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

Tomás Santos Azenhas Boavida Malcata

Instituto Superior Técnico, Lisboa, Portugal

November 2022

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

1. Introduction

When the TV started to become commercialised and used around the globe, it was primarily used analogue TV. In analogue TV, all the channels are provided in 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 wanted to change channels, the TV is required to tune the correspondent frequencies. Using this technology, the delay time for changing a channel was around 200 ms [2]. 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 advantages of providing superior quality channels, having easy integration with other Internet Protocol (IP) based services and a Personal Video Recorder (PVR). However, with the implementation of such functionalities and the change of content delivery method, the zapping time has increased substantially from 200 ms [2] to a delay between 0.9 s and 70 s in some IPTV configurations [3] or 2.6 s in DVB [4]. Yet, the acceptable Quality Of Experience (QoE) is 430 ms [5]. Note that the zapping time is measured from the moment the user sends a new channel request to the time the user starts watching it on the TV.

This operation can be divided into five different moments: the channel change request (between the remote and the STB), the distribution network access delay (communication with the server to stop receiving the multicast traffic from a channel and start receiving the new one), the synchronisation delay (time until receives an I-frame to start decoding, buffer delay (time to fill the buffers) and last the STB processing delay (time for the STB to be able to stream the new channel) [6]. On a DVB architecture there is also the tuning delay. However the most relevant phases of the process, the synchronisation and buffer delay are shared. The remaining zapping phases can be considered as virtually zero [1]. The zapping delay takes on average 2 s [1]. For reducing this delay, the STB can request the new channels previously and if the user changes to an already requested channel the zapping delay will be virtually 0 s [1].

1.1. 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 fol-

lowing: 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 RAM and disk space) available in STBs are known to be quite modest.

In order to simulate a STB it is commonly used a Raspberry Pi [7] [8] [9]. 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. It was also used during these performance tests: a Raspberry Pi 4 model B with a quad core Cortex-A72 CPU operating at 1.5 GHz, a Raspberry Pi 2 model B with a quad core Cortex-A7 CPU operating at 900 MHz and an ARM processor with a cortex-A52 dual core embedded in a FPGA Zynq.

2. State of the Art

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 [10]. 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.

This research work considers the predictive method to reduce the zapping delay in digital TV since it 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. In fact, this approach allows reducing the delay to below 0.43 s which is not perceptible by the users [3]. 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.

2.1. Adjacent group join-leave method

The first predictive model for shortening the zapping delay was proposed in 2004 [11]. 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 [11], it was later evaluated using data from 25500 users for 150 different channels [6]. 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 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 thesis, described in Section 4, the up and down model plus the previous channel was performed, and its results can be seen in Table 3, where the best combination for the amount of predicted channels is chosen. It is visible that the results were not observed in [1] [6].

However, the dataset is different, and it is possible to compare the results on the same dataset with the results that this project will present.

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.

2.2. 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 [12] and [13], 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 [14] to create a predictive channel system based on recording the TV channels by their popularity.

Even though there were promising results in [12] [14], it was later performed 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 [6], 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.

2.3. Machine learning approach

A more complex approach for the predictive problem was proposed in [10]. This research work suggests that a purely ML 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 [12]. It was created, trained and analysed a set of ML models. The best accuracy for this model is presented in Table 3.

The ML models experimented with were mostly tree-based models, and the best accuracy result was achieved using a random forest. To create it, the channel change log is saved once a channel change is requested. These logs are used for training the model. The model is rebuilt 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 [15] was used externally to the STB computer. Also, it used synthetic data based on a previous TV users study [12] for training and predicting with the model. It achieved an accuracy result of 37.39 % for the model with the best performance.

On a different note, a Neural Network (NN) based model was created on [16] 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 [16], 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 on 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. Implementation

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. The model should be deployed inside a STB and use its resources to predict and train. 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 is not known by the NN model and, in most cases, should be dependent on 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. 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.

A NN based model was chosen since, after the training, the inference about new values is very fast. Furthermore, since this model will be used on actual data collected in real-time, an NN provides high tolerance to failure even if some input is incomplete. The capability of an NN changing its behaviour was also taken into consideration. On the predicting channel changes problem, it is widespread that suddenly a user changes their favourite channels or starts having different schedules. An NN can adapt its weights to this change while training without having to be recreated. In both cases, the model will then infer which channel should be requested next, using the list of 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 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. An 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.

The framework that was chosen to implement the proposed RNN model was the Kann library [17]. This decision was the result of a careful analysis of the pros and cons of several frameworks. 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 [17] such as the one devised in this work.

3.1. Data used

To create the model, the only information used was relative to the channel change logs stored in each user STB, which only includes the registry of the new channel and the time when the change occurred. Therefore, the model does not require information to be shared among different users.

For keeping the channel change history, it is just needed the time when the channel change was requested and the channel requested. This minimal approach of data storing can then be enriched with more information such as the previous channel, day of the week and day of the year and 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.

To use this data in the model from each log it is created a vector in the format: $[DayOfTheYear, TimeOfTheDay, WeekDay, CurrentChannel]$ for the inputs and the correspondent output $[NextChannel]$ where $DayOfTheYear$ is the number of the day divided by the total days in a year, the $TimeOfTheDay$ corresponds to the hour and this value was normalised using a sinusoidal function with a period correspondent to 24 hours in order to normalise it and to avoid discontinuities between 23h59 and 00h00, the $WeekDay$ is a one-hot encoded array with the value of one for the correspondent day and zero for the others. At last, the channels are also one-hot encoded, since each channel is a class, the $CurrentChannel$ is the channel where the user was and the $NextChannel$ is the channel the user changed to. Since there will be several inputs a vector of inputs is created with the respective output to feed on the RNN for training and the same without the output for the inference.

Finally, the output for the training will be the $NextChannel$, although, during inference, the output is an ordered list of channels by the model probability for each channel, since the STB may have resources to receive more than one extra channel.

3.2. 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 1.1. 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.

3.3. Channel mapping

Nowadays, the total of available channels in a TV provider is very large and can sum up to more than a thousand. For the creation of this model, the channels are treated as classes and the model output should be one class corresponding to a channel. Hence, there might be the problem of 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, using the available dataset, most of the users do not use five percent of the available channels (around 50 channels). Another drawback of such 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. 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 [16].

The main drawback of this implementation is that the model will never predict channels that are not mapped in the reduced dimension, 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. This mapping will have a maximum size.

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.

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.

3.4. 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. 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.

A successful evaluation of the model requires not only that the model provides have good accuracy but also that it complies with the constraints defined in Section 1.1 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 parameter 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. Dataset and hyperparameters

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 2.

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), 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).

4.1. Hyperparameters

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 contain 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. The remaining 270 users are used to validate the model generalisation properties.

For creating this subset, it was created 100 different RNN models by randomly varying the hyperparameters within their typical values. All the users would be assigned a randomly chosen configuration (from the 100 combinations), and the five users that would perform worst would be dropped from the subset for analysing the performance of a user with a specific RNN architecture for this algorithm it was only taken into consideration the accuracy. This process is repeated until only 30 users are left. This 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 1.

For finding the model's best hyperparameters, using the smaller dataset, it was necessary to create an initial guess for each hyperparameter. After, a range for each hyperparameter was defined within the standard values. The model was executed several times by having these initial hyperparameters fixed and only one (the hyperparameter in study) varying. After, the hyperparameter would be replaced with the value that provided a better result. This process was repeated for each hyperparameter several times until it reached a stable combination, presented in Table 1.

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 for evaluating the results it was 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 that the generated RNN has which influences the memory needed to allocate the model, the average accuracy, the number of users that performed better and the training time training samples.

For the full functioning of the model in terms

Table 1: 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

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 Root Mean Square Error (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 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, and it would be possible to be ignoring data and have a model converging faster, on the other hand, it is better to train with a more extensive dataset than with a smaller 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 [13]). 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 2, not many users have more than this number of channels, and therefore the model is rebuilt fewer times, which is also supported by [16]. 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. Results

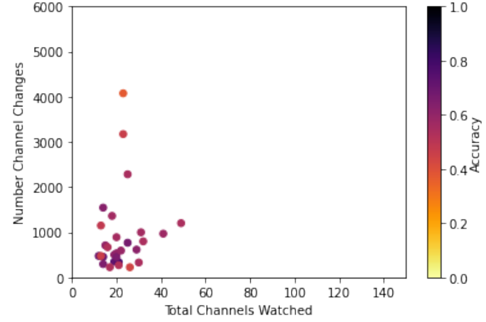

Figure 1: Heat map regarding the accuracy for one channel, number of channels watched and number of channel changes for the best 30 users group.

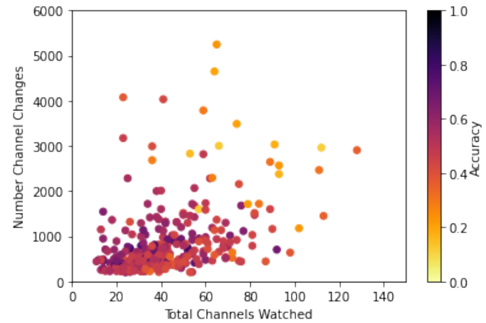

Figure 2: Heat map regarding the accuracy for one channel, number of channels watched and number of channel changes for all users.

To evaluate the model, it is crucial to assert how a more significant number of users perform. The accuracy results are shown in Table 3, 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 2 and for the best users group in Figure 1. 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 costs of receiving multiple channels for the STB. The best 30 users group changes from 57.3% to 83.8 %, and the costs change accordingly.

For the 270 users, the accuracy obtained was 49.3 % with only one channel and it increases to 65.7 % for five channels. 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. These values show that the model can generalise for users whom the RNN were not specifically designed for.

By comparing Figure 1 and Figure 2, 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 2 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 that a particular would experience with this implementation using the accuracy and the dataset size.

Because the average accuracy for one channel fetched is 50.2 %, 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 67.7 %. 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.

5.1. Validation on embedded systems

Apart from the results accuracy results, 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 peak of RAM used is 2.4 MB. This value represents the memory needed to allocate the RNN and the training dataset for the training phase. The peak RAM value is low enough for a STB. This value is independent of the number of channels the model predicts since it 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 the embedded devices with the definitions defined on Section 1.1. The last device tested was a personal computer with an i7-5500U CPU operating at 2.4 GHz with a limitation of 50 % on the CPU usage.

However, it was only possible to test for two 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 an i7-5500U. However, due to processor limitations, the training time will increase. On Table 2 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 2 that the times are slower on ARM Cortex-A53. 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 it provided performance results in the same order. This embedded system has a higher computing power than the rest, so it is expected to be faster. Regarding the ARM Cortex-A9, it provided an average of 2758 s for all the users, so, 45 minutes which is a very acceptable value. Finally, the ARM Cortex-A7 was the slowest, as expected, but it still executed within 3888 s for all

users (1h04) and 1614 s (16 minutes) to the best 30 group.

Table 2: Training time in the different devices.

	Time (s)	
Dataset	**All Users**	**Best 30**
i7-5500U	441	238
ARM Cortex-A72	627	310
ARM Cortex-A53	1465	506
ARM Cortex-A9	2758	546
ARM Cortex-A7	3888	1614

5.2. Comparison with other models

On Table 3, it is possible to see how the model performs when comparing the state-of-the-art models presented. It is important to notice that for the models presented in [10] and [16], the accuracy was calculated in a different dataset (the one present in each respective research) therefore, it can not be directly compared. For the up and down model, the presented results only consider 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 3: Accuracy results comparison between the model proposed and the ones defined in the state of art by number of channels.

#	**Proposal (%)**	**Adjacent [11] (%)**	**Tree [10] (%)**	**SL [16] (%)**
1	50.2	12.2	37.39	22.23
2	56.9	23.5	NA	NA
3	61.4	31.2	NA	47.98
4	64.8	NA	NA	NA
5	67.7	NA	NA	63.55

It is noticed that the usage of an 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 [16] is higher for one channel, although it is very similar for five channels. This is primarily due to the similarity of these two models. 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.

6. Conclusion

This thesis presented a predictive model based on a RNN suitable to be implemented in STB to predict the channel a user is most likely 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 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 following channel.

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 model was built using the Kann library [17], 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 three embedded systems, which their hardware configurations resemble a STB.

To test the model, different embedded systems were used (with hardware configurations similar to a STB) 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 [10] in 12.81 %. When compared to the LSTM model not implemented in a STB [16] the results provided by the proposed model are also better: 27.97 % for one channel and 4.15 % for five channels.

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. 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 stop 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.

References

[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] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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