IEEE Transactions on Nuclear Science*, vol. 44, no. 3 PART 3, 1997.

[61] G. Petneházi, "Recurrent Neural Networks for Time Series Forecasting," 2019. [Online]. Available: https://arxiv.org/abs/1901.00069

[62] M. Adil, R. Ullah, S. Noor, and N. Gohar, "Effect of number of neurons and layers in an artificial neural network for generalized concrete mix design," *Neural Computing and Applications*, 2020.

[63] D. A. Roberts, S. Yaida, and B. Hanin, "The Principles of Deep Learning Theory," 6 2021. [Online]. Available: https://arxiv.org/abs/2106.10165

[64] O. M. Yamashita, J. R. Betoni, S. C. Guimarães, and M. M. Espinosa, *Deep Learning (Adaptive Computation and Machine Learning series)*, 2009, no. 84.

[65] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, 2014.

Code of Project

Listing A.1: Code used to train the RNN

```
1  Epoch* train(kann_t ann, Vector *data, int ulen, float max_train_rmse,
2          float learningRate, int verbose, MappingList ml,
3          LogResultsStruct* logStruct){
4      if (logStruct != NULL){
5          fprintf(logStruct->errorFile, "epoch, cost, classError\n");
6      }
7      float *r; kann_t *ua; int n_var, n_in, n_out, i, j;
8      n_in = kann_dim_in(ann); // number of inputs
9      n_out = kann_dim_out(ann); // nmber of outputs
10     if (n_in < 0 || n_out < 0) return NULL;
11     n_var = kann_size_var(ann);                // total size of variables
12     r = (float*)calloc(n_var, sizeof(float)); // temporary array for RMSprop
13     Epoch *epochStruct = initializeEpoch();
```

```
14      float *x = (float*)malloc(sizeof(float*)*ulen);
15      for(i = 0; i < ulen; i++) x[i] = (float*)calloc(n_in,sizeof(float));
16      float *y = (float*)malloc(1 * sizeof(float*));
17      y[0] = (float*)calloc(n_out,sizeof(float));
18      ua = kann_unroll(ann, ulen); // unroll; the mini batch size is 1
19      kann_set_batch_size(ua, 1);
20      kann_feed_bind(ua, KANN_F_IN,    0, x); // bind x to input nodes
21      kann_feed_bind(ua, KANN_F_TRUTH, 0, y); // bind y to truth nodes
22      kann_switch(ua, 1); float time_spent = 0;
23      for (epochStruct->lastEpoch = 1; epochStruct->lastEpoch <
24      max_epochs; ++epochStruct->lastEpoch) {
25          clock_t epoch_start = clock(); double cost = 0.0; int tot = 0;
26          tot_base = 0; n_cerr = 0;
27          for (j = 0; j < data->nlines - ulen; j += 1) {
28              int k;
29              for (int index = 0; index < ulen; index += 1) {
30                  oneHotXLine(x[index], data, index+j, n_out, ml);
31              }
32              oneHotYLine(y[0], data, j+ulen-1, n_out, ml);
33              cost += kann_cost(ua, 0, 1)*ulen; n_cerr += kann_class_error(ua, &k);
34              tot_base += k; kann_RMSprop(n_var,learningRate,0,0.9f,ua->g,ua->x,r);
35              tot += ulen;
36          }
37          clock_t epoch_end = clock();
38          time_spent += (double)(epoch_end-epoch_start)/CLOCKS_PER_SEC;
39          if (verbose == 1) {
40              fprintf(stderr,"epoch: %d; cost: %g (class error: %.2f%%)\n",
41                      epochStruct->lastEpoch, cost/tot, 100.0f * n_cerr/tot_base);
42          }
43          if (logStruct != NULL) fprintf(logStruct->errorFile, "%d, %g, %.2f\n",
44                      epochStruct->lastEpoch, cost/tot, 100.0f * n_cerr/tot_base);
45          if (cost/tot < max_train_rmse) break;
46          if(isTimeExceeded(1.5, time_spent)) break;
47      }
48      for(i = 0; i < ulen; i++) {free(x[i]);}
49      kann_delete_unrolled(ua); free(x); free(y[0]); free(y); free(r);
50      return epochStruct;
51  }
```

B

User's prediction graphics.

B.1 Artificial User

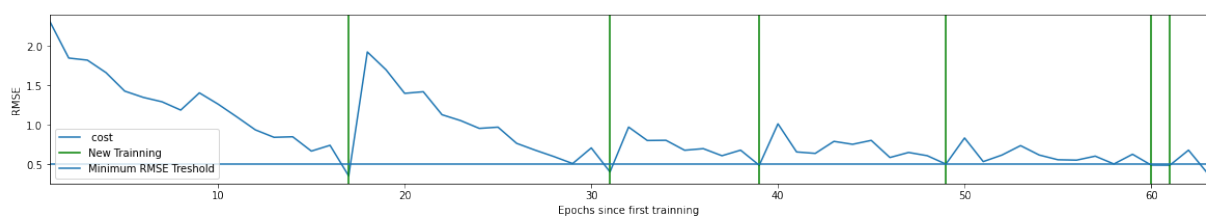


Figure B.1: Error loss plot during training for the artificial user.

Figure B.2: Channel changes performed by the artificial user on the first week.

Figure B.3: Channel changes by the artificial user and model prediction for the second week.

Figure B.4: Channel changes by the artificial user and model prediction for the third week.

Figure B.5: Channel changes by the artificial user and model prediction for the fourth week.

Figure B.6: Channel changes by the artificial user and model prediction for the fifth week.

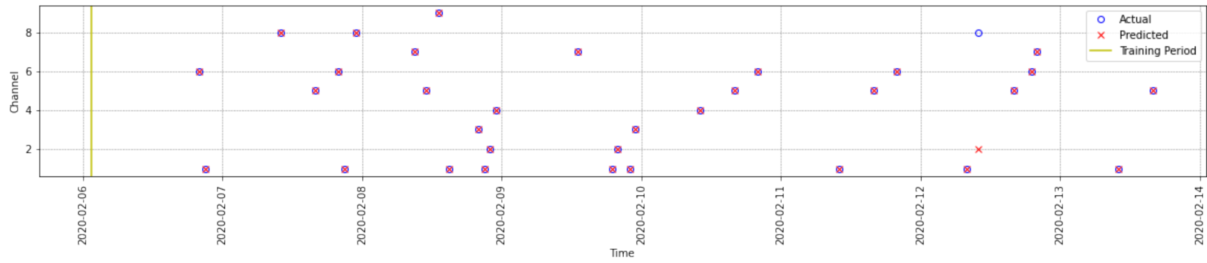


Figure B.7: Channel changes by the artificial user and model prediction for the sixth week.

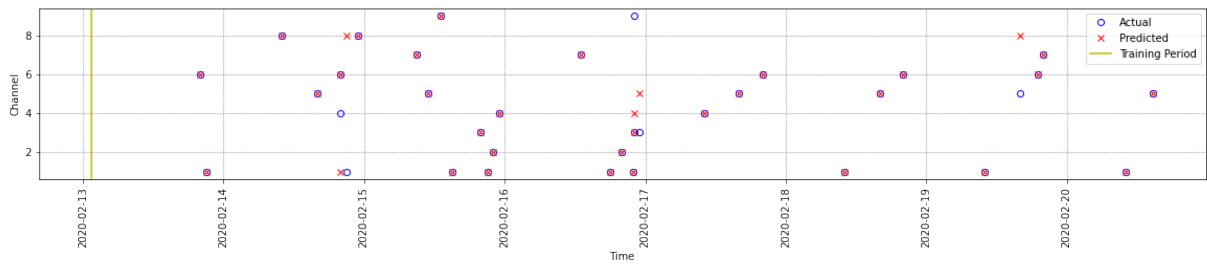


Figure B.8: Channel changes by the artificial user and model prediction for the seventh week.

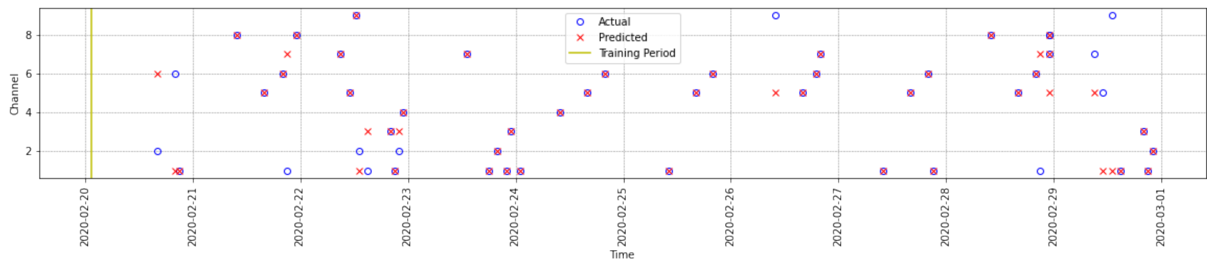


Figure B.9: Channel changes by the artificial user and model prediction for the eighth week.

B.2 Best User

Figure B.10: Error loss plot during training for the best user.

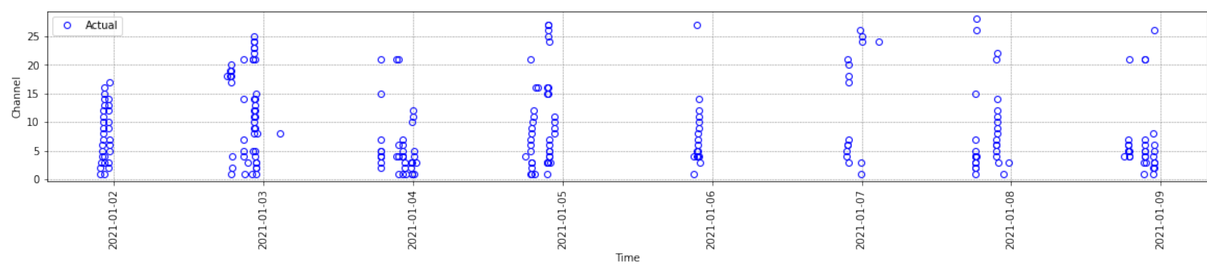


Figure B.11: Channel changes performed by the best user on the first week.

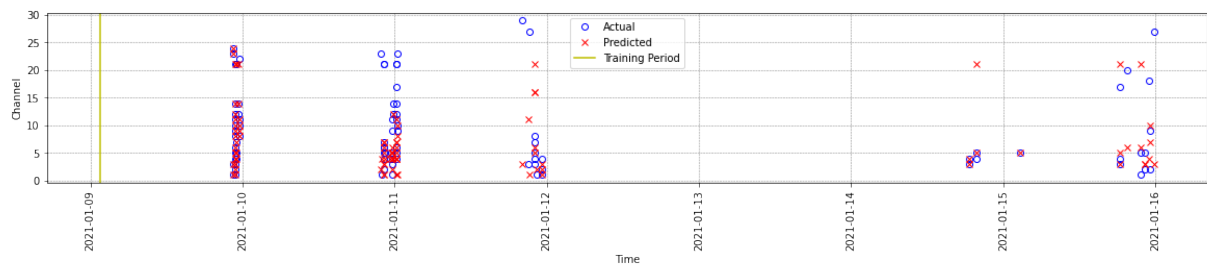


Figure B.12: Channel changes by the best user and model prediction for the second week.

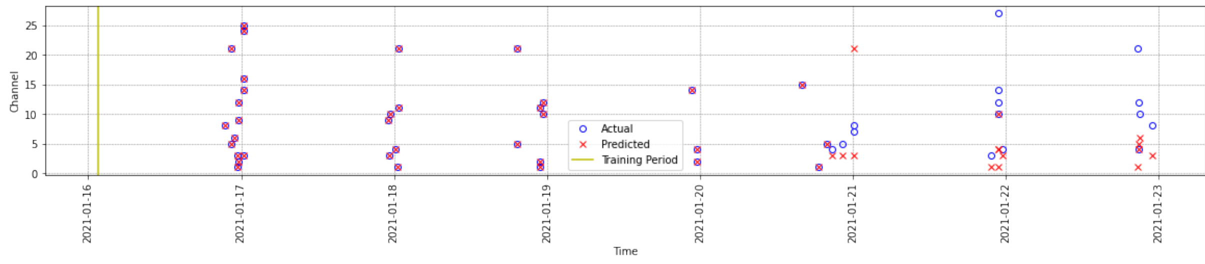


Figure B.13: Channel changes by the best user and model prediction for the third week.

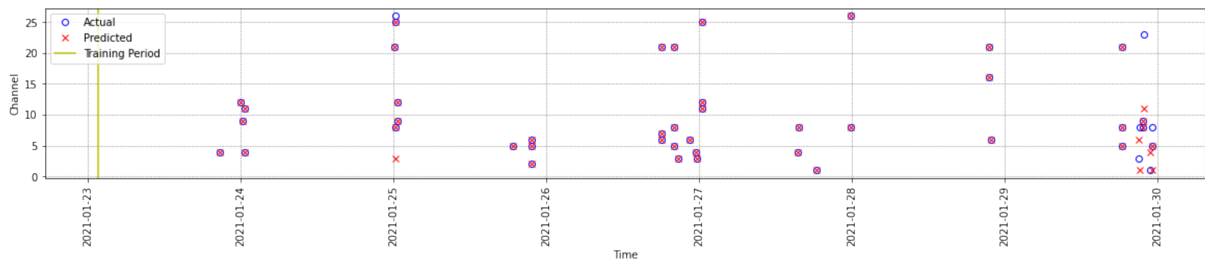


Figure B.14: Channel changes by the best user and model prediction for the fourth week.

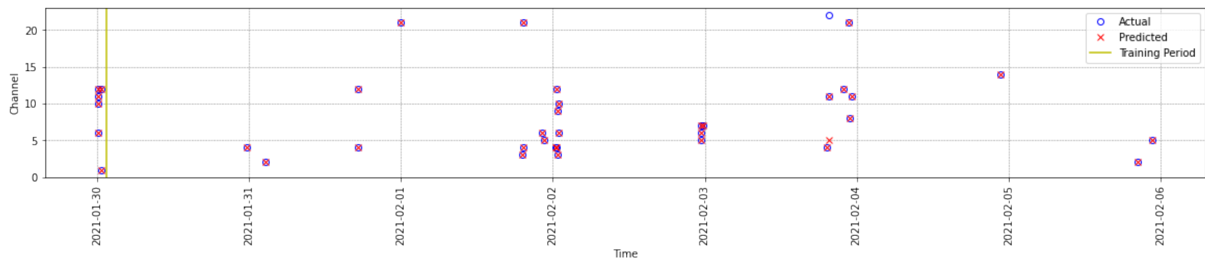


Figure B.15: Channel changes by the best user and model prediction for the fifth week.

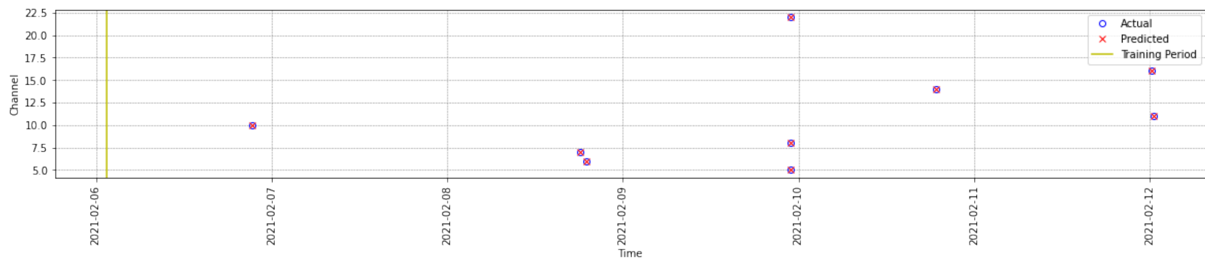


Figure B.16: Channel changes by the best user and model prediction for the sixth week.

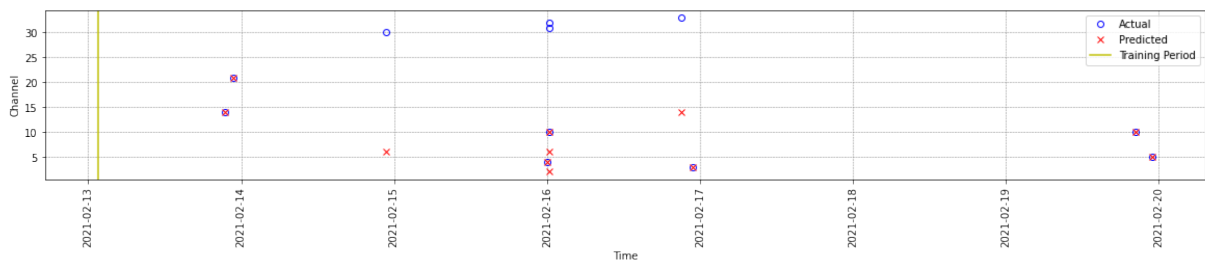


Figure B.17: Channel changes by the best user and model prediction for the seventh week.

B.3 Worst User

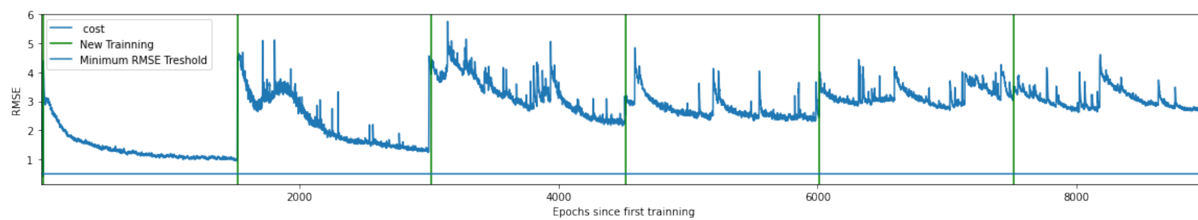


Figure B.18: Error loss plot during training for the worst user.

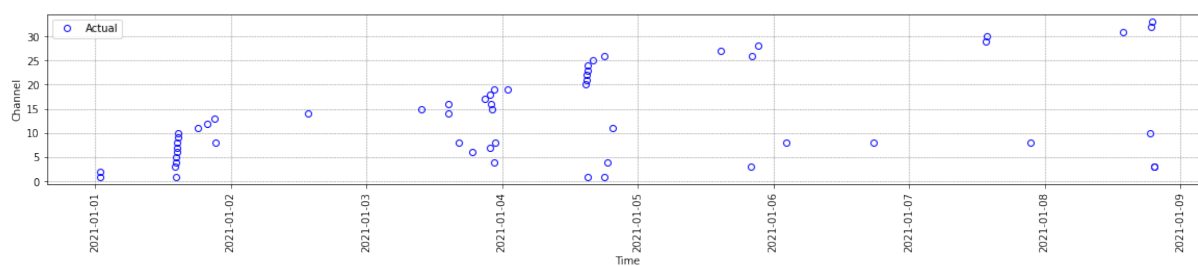


Figure B.19: Channel changes performed by the worst user on the first week.

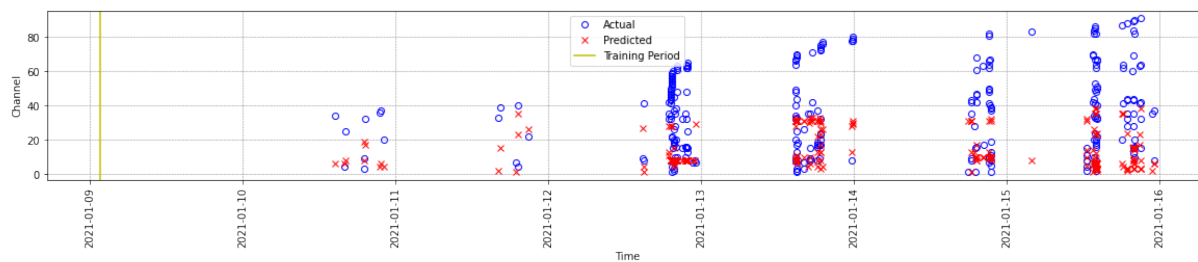


Figure B.20: Channel changes by the worst user and model prediction for the second week.

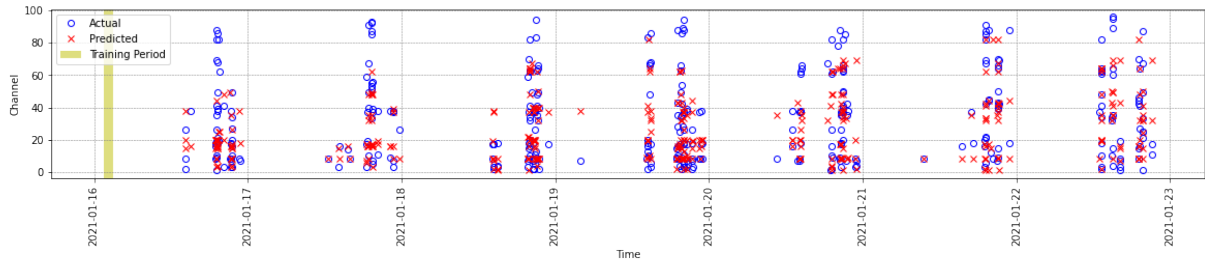


Figure B.21: Channel changes by the worst user and model prediction for the third week.

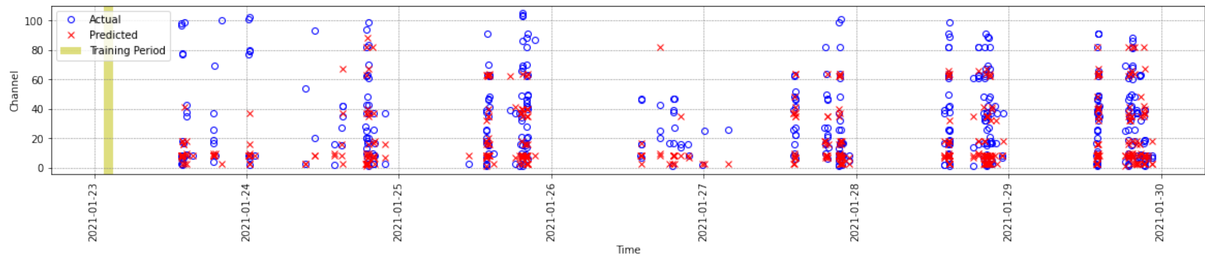


Figure B.22: Channel changes by the worst user and model prediction for the fourth week.

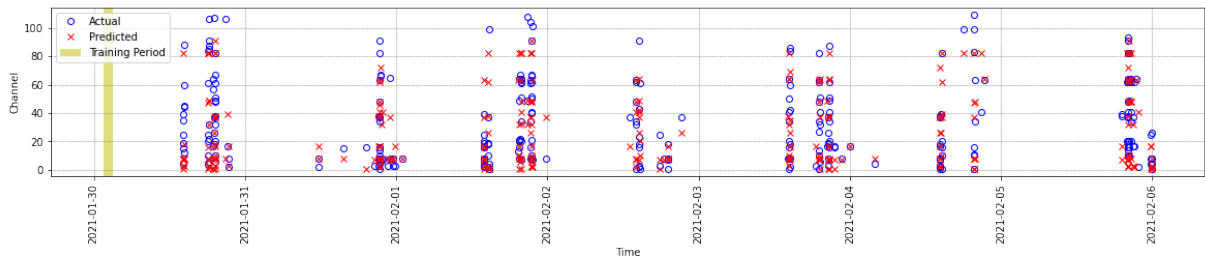


Figure B.23: Channel changes by the worst user and model prediction for the fifth week.

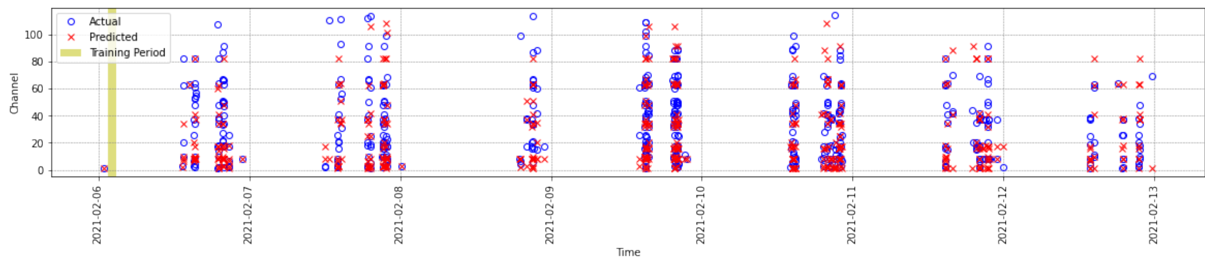


Figure B.24: Channel changes by the worst user and model prediction for the sixth week.

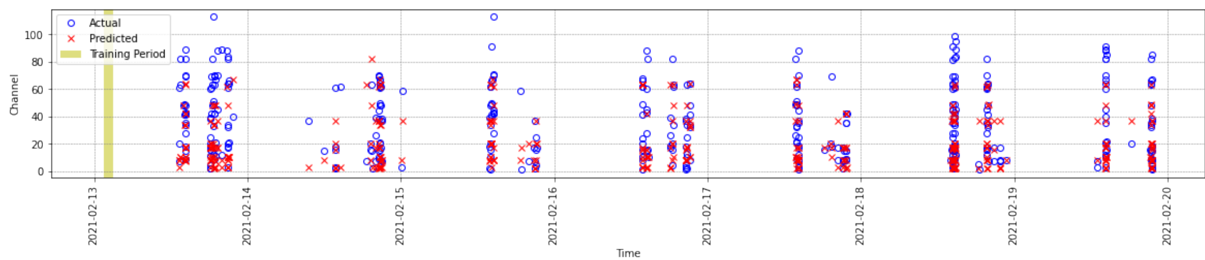


Figure B.25: Channel changes by the worst user and model prediction for the seventh week.

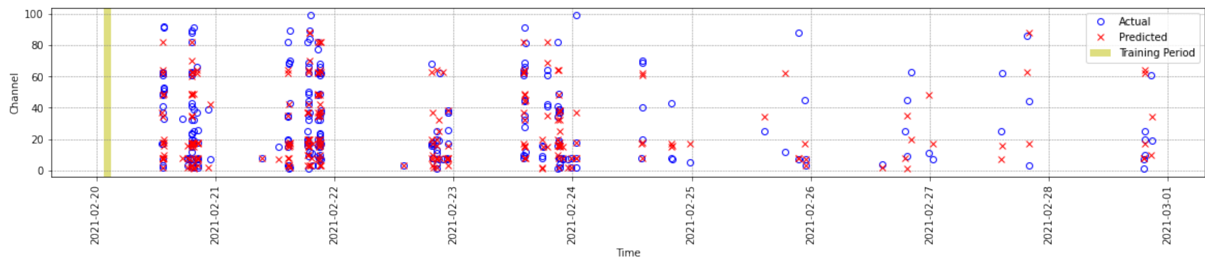


Figure B.26: Channel changes by the worst user and model prediction for the eighth week.