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

Tomás Malcata[1,2,3][0000−0002−2296−9803], Nuno Sebastião[4,5][0000−0001−9537−0554], Tiago Dias[1,4,5][0000−0001−7445−5823], and Nuno Roma[1,2,3][0000−0003−2491−4977]

[1] INESC-ID
[2] Instituto Superior Técnico
[3] Universidade de Lisboa
[4] Instituto Superior de Engenharia de Lisboa
[5] Instituto Politécnico de Lisboa
tomas.malcata@tecnico.ulisboa.pt, nuno.sebastiao@isel.pt, tiago.dias@isel.pt, nuno.roma@inesc-id.pt

Abstract. With the generalization of digital Television (TV), keeping the channel change delay as low as possible gradually became a difficult requisite in what concerns the resulting user's Quality-of-Experience (QoE). Frequently, this latency may be higher than 2 seconds. While many state-of-the-art Set-top-Boxes (STBs) already include a shadow tuner to anticipate the tuning of the next channel, they strive to predict which channel should be pre-tuned, generally opting for one of the adjacent channels. The presented research proposes the use of a predictive system to assist the STB in the forecast of the channel(s) the user will select next. The implemented predictor is based on a Recurrent Neural Network (RNN) and makes use of STB log data concerning the user's channel changes history to train (and adjust) the model every week. To attain this objective, the most convenient hyperparameter combination that not only fulfilled the aimed prediction accuracy but also suited the rather limited computational constraints of most current STBs had to be identified. The obtained experimental results, validated using four embedded processor families commonly equipping commercial STBs, showed a prediction accuracy of 50.2% for a single-channel prediction and 67.7% when five channels were simultaneously predicted. When combined with the existing dual-tuning system of current STBs, the proposed predictor can save as much as 1000 seconds per month in TV channel change delays, greatly improving the resulting user's QoE.

Keywords: Channel Change Delay · Set-Top-Box · Predictive System · Recurrent Neural Network.

1 Introduction

Digital Video Broadcasting (DVB) systems, either DVB-C, DVB-T or DVB-IPTV, typically suffer from a high channel change (zapping) time. This results from the inherent limitations of transmitting the data at a constant bitrate, on a broadcast method (in which there is no ability to request additional data) and having only specific points in time where decoding can start (due to the video compression techniques). Such limitations make the total zapping time range from a couple of seconds to tenths of seconds, which significantly reduces the user's Quality-of-Experience (QoE) when zapping through the channels.

Operators that control their delivery networks, typically those that have DVB-C and DVB-IPTV networks, commonly rely on custom built specialized hardware devices installed in the customers premises, denoted as Set-top-Boxes (STBs), to deliver their services. Operators may decide to include in these devices additional features that allow to reduce the channel change time. Nevertheless, due to cost reasons, such embedded devices are commonly restricted both in terms of these additional features as well as in their computational capabilities.

Various methods have been devised to reduce the channel change time [3, 5, 8]. However, all of them have an additional cost to the operator, either because it requires additional hardware resources or additional bandwidth in the network. Since these resources are limited, the best QoE is achieved when the user actually changes to the channel(s) for which the system was previously optimized.

Using prediction systems based on Neural Networks (NNs) [6] has now become more feasible, which allows to more accurately predict the channels that the user will watch next. With this additional information it becomes possible to dynamically configure the STB in order to achieve the previous goal (i.e., provide the smallest possible delay to change to the desired channel), by configuring the local system resources to preemptively get the data needed to start playback of a new channel. Furthermore, it is preferable to have the prediction system using only the already available computational resources of the STB, as opposed to having a centralized solution, due to the smaller cost. However, this means that the computational load of the predictor should be light enough so that it is compatible with the limited computational resources that exist in the STB.

This paper presents a method that predicts the channel change behavior of a given user based on a Recurrent Neural Network (RNN) that suggests a list of channels that the user will most probably watch at any given moment. Such RNN can be trained and executed entirely in the computationally restricted environment of a STB. In Section 2, an overview of the related work is provided while Section 3 presents the proposed solution to predict the user's channel change behavior. Section 4 presents the assessment of the proposed model both in terms of its prediction accuracy and of the computational requirements on various typical embedded platforms, similar to those used in STBs. Finally, some conclusions are drawn in Section 5

2 Related Work

The channel change time is the result of the sum of various delays that can be divided into: i) distribution network access delay, i.e., the carrier frequency tuning time, for DVB-C or DVB-T, or the typical multicast group join time of DVB-IPTV; ii) synchronization delay, i.e., the time it takes until a random access point in the transport stream is received; iii) video buffering delay, i.e., the time it takes to fill the buffers with the data broadcasted at a constant bitrate to a level at which playback can start; and iv) device processing delays, i.e., key press event handling and internal processing - typically quite small when compared to the other delays. The sum of these delays may vary from a couple of seconds to tenths of seconds, with the synchronization and buffering delays being the largest contributors.

Some STBs possess dedicated hardware to assist in reducing the channel change time, e.g., by having the ability to simultaneously capture multiple streams, demultiplex various channels and even the ability to decode more than one video stream. The channel change time when using all of these capabilities, including simultaneous video decoding, allows the channel change time to be as low as 20 ms. However, since these capabilities come at a cost, the actual number of channels that can be simultaneously processed is quite limited.

Besides the use of additional dedicated hardware, it is also possible to reduce the channel change time by having additional companion streams or an additional medium to fetch the required data to fill in the video buffers in a faster way. This additional medium is most commonly a network connection, which is nowadays a common feature even on the STBs of the DVB-C operators.

Regardless of the technique used to reduce the channel change time, the amount of resources is always limited and thus requires the system to select which channel (or channels, depending on the available resources) it should prepare to change to. Ideally, the system would always be prepared to change to the channel that the user will watch next. Furthermore, when looking at solutions that require additional bandwidth to support a faster channel change (e.g., additional dedicated streams or a network connection), it is important to note that these also impose an increased cost to the operator and should be minimized.

Furthermore, whether the distribution network is DVB-C, DVB-T or DVB-IPTV, the use of prediction systems to determine the next watched channel significantly helps in making a better use of the available resources to reduce the channel change time [1, 5, 8, 12], either by configuring the local system resources (e.g., demuxer or video decoder) [4] or by preemptively getting the data from the network to provide the streams with optimal delay [3].

3 Proposed Solution

3.1 RNN Model

NNs [6, 9] present several advantages also for the development of channel change prediction systems. Firstly, the inference procedure in NNs is very fast, which

is quite important to minimize the delay when choosing the next channel to be displayed in a channel change event. Secondly, NNs provide high tolerance to failure even if some input data is incomplete. This feature is quite relevant for STB-oriented prediction systems, since the input data (i.e., the users' channel changes history) is stored in logs that might not always contain all the required data due to unexpected events, such as log corruption or lack of disk space. The capability of a NN to adapt its behavior dynamically is another important aspect, since it is a crucial feature when dealing with data collected in real-time. For the channel change prediction problem, it is well known that users quite often change not only their favorite channels list but also their viewing schedules. A NN model can easily adapt to such changes simply by retraining the network, i.e., adjusting the neurons weights. This makes it possible to retrain the model periodically with some new data (e.g., weekly), without having to discard the existing model and recreate a new one from scratch. Finally, RNNs allow having the model output depending on sequential data, which is quite advantageous to significantly improve the accuracy of the prediction in time series problems [9], like the channel switching sequences of a user.

For the aforementioned reasons, the channel change prediction method herein proposed is based on a RNN model with a many-to-one configuration, where the channel prediction model takes into consideration not only the current date, time and Television (TV) channel the user is watching for channel change events but also a set of channels previously displayed to significantly improve the accuracy of the prediction. Although other existing models consider additional user data [2, 5, 8, 10], such as the user preferred TV genres, favourite programs, surfing behaviour, demographic information, etc., we choose to adopt a simpler model tailored for implementations on STBs, due to the tight computational performance, Random Access Memory (RAM) and persistent storage constraints of these platforms.

The architecture of the proposed RNN model is shown in Fig. 1, where 'Day of the year', 'Time of the Day' and 'Week Day i' correspond to the instant the channel change event occurred, 'Previous channel j' are the most recent j channels watched by the user at such instant, and 'Output Channel k' is an ordered list of k channels that the user is most likely to switch into, arranged from the highest to the lowest probability. To reduce the propagation of errors, the 'Day of the year' and 'Time of the Day' values are normalized to year days and seconds, respectively. Conversely, the 'Week Day i', 'Previous channel j', and 'Output Channel k' values are treated as classes and one-hot encoded. Consequently, the model has $9+C$ input nodes and C output nodes, where C is the number of recent channels watched by the user that are considered for the prediction. The number of nodes in each hidden layer is also C, as shown in Fig. 1. The proposed RNN model considers a maximum of 50 channels ($C = 50$), since recent studies show that most users do not watch more than 50 different channels [12].

The amount of hidden layers, the number of nodes in each hidden layer, the type of neurons, and the unroll length (i.e., the length of the input sequence containing the history of the last channels watched by the user) are the model

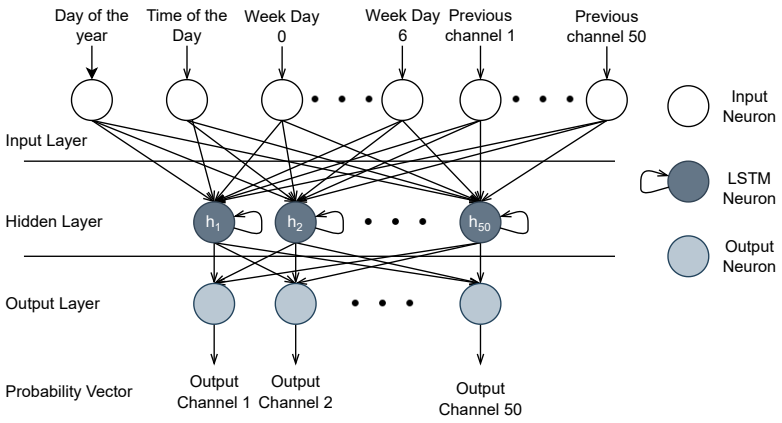

Fig. 1. Architecture of the proposed RNN.

hyperparameters influencing the RNN architecture, as discussed in Section 4.2. The remaining model hyperparameters are the learning rate, the dropout rate, the number of weeks and epochs used to train the network.

3.2 Model Implementation

The proposed RNN model was implemented using the KANN framework [7], which is an open-source standalone and lightweight deep learning library developed using the C programming language. When compared to other state-of-the-art frameworks, KANN is as efficient as most of them for small NN, like the RNN model herein proposed. Nonetheless, KANN presents important advantages for implementations on constrained embedded platforms, such as STBs. In particular, the computational efficiency of this framework is very high, due to being implemented using a low-level programming language. For the same reason, its RAM and persistent storage requirements are quite modest. For example, the framework contains only four files that occupy 132 KB in the filesystem of an embedded platform. Furthermore, KANN has no software dependencies (except for the C standard library), which makes it highly portable and suitable for implementations in most Unix-based computational systems, such as STBs.

Using the KANN framework, the proposed RNN model was implemented with a specially developed C function based on the following API functions: `kann_layer_inputs()`, to receive the model inputs; `kann_layer_rnn()`, `kann_layer_lstm()`, and `kann_layer_gru()`, to implement the hidden layer(s); `kann_layer_dropout()`, to perform the dropout regularization; and `kann_layer_cost()`, to select the output(s). The inputs to such functions are the RNN model hyperparameters, i.e., the number of inputs, outputs, and hidden layers, the amount of neurons per hidden layer, the neurons' types, and the dropout rate.

To train the proposed RNN model another custom C function was developed, since the KANN API only provides a training function for simple Feedfor-

ward Neural Networks (FNNs). Such training procedure encompasses two parts: *setup* and *action*. In the *setup* part, all the information necessary to conduct the training procedure is obtained, i.e., the amount of inputs, outputs, and trainable variables of the model. Then, a new NN model is created by unrolling the initial RNN model through time and turning it into an equivalent FNN. The training is conducted using this FNN model and the Backpropagation Through Time (BPTT) algorithm [11] and the resulting parameter values are used to configure the original RNN model. Finally, in the *action* part, the model weights are updated as a result of the training procedure.

4 Experimental Evaluation

4.1 Considered Datasets

The datasets used in this work, from which the inputs to the model were obtained, consist of channel change events, considering the day of the week and the time at which those events occured. Such events are directly obtainable in the STBs from the users' channel change actions (e.g., channel up button press).

For the analysis presented in this paper, two main datasets were considered: i) a synthetic dataset, mainly used to make an initial assessment of the model and its performance; and ii) a real user dataset, which represents the actual behaviour of a set of STB users. From the real user dataset, a subset of 30 users was also used to determine the hyperparameters of the model.

The synthetic dataset was synthesized to represent an artificial user that has a mostly consistent and predictable zapping pattern (i.e., a user that repeatedly watches the same channel on various days at the same time - e.g., to watch the prime time news program that airs every night at 8 p.m.) that repeats every week. To achieve this goal, two types of channel change events were added to this dataset: i) recurrent channel change events and ii) random channel change events. The recurrent channel change events are those that occur when the user is starting to watch his routine events (e.g., the news). These occur at similar hours and happen every week (with slight variations on the actual channel change time, e.g., +/-30 minutes). The random channel events represent the other channels that the user watches. For this dataset, the channel changes for a period of two months were synthesized, which made a total of 227 events. Due to the random channel events, the prediction accuracy when using this synthesized dataset is not 100% but is still quite high (85%), as shown in Table 2.

The real user dataset is comprised of the actual channel change events of 300 distinct STB users throughout a two months period. This dataset was split into two subsets: one that is used to determine the best hyperparameter values for the RNN, which is composed of the data pertaining to 30 users, and a second that is used to assess the performance of the proposed solution, which is composed of the data pertaining the remaining 270 users.

The 30 users subset, which is used to determine the hyperparameters of the RNN, is comprised of specifically selected users whose behaviour causes the RNN

to have a higher accuracy (denoted as best 30 users dataset). The selection of the users to be included in this dataset is based on a random and iterative process. This process starts by creating 100 distinct RNN models by randomly varying the hyperparameters. Then, each user from the real users dataset is randomly assigned to one RNN model and its accuracy is determined. Afterwards, the 5 users with the lowest accuracy are discarded. This process is repeated until 30 users are left, which will comprise the best 30 users dataset.

The approach to use only a subset of the users to determine the hyperparameters was preferred since not only it reduces the effort to find a good combination of hyperparameters but also contributes to exclude users that have no clear pattern from affecting the RNN model. Moreover, given the available dataset, it allows testing how a model generalises to other users.

4.2 Network Parameterization

Based on the defined model and using the best 30 users dataset, the RNN hyperparameters that are best suited to generate models to predict the next channels were determined. For each hyperparameter, various RNNs were generated (one for each possible value of a given hyperparameter). Each of these RNNs were then used to predict the next channels for each user in the best 30 users dataset and its accuracy was determined. It is relevant to note that a given hyperparameter may result in a higher accuracy for only a specific user. Hence, besides the average accuracy of the 30 users for a given hyperparameter value, the number of users that present a better accuracy for that hyperparameter value was also taken into account when deciding which hyperparameter to adopt. As part of the criteria to choose the best value for each of the hyperparameters, the training time and the model complexity (measured as the number of trainable parameters) were also considered, due to the fact that these are of particular relevance when considering that the training phase is executed in a device with limited computational capabilities, as is the case of a STB.

The evaluated hyperparameters were the following: dropout rate, mminimum Root Mean Square Error (RMSE), network type, number of hidden layers, unroll length, and number of weeks used in the training. The dropout rate is responsible for freezing some neurons (a percentage equivalent to the dropout rate) during the training phase, which counteracts the over-fitting of the RNN. The minimum RMSE is used as one of the two stop conditions of the chosen RNN (the other being a maximum of 1500 epochs per train); it represents the risk level of the RNN - if it is low, then the RNN may perform worst with the test dataset. The network type (identified by its neuron type) has a significant effect on the training time and the number of overall parameters, with the Long Short Term Memory (LSTM) being the most complex cell and the vanilla RNN being the least complex cell. The number of hidden layers used in the model linearly increases the number of parameters that require training. A higher number of hidden layers has the consequence of increasing the training time as well as the memory requirements of such model. Hence, it is advantageous to keep this value to the minimum possible. The unroll length determines the length of the

Table 1. Hyperparameters - assessed values and chosen values (depicted in bold and underline): # parameters - the amount of parameters to train in the model ($\times 10^3$); Train time - the time it takes to train 100 samples (in seconds); Accuracy - the average accuracy of the 30 users (in %); # users - the amount of users of the 30 user dataset that obtain better results with the given hyperparameter value

	Dropout rate value				Minimum RMSE					Neuron type			Weeks used to train				
	0.5	0.6	0.7	0.8	0.1	0.2	0.3	0.4	**0.5**	**LSTM**	GRU	Vanilla RNN	1	2	3	4	**5**
# parameters	14.1	14.1	14.1	14.1	14.1	14.1	14.1	14.1	14.1	14.1	10.9	4.5	14.1	14.1	14.1	14.1	14.1
Train time (s)	145	142	148	149	278	272	245	234	235	159	204	82	17	62	106	131	153
Accuracy (%)	60.2	59.4	58.7	54.1	61.8	61.5	61.8	62.1	61.9	60.3	56.2	53.6	44.4	51.5	55.6	59.0	60.4
# users	14	4	9	1	4	7	4	7	8	22	6	2	0	0	0	7	23

	Unroll length							Learning rate						Number of layers			
	2	3	**4**	5	6	7	8	**0.01**	0.03	0.05	0.07	0.09	0.11	**1**	2	3	4
# parameters	14.1	14.1	14.1	14.1	14.1	14.1	14.1	14.1	14.1	14.1	10.9	4.5	14.1	14.1	26.5	3.9	51.2
Train time (s)	161	147	152	204	255	304	373	159	212	228	253	273	314	159	623	935	963
Accuracy (%)	59.6	60.6	60.1	59.7	59.7	59.4	58.0	60.3	60.4	60.8	60.0	59.2	59.1	60.3	55.6	48.6	44.1
# users	6	7	6	4	3	3	1	6	5	8	2	3	6	24	5	1	0

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. The learning rate value affects how fast the model can change and, consequently, it influences how fast the model converges and whether or not it reaches the minimum RMSE risk or not. For this hyperparameter, it was preferred to reduce the training so that it can be better fitted to the restrictions of the embedded system for which it is targeted. 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, since it is not viable to store all channel change history in the STB from its initial set up and use it all to train the RNN.

To determine the best value for each of the RNN hyperparameters, a set with an initial random value for each hyperparameter was created. From this base set, other sets with different values for the hyperparameter under assessment were created. Subsequently, for each of those sets, the corresponding model was created and evaluated for each user in the best 30 user dataset. The hyperparameter value that provided the best performance was selected and the process continued with the next hyperparameter under assessment.

Table 1 shows a summary of the various hyperparameters, the used values in the assessment phase and the ones chosen as the model's hyperparameter values.

4.3 Accuracy Results

After having settled the RNN hyperparameters, the proposed solution was assessed using both the synthetic dataset - used to determine if the model behaves as expected - and the real user dataset. For the later analysis, the 270 real users dataset was used as well as the complete 300 users dataset. For each user, the accuracy is calculated on a per-week basis. Initially, only the first week of data is considered in the training and for the second week, the accuracy of the prediction is obtained. Subsequently, in the third week, the data from the previous two

Table 2. Accuracy and memory usage results for the model's execution on the used datasets: Artif - the artificial user dataset; Real users - the dataset which contains the events for 300 real users; 270 u - the subset containing 270 users of the real users dataset; 30 u - the subset of the best 30 real users used in the hyperparamter parameterization; all - all of the real users dataset (300); best - the user with the highest accuracy in the real users dataset; worst - the user with the lowest accuracy in the real users dataset

#Channels	Accuracy						Max mem (MB)	
	Artif	Real users					Artif	Real users
		270	30 best	All	Best	Worst		
1	83.5	49.3	57.3	50.2	81.6	10.7	0.5	2.4
2	88.3	55.3	69.0	56.9	83.7	17.0		
3	93.1	59.8	75.5	61.4	86.8	21.9		
4	94.9	63.0	80.2	64.8	88.3	25.5		
5	95.2	65.7	83.8	67.7	90.2	29.0		

weeks is used to train and the accuracy is obtained for the events in the third week. This process repeats itself until the full two months of data is considered. At the end, the average accuracy for each user is gathered and an average of the accuracy of all the users is presented.

Besides the average accuracy of the artificial user and of the real users, the accuracies for the worst user and the best user are also presented. In this situation, the worst user is the one for which the proposed solution does not improve much the channel change time, whereas the best user shows the model's performance with the highest prediction accuracy.

Table 2 shows the accuracy for the various datasets as well as the maximum amount of used RAM. It is possible to observe that the proposed model achieves a significantly high prediction success for the 270 real users dataset. It is also possible to observe that for the 300 real user dataset, the accuracy is slightly higher, however this dataset includes the users that were initially chosen to determine the model's hyperparameters, which inherently have a higher accuracy.

As expected, the artificial user dataset presents the highest accuracy values for the various channel configurations due to the way that this user was synthesized to have a regular and predictable behaviour. For the real users, it is worth noting that the model was able to have an average accuracy of 49.3% when considering the prediction of just one channel for the 270 users dataset (the most significant user group) and of 65.7% when considering a prediction of five channels. When considering the whole 300 real user dataset, the average accuracy is 50.2% for one channel (ranging from 81.6% to 10.7%) up to 67.7% for five channels (ranging from 90.2% to 29.0%).

4.4 Execution Performance and Required Resources

To validate and evaluate the viability of the proposed prediction mechanism, the developed model was executed in four different off-the-shelf embedded platforms, equipped with ARM processors commonly adopted by commercial STBs. Table 3 presents the characteristics of each considered embedded platform.

Table 3. Embedded platforms considered in the conducted performance evaluation.

	ARM1176JZF-S	Cortex-A9	Cortex-A53	Cortex-A72
Frequency	700 MHz	866 MHz	1.2 GHz	1.5 GHz
# Cores	1	2	4	4
DRAM	512 MB	497 MB	1 GB	8 GB

Table 4. Average training times (in seconds) in the considered embedded platforms.

	ARM1176JZF-S	Cortex-A9	Cortex-A53	Cortex-A72
Artificial user	0.91	0.30	0.19	0.20
Best user	303	133	72	14
Best 30 users	1614	546	506	310
Worst user	4608	4501	4281	1705
All users	3888	2758	1465	627

The executable binary and required libraries occupy 401 KBytes, which can be easily accommodated in the filesystem of any of these platforms. Since the training data is obtained directly from the existing log files and related data structures of the STB, it does not require any relevant added storage space. In what concerns the system memory (DRAM), it was observed that all conducted experiments did not require more than 2.4 MB. This value represents the amount of memory needed to allocate the RNN and the training dataset. As it can be also observed in Table 3, this (peak) memory requirement is easily satisfied by the considered embedded platforms and for most current STBs. It should be noted that this value is independent of the number of channels being predicted because the model runs only once and returns an ordered list of channels from which it is chosen the number of channels to consider.

Table 4 presents the observed training times for each of the considered embedded platforms. As it would be expected, the obtained performance is highly dependent on the processor family and on the corresponding operating frequency.

To ensure the best user's QoE, the considered setup assumed that the training procedure is executed weekly (thus conforming with the contents periodicity that is usually adopted by most TV broadcasting networks) and it was scheduled to a period of the day when the STB is less likely to be used for TV playback (i.e., 5 a.m.). Naturally, such scheduling can be easily tuned to each particular user profile based on the observation of its routine. Furthermore, and to avoid any possible perturbation to the user's QoE, a maximum timeframe was considered for the whole training procedure (i.e., 90 minutes). As it can be observed, all the considered models (apart from the rather unrealistic worst user scenario) do not saturate at this stop condition - even when executing in the most restricted computing platform, presenting an execution time that rarely exceeds an entire hour (3600 s). In particular, it was observed that 10 to 30 minutes is more than enough to run this training procedure for the most regular users. Therefore, the traning parameters and the resulting accuracy validate the feasibility of this model in these embedded platforms and opens space to define other system-optimized criteria to balance between the desired accuracy, the resulting saved

Table 5. Comparison of the proposed model with other models referred in the literature in what concerns the obtained average accuracy [%].

#Channels	Up&Down[3]	J48 Default[1]	Ada Boost[1]	CL[12]	SL[12]	Proposed
1	12.2	37.4	37.4	19.3	22.2	50.2
2	23.5	–	–	–	–	56.9
3	31.2	–	–	23.8	48.0	61.4
4	–	–	–	–	–	64.8
5	–	–	–	54.8	63.6	67.7

time upon a channel change, and the cost of fetching the required number of channels.

4.5 Comparison with Other Approaches

Table 5 compares the proposed model with other state-of-the-art models. For the models presented in [1] and [12], the accuracy was calculated using the datasets considered in the respective papers. Nevertheless, such results are expected to be very similar to those that would be obtained if the dataset used in the proposed model was used. For the Up&Down model [3], the presented results are a summary of the results reported by the authors but only considering the best combination of one, two, or three channels to be predicted. This last accuracy was obtained using the same dataset as the one used to train the proposed model.

As it can be observed, the proposed RNN model, specifically implemented for execution environments with low computational resources, clearly outperformed both the Up&Down model and the other tree-based models. Also, the accuracy of the proposed model is significantly higher than the one presented in [12] for one single channel prediction, although it is very similar to a five channels prediction. This is primarily due to the similarity of these two models, since both are LSTM based. Furthermore, it also shows that the proposed low-resource implementation for embedded environments based on the Kann framework [7] does not impact the resulting accuracy, since slight improvements could even be achieved when compared with the model trained without performance restrictions.

5 Conclusions

A new predictive system based on a RNN model was proposed to assist the tuning mechanism of current STBs in the forecast of the channel(s) a user will select next, in order to reduce the involved channel change delay. The implemented predictor makes use of log data concerning the user's channel changes history to train (and adjust) the model. Considering the tight computational constraints of most current STBs, the most convenient hyperparameter combination that still fulfills the aimed prediction accuracy had to be identified. The obtained results, validated using four embedded processor families commonly equipping commercial STBs, showed a prediction accuracy of 50.2% for a single-channel prediction and 67.7% when five channels were simultaneously predicted. When combined with the existing dual-tuning system of current STBs, the proposed

predictor can save as much as 1000 seconds per month in what concerns the TV channel change delay, greatly improving the resulting user's QoE.

Acknowledgements This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project UIDB/50021/2020.

References

1. Basicevic, I., Kukolj, D., Ocovaj, S., Cmiljanovic, G., Fimic, N.: A fast channel change technique based on channel prediction. IEEE Transactions on Consumer Electronics **64**(4) (2018). https://doi.org/10.1109/TCE.2018.2875271
2. Cha, M., Rodriguez, P., Crowcroft, J., Moon, S., Amatriain, X.: Watching television over an IP network. In: Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC (2008). https://doi.org/10.1145/1452520.1452529
3. Cho, C., Han, I., Jun, Y., Lee, H.: 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). https://doi.org/10.1109/icact.2004.1293012
4. Fimic, N., Basicevic, I., Teslic, N.: Reducing Channel Change Time by System Architecture Changes in DVB-S/C/T Set Top Boxes. IEEE Transactions on Consumer Electronics **65**(3) (2019). https://doi.org/10.1109/TCE.2019.2913361
5. Kim, Y., Park, J.K., Choi, H.J., Lee, S., Park, H., Kim, J., Lee, Z., Ko, K.: 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). https://doi.org/10.1109/ISBMSB.2008.4536621
6. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**, 436–44 (05 2015). https://doi.org/10.1038/nature14539
7. Li, H.: kann: A lightweight C library for artificial neural networks, https://github.com/attractivechaos/kann
8. Ramos, F.M., Crowcroft, J., Gibbens, R.J., Rodriguez, P., White, I.H.: Reducing channel change delay in IPTV by predictive pre-joining of TV channels. Signal Processing: Image Communication **26**(7) (2011). https://doi.org/10.1016/j.image.2011.03.005
9. Sherstinsky, A.: Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. Physica D: Nonlinear Phenomena **404**, 132306 (2020). https://doi.org/https://doi.org/10.1016/j.physd.2019.132306
10. Tongqing, Q., Zihui, G., Seungjoon, L., Jia, W., Qi, Z., Jun, X.: 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. p. 275–286 (2009). https://doi.org/10.1145/1555349.1555381
11. Werbos, P.J.: Backpropagation Through Time: What It Does and How to Do It. Proceedings of the IEEE **78**(10) (1990). https://doi.org/10.1109/5.58337
12. Yang, C., Ren, S., Liu, Y., Cao, H., Yuan, Q., Han, G.: Personalized Channel Recommendation Deep Learning from a Switch Sequence. IEEE Access **6** (2018). https://doi.org/10.1109/ACCESS.2018.2869470