Echo State Hoeffding Tree Learning

Diego Marrón DMARRON@AC.UPC.EDU
Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain

Jesse Read JESSE.READ@TELECOM-PARISTECH.FR
Albert Bifet ALBERT.BIFET@TELECOM-PARISTECH.FR
Talel Abdessalem TALEL.ABDESSALEM@TELECOM-PARISTECH.FR
LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay, 75013, Paris, France

Eduard Ayguadé EDUARD.AYGUADE@BSC.ES
Computer Sciences Department, Barcelona Supercomputing Center and Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain
José R. Herrero JOSEPR@AC.UPC.EDU
Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain

Editors: Robert J. Durrant and Kee-Eung Kim

Abstract

Nowadays, real-time classification of Big Data streams is becoming essential in a variety of application domains. While decision trees are powerful and easy–to–deploy approaches for accurate and fast learning from data streams, they are unable to capture the strong temporal dependences typically present in the input data. Recurrent Neural Networks are an alternative solution that include an internal memory to capture these temporal dependences; however their training is computationally very expensive, with slow convergence and not easy–to–deploy (large number of hyper-parameters). Reservoir Computing was proposed to reduce the computation requirements of the training phase but still include a feed-forward layer which requires a large number of parameters to tune. In this work we propose a novel architecture for real-time classification based on the combination of a Reservoir and a decision tree. This combination makes classification fast, reduces the number of hyper-parameters and keeps the good temporal properties of recurrent neural networks. The capabilities of the proposed architecture to learn some typical string-based functions with strong temporal dependences are evaluated in the paper. The paper shows how the new architecture is able to incrementally learn these functions in real-time with fast adaptation to unknown sequences and analyzes the influence of the reduced number of hyper-parameters in the behaviour of the proposed solution.

Keywords: Real-time classification, big data streams, Echo State Network, Hoeffding Tree, incremental learning, temporal dependencies.

1. Introduction

There is a need to perform real-time learning analytics of large amounts of data, usually generated in a decentralised fashion from a variety of data sources (e.g. social media, web feeds, IoT, sensors, mobile devices, etc.). This need is becoming the norm in a large variety of applications today that are dynamically fed with data that is short-lived and rapidly superseded by new data. Given the huge amounts of data generated, applications need to process data on-the-fly, with fast reaction and adaptation to changes.

This paper focusses on real-time classification, which introduces the following constraints: the classifier must be ready to predict at any time, deal with potentially infinite data streams and use each sample in the data stream only once (with limited amount of CPU cycles and memory). The preferred choice for real-time classification in a variety of applications is the use of incremental decision trees, which are able to learn with high accuracy and in a timely manner; however, these solutions fail to capture the strong temporal dependencies that are typically present in data streams because they make no consideration of past instances or labels at prediction time

Neural Networks (NN) are very popular nowadays due to the rapid growth of success stories of Deep Learning methods. Although deep neural networks can learn incrementally, they have so far proved to be too sensitive to their hyper-parameters and initial conditions; for this reason NN are not considered an effective *off–the–shelf* solution to process data streams [Marrón et al. (2016)]. Recurrent Neural Networks (RNN) are a type of NN with an internal memory that allow them to capture temporal dependencies. Training a RNN is challenging and requires a large amount of time, making them not viable for real-time learning [Werbos (1988); Martens and Sutskever (2011)]. In the recent years Reservoir Computing (RC) has emerged as an alternative for training RNN, aiming for a simpler and faster training [Maass et al. (2002); Jaeger (2001)]. RC can be seen as an standard NN that learns from what is called a "reservoir" unit, which is responsible for capturing the temporal dependencies of the input data stream. RC performs the training only at the readout step. Although conceptually simpler than most RNN, computationally cheap, and easier to implement, they still have high sensitivity to hyper-parameter configurations (i.e. small changes to any of them affect the accuracy in a non-predictable way [Lukoševičius (2012)]).

In this work we continue studying the use of NN (and RNN) to process data streams. In particular, this paper contributes with the proposal of a novel approach to learn time dependencies on data streams in real-time; we call it the Echo State Hoeffding Tree (ESHT) because it combines the use of the reservoir in the Echo State Network (ESN) proposal and a Hoeffding Tree (HT), both described in the next section. The proposed ESHT architecture requires few iterations to adapt to unseen sequences, and less hyper-parameters to tune (only two) than the standard ESN while still being able to model time dependencies. Also, the hyper-parameters needed by the ESHT are simpler to understand, and their influence on the final accuracy is more predictable.

As a proof of concept, we have built a prototype to evaluate the proposed architecture in terms of its ability to learn functions typically implemented by a programmer. In particular we use the *Counter* and *lastIndexOf* functions from the Java StringStream library and an *emailFilter*. The outputs of these functions clearly depend on the current string-stream historic, as it happens in handwriting or speech recognition tasks. A more efficient implementation is part of our future work, allowing us to perform a deeper evaluation of the proposed system with an extended set of functions and larger input datasets.

The rest of the paper is organised as follows: in Section 2 we discuss the state-of-the-art and related work. The proposed ESHT architecture design is presented in Section 3.

Section 4 evaluates ESHT with the aforementioned functions and shows the influence of the hyper-parameters that are used to configure it. Finally we conclude the paper and discuss some future work in Section 5.

2. State-of-the-art and Related Work

2.1. Incremental Decision Trees for Data Streams Classification

Incremental decision trees have been proposed in the literature for learning in data streams, making a single pass on data and using a fixed amount of memory. They are usually able to keep up with the rate at which data arrives and are easy to deploy (work out–of–the–box, no hyper-parameters to tune). The Hoeffding Tree (HT) and its variations [Domingos and Hulten (2000); Bifet et al. (2010); Ikonomovska et al. (2010)] are the most effective and widely used incremental decision trees able to build very complex trees with acceptable computational cost.

The HT makes use of the Hoeffding Bound (Hoeffding, 1963) to decide when and where to grow the tree with theoretical guarantees. Internal nodes are used to route a sample to the appropriate leaf where the sample is labelled. The HT produces a nearly-identical tree built by a conventional batch decision-tree inducer.

The main disadvantage of the HT and its variations is that they are not able to capture temporal dependencies on data streams. In the next section we briefly describe one of these variations, the FIMT-DD (Fast Incremental Model Tree with Drift Detection [Ikonomovska et al. (2010)]), which is used as one of the building blocks in our proposed architecture.

2.2. Neural Networks and Recurrent Neural Networks

Neural Networks (NN) are providing outstanding accuracy in many tasks, being able to outperform humans in tasks such as image recognition [He et al. (2015)] or even defeat professional Go players [Silver et al. (2016)].

Although NN can be trained incrementally [Rumelhart et al. (1988); Bottou (1998)] using backpropagation algorithms, each sample still requires two steps: 1) a forward step to compute the error, and 2) a backward propagation of the error which usually requires the computation of derivatives. NN are designed for batch learning requiring many iterations over the data and large amounts of data to achieve good accuracy, complicating a real-time response when the number of layers grow. Another important issue when applying NN for real-time analysis is their high sensitivity to hyper-parameter configurations that complicates their deployment [Marrón et al. (2016)].

In practice data streams usually present strong temporal dependencies that are not easily captured by typical NN. Recurrent Neural Networks (RNN) are a type of NN with an internal memory that allows them to exhibit dynamic temporal behaviour. RNN are widely used in natural language processing and speech/handwriting recognition [Sak et al. (2014); Graves et al. (2009)].

In exchange to the ability to model time, the training of an RNN is more complex than for a standard NN. This is mainly due to gradient explosion effect [Bengio et al. (1994)]: a small change to an iterative process can compound and result in very large effects many iterations later. There is no standard algorithm to train RNN. For example, Back Propagation Through Time (BPTT) [Mozer (1995); Robinson and Fallside (1987)] unfolds the RNN to train it similarly to a feed-forward NN with standard backpropagation algorithm. Each instance or batch of instances requires more computations, which complicates the *"be ready to predict at any time"* constraint for real-time classifiers. Hessian Free Optimisation [Martens and Sutskever (2011)] is a very effective method that uses the conjugate gradient to approximate the curvature matrix. It converges faster than BPTT but still requires a large amount of computations.

2.3. Reservoir Computing

Reservoir Computing (RC) uses a fixed random RNN called the *reservoir*, whose overall dynamics are driven by the input (and also affected by the past). Liquid-State Machines (LSM) [Maass et al. (2002)] and Echo State Networks (ESN) [Jaeger (2001)] are the two major reservoir-based proposals. RC has been applied in tasks with strong temporal dependencies, such as speech recognition [Skowronski and Harris (2007)] or predicting chaotic time series [Jaeger and Haas (2004)].

In this paper we focus on the ESN, whose architecture is shown in Figure 1. The ESN includes a single-layer reservoir, which we name *Echo State Layer* (ESL), and a fully-connected feed-forward NN. The ESL allows to the ESN to act as a dynamic short-term memory, outperforming RNN trained with Hessian Free Optimisation [Jaeger (2012)]; in addition, the ESN is also able to model non-linear patterns [Ozturk and Príncipe (2007)].

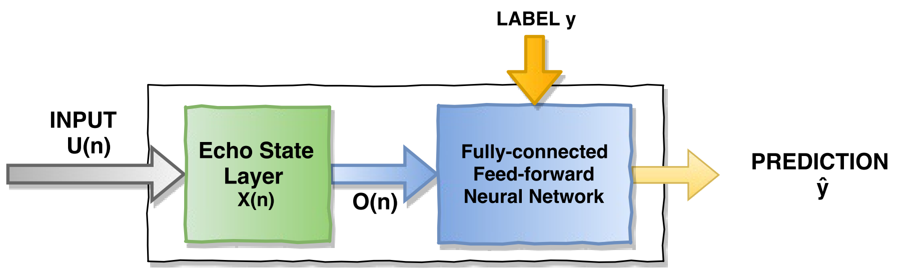


Figure 1: Echo State Network: Echo State Layer and fully-connected feed-forward NN

As we will show in the next section, the ESL update is computationally inexpensive, with no derivatives and no error backpropagation that are required to train other RNN. This fast update, while still being able to model temporal dependencies, makes the ESL very attractive for real-time analysis.

The ESL needs to satisfy the so called Echo State Property (ESP): for a long enough input U(n) the echo state X(n) has to asymptotically wash out any information from the initial conditions. The ESP is usually guaranteed for any input if the spectral radius of the ESL Weight Matrix is smaller than unit but is not limited to it: under some conditions the larger the amplitude of the input the further above the unit the spectral radius may be while still obtaining the ESP [Yildiz et al. (2012)].

3. Echo State Hoeffding Tree

In this section we present the Echo State Hoeffding Tree (ESHT), our new approach to learn from data streams with strong temporal dependencies. We propose a hybrid approach that combines the ESL ability to encode time dependencies in the ESN, with a very efficient incremental Hoeffding Tree regressor (the FIMT-DD [Ikonomovska et al. (2010)]). The proposed architecture is shown in Figure 2.

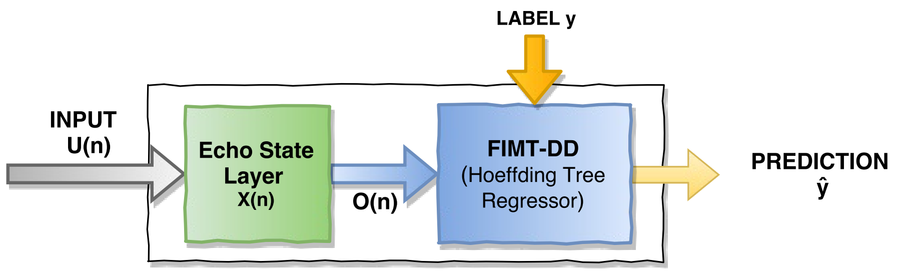


Figure 2: Echo State Hoeffding Tree design

The input data stream feeds an ESL, which is further detailed in Figure 3. It is a fixed single-layer RNN that transforms time-varying input U(n) to a spatio-temporal pattern of activations on the output O(n). The input $U(n) \in \mathbb{R}^K$ is connected to the echo state X(n) $\in \mathbb{R}^N$ through a weight matrix $W^{in}_{N,K}$. The echo state is connected to itself through a sparse weight matrix $W^{res}_{N,N}$.

$$\tilde{w}(n) = \tanh(W^{in}U(n) + W^{res}X(n-1)) \tag{1}$$

$$X(n) = (1-\alpha)X(n-1) + \alpha\tilde{w}(n), \tag{2}$$

$$O(n) = X(n) \tag{3}$$

The α is used during the echo state $X(n)$ update, and it controls how the echo state $X(n)$ is biased towards new states or past ones, i.e, controls how sensible the $X(n)$ is to outliers. Optionally, we could connect the input $U(n)$ to the output $O(n) \in \mathbb{R}^N$ using a weight matrix $W^{out}_{N,K}$. In this work, however, we do not use $W^{out}_{N,K}$ (see Eq. 3) since it requires the calculation of correlation matrices or pseudo-inverses which are computationally costly.

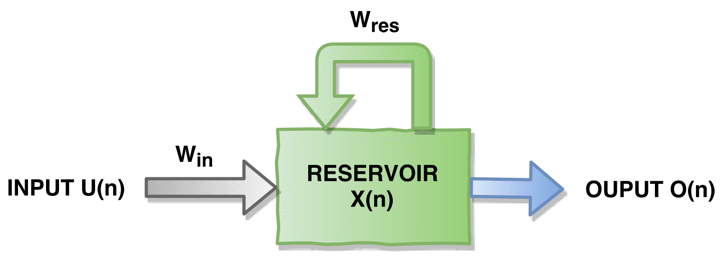


Figure 3: Echo State Layer

All matrices in the ESL are randomly initialised. The echo state $X(n)$ is initialised to zero and it is the only part that is updated during the execution. Note that the echo state $X(n)$ only depends on the input to change its state. As shown in Eq. 2, calculating the X(n) is computationally inexpensive since it only computes a weighted vector addition (compared to other RNNs training algorithms, the update cost is almost negligible).

The hyper-parameters required to configure the ESL are: α in Eq. 2, number of neurons, and density of the sparse matrix $W_{N,N}^{res}$ in Eq. 1 (in this work, density $\in (0, 1.0]$). Regarding the number of neurons in the echo state $X(n)$, [Lukoševičius (2012)] states as a generic rule to set it up proportional to the number of time steps an input should be remembered.

As can be observed in the architecture in Figure 2, our proposal replaces the fully connected feed-forward NN in the original ESN proposal with a FIMT-DD, a variation of the HT designed for regression. The basic idea of FIMT-DD is to train perceptrons in the leaves of the incremental decision tree by updating the weights after each consecutive example. The FIMT-DD includes all numerical attributes in the regression equation with linear output. The attribute split is done as in the standard HT for continuous attributes, and using the extended binary search tree [Gama (2003)] for numeric ones.

The main advantage of FIMT-DD is that it requires less samples than a NN to achieve good accuracy. In addition it is easier to deploy since it does not require the configuration of hyper-parameters, reducing the deployment complexity of the proposed architecture.

3.1. Methodology: Learning Functions

As a proof of concept, we propose the ESHT to learn functions typically implemented by programmers. In the evaluations we use what we call a *module* (Figure 3.1) that is composed of a label generator and an ESHT. A module learns one function by only looking at its inputs and output (no other information is provided to the ESHT) in real-time.

The label generator uses the function we want to learn to label the input. The input to the ESHT can be randomly generated or read from a file, and can be a single integer (Sections 4.1 and 4.2) or a vector (Sections 4.2 and 4.3). Both input and label are forwarded to the ESHT.

In this work we use only one module in the evaluations, but modules could be combined to solve complex tasks the same way programmers combine functions when writing programs. A potential application of this methodology is to treat programming as a black box: we could write the tests for a function and use the ESHT to learn the function instead of implementing it. This way, scaling computations is a matter of scaling a single model. This is part of our future work, once we clearly understand how ESHT behaves in a well optimised implementation.

4. Evaluation

This section evaluates the behaviour of the proposed ESHT architecture for learning three character-stream functions: *Counter*, *lastIndexOf* and *emailFilter*. Function *Counter* counts the number of elements that appear in the input between two consecutive zeros. Function *lastIndexOf* outputs the number of elements in the input since we last observed the current symbol. In other words, it counts the number of elements between two equal symbols in

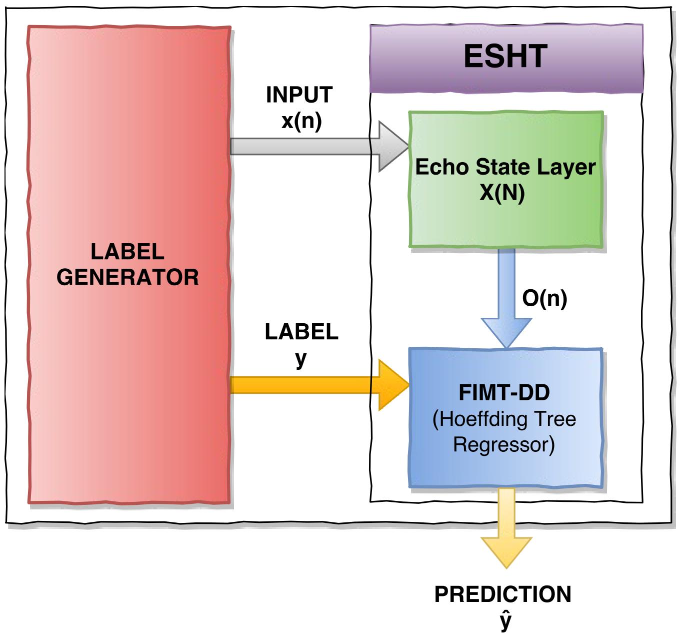


Figure 4: Module internal design: label generator and ESHT

the stream (i.e. use one *Counter* for each symbol). Finally, *emailFilter* is a function that detects valid email addresses in a character stream.

In order to understand the behaviour of ESHT we study the effect of its two hyper-parameters: α in Eq. 2 and the *density* of the sparse matrix $W^{res}_{N,N}$ in Eq. 1. We use *Counter* and *lastIndexOf* functions for this purpose. In both evaluations we fix the number of neurons to 1,000. In the *Counter* evaluation we test combinations of α and density in the range $[0.1, 1.0]$ in steps of 0.1. In the *lastIndexOf* evaluation we use the outcomes to test only some combinations of α in the same range.

We use *emailFilter* function to compare the behaviour of the proposed ESHT architecture with a FIMT-DD regressor tree, a standard fully-connected feed-forward NN and the ESN.

Two metrics are used for the purposes of evaluating the behaviour of ESHT, both derived from the *errors* detected in the output: *cumulative loss* and *accuracy*. An error occurs when the output is incorrectly classified, i.e. when $|y_t - \hat{y}| >= 0.5$, being $\hat{y}$ the predicted label and y_t the actual label; we use a distance of 0.5 since all labels are integer numbers. The *cumulative loss* shows the accumulated $|y_t - \hat{y}|$ for all the incorrectly classified inputs. And *accuracy* shows the proportion of correctly predicted labels with respect to the number of inputs.

Since ESHT is proposed for real-time analysis, it is also important to analyze the number of iterations that are needed in order to correctly output the correct label for a previously seen sequence. For example, for the *Counter* function evaluation, when we say that ESHT needs to observe a sequence two times, it means that at the third time the ESHT observes that sequence it outputs its correct length.

4.1. *Counter*

Two variations of the *Counter* function have been implemented (shown in Figure 5). In *Opt1* the label for each symbol in the input stream is the number of symbols since the last zero appeared (and 0 when the zero symbol appears); in *Opt2* the label is different than 0 only when zero appears in the input stream, returning in this case the number of symbols since the previous zero appeared.

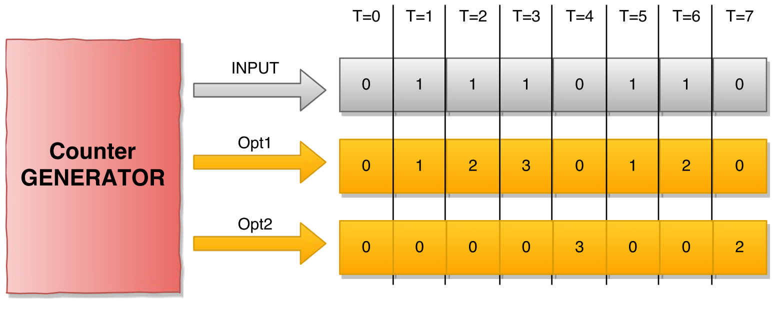


Figure 5: *Counter* generator functions

The input stream is a random sequence of 0/1 symbols generated following a normal distribution in order to analyze the influence of parameters α and *density* on the *loss* and *accuracy* mentioned in the previous section for an input of 1,000 samples.

From a visual inspection of the output generated, the first conclusion that we obtain is that ESTH is able to learn possible sequences of the input symbols after seeing them two or three times. Figure 6 shows the evolution along time (number of input symbols) for the cumulative *loss* and *accuracy*, for each *Counter* option and for two different combinations of parameters α and *density*.

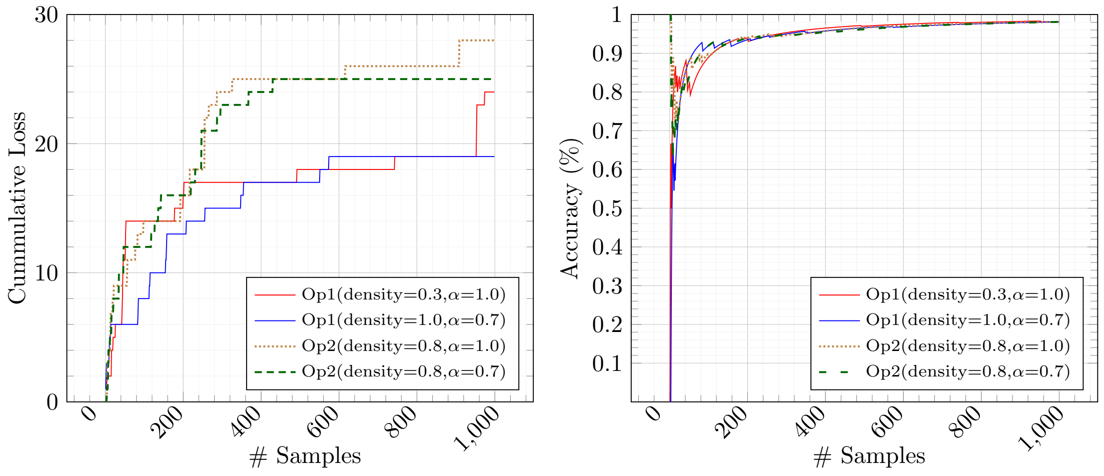
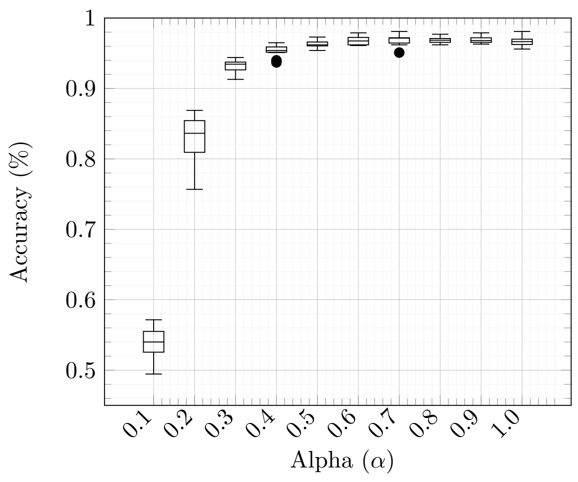


Figure 6: Cumulative *loss* (left) and *accuracy* (right) on the Counter stream.

The first observation from the cumulative *loss* plot on the left is that the *errors* and *loss* rapidly decrease with time. This is the expected behaviour for a randomly generated sequence that follows a normal distribution: the more samples we generate the less chance of an unseen sequence (giving ESHT ability to learn already seen sequences). From the *accuracy* plot on the right of Figure 6 we conclude that an accuracy of 0.9 is achieved with only a few hundred samples (200 in this specific configuration of the α and *density* parameters); in other words, almost all loss is incurred on the first few hundreds of samples, and after this, the loss stabilizes. Some of the results have a temporal accuracy of 1 for the first item; this is due to the first item in the input-stream being always zero and so its label, and the ESHT always outputs a zero for the first element.

Figure 7 shows the influence of the α and *density* parameters in the *accuracy*. In both plots, the horizontal axis shows the variation of one of the parameters while the box plot shows the variation of the other parameter (with values inside the box that have an accuracy with the standard deviation). The plot on the left shows that there is a monotonic growth of the *accuracy* with parameter α; lower values for α place relatively more importance on older reservoir states (see Eq.2), which has the effect that it takes longer for the model to learn new sequences. In this same plot, the influence of *density* seems to have a less relevant influence. In fact the plot on the right shows that there is no clear correlation between *density* and *accuracy*. The outliers in that plot correspond to the low values of α that were already commented in the previous plot.



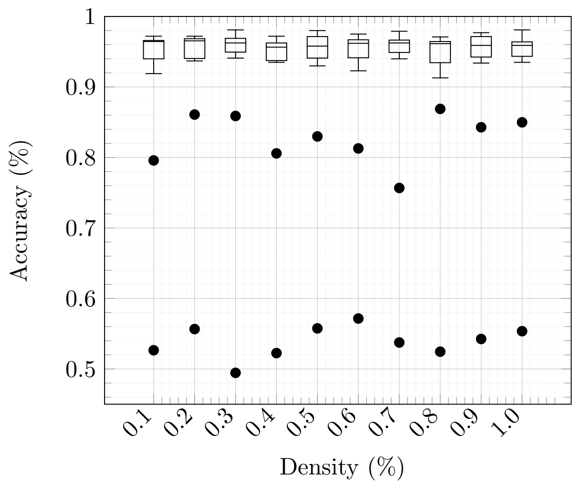


Figure 7: Influence of parameters α and *density* on the *Counter* stream. In each figure, the box plot shows the influence of the other parameter.

4.2. *lastIndexOf*

Figure 8 shows the output of the *lastIndexOf* function (which is how this function is known to Java programmers). Given a sequence of input symbols, the function returns for each symbol the relative position of the last occurrence of the same symbol (i.e. how many time steps ago the symbol was last observed). Note that for each time-step all symbols but the current one are one step farther, thus, generating a highly dynamic output.

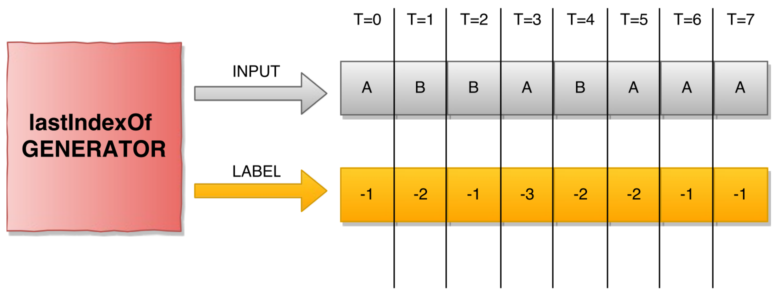


Figure 8: *lastIndexOf* generation function

The input stream is a sequence of symbols of an alphabet randomly generated following a normal distribution. Sequences of up to 10,000 samples and alphabets of 2, 3 and 4 symbols have been used to perform the evaluation of ESHT in terms of *accuracy*.

As the number of symbols in the alphabet grows more samples are needed to learn a pattern, since the number of combinations grows exponentially with the number of symbols. Figure 9 shows this trend for two different pairs of values α and *density*. Alphabets with 2 and 3 symbols are relatively simple to be learnt (the ESHT achieved 80+% accuracy with only 1,000 samples) while with a 4-symbols alphabet the ESHT needed 10,000 samples to achieve 75% accuracy.

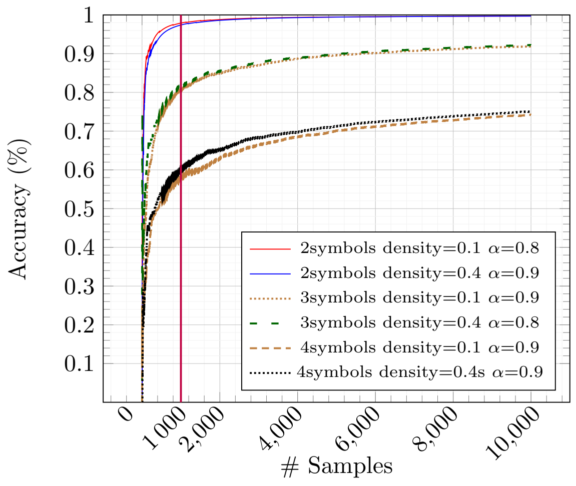


Figure 9: Accuracy for the *lastIndexOf* function for alphabets with 2, 3 and 4 symbols.

From a visual inspection on the ESL output we observed that a relatively small number of samples were needed to saturate the output signal. To delay this saturation we decided to use a vector of features (one element for each symbol) as the input instead of a scalar value. The position corresponding to the current symbol index is set equal to *0.5* and the rest equal to zero. This way, the input signal to the FIMT-DD has different levels (in contrast to the saturated signal observed when using a scalar input). Figure 10 shows the improvement of using a vector instead of scalar input for different values of the α hyper-parameter. For the rest of the evaluations in this subsection we will use the vector input.

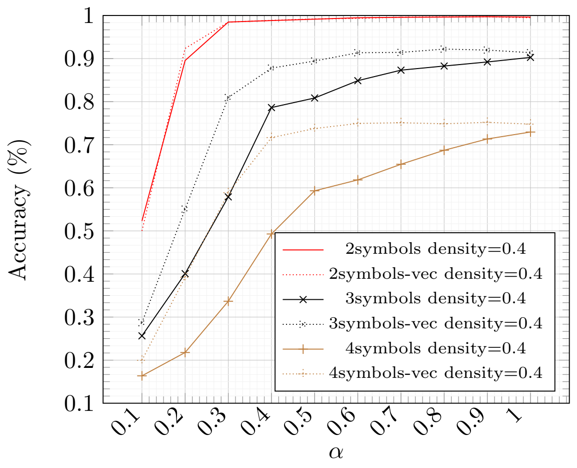


Figure 10: Effect on the accuracy of coding the input to *lastIndexOf* as a scalar or a vector of features (density=0.4)

Figure 11 shows the influence of the α and *density* hyper-parameters on the *accuracy*. In both plots, the horizontal axis shows the variation of one of the parameters and different lines are used to show the variation of the other parameter. From the plot on the left it is clear the monotonic growth of the *accuracy* with parameter α. In this same plot, the influence of *density* seems to have a less relevant influence if the value of *alpha* is correctly set. In fact the plot on the right shows that there is not a clear correlation between *density* and *accuracy*. Similarly, one could predict a similar conclusion when changing the number of neurons in the ESL.

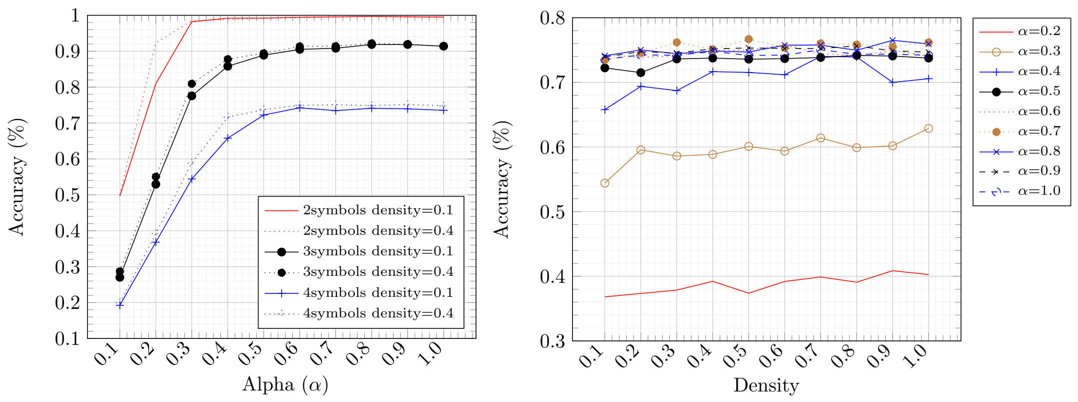


Figure 11: Effect of *alpha* and *density* on the accuracy for *lastIndexOf*

4.3. *emailFilter*

The *emailFilter* function labels the input stream with the length of the email address detected or *0* otherwise (including wrong formatted email address). For the evaluation of the ESHT and comparison with previous proposals we use a synthetic dataset based on the *20 newsgroups* dataset, that comprises around 18,000 newsgroups posts on 20 topics [Lang (2008)]. We extracted a total of 590 characters and repeated them eight times.

Each repetition, or block, contains 11 email addresses and random text (including wrong formatted email addresses) at the same proportion. The resulting dataset has a label balance of 97.8% zeros.

Based on the conclusion from Section 4.2, we decided to represent the input as a vector of features, one for each symbol in the alphabet. However, this would require a vector for an ASCII encoded input, which would increase the memory consumption (larger input matrix on the ESL) and would require more samples to abstract a pattern. To speed up the learning process we reduced the input space to only four symbols, those strictly necessary to identify a correctly formatted email address. Table 1 shows the map used to create the 4-symbols dataset. The reduced input vector implies faster vector-matrix multiplication (low dimensionality) and less memory consumption (due to smaller matrix size). In addition, the reduced input space improves the learning speed.

Table 1: Map from ASCII domain to 4-symbols

ASCII Domain	4-Symbols Domain	
Original Symbols	Target Symbol	Target Symbol Index
[\t\n\r]+	Single space	0
[a–zA–Z0−9]	x	1
@	@	2
[.]+	Single dot	3

For the comparison we configured the different algorithms (FIMT-DD, feed-forward NN, ESN and ESHT) as shown in Table 2. For the ESN we explored different values for the α, *density* and *learning rate* hyper-parameters in the range $[0.1, 1.0]$ and linear output. For the standard NN we also explored values for the *learning rate* in the same range. In order to configure ESHT, we used the results obtained for *Counter Opt2* in section 4.1, with $\alpha = 1.0$ and *density*=0.4. We increased the number of neurons to 4,000.

Table 2: Email address detector results

Algorithm	Density	α	Learning rate	Loss	# Errors	Accuracy (%)
FIMT-DD	-	-	-	4,119.7	336	91.61
NN	-	-	0.8	2,760	88	97.80
ESN1	0.2	1.0	0.1	1,032	57	98.47
ESN2	0.7	1.0	0.1	850	61	98.47
ESHT	0.1	1.0	-	**180**	**10**	**99.75**

The first conclusion from the results shown in Table 2 is the well known inability of both FIMT-DD and NN to capture time dependencies in the input. The NN defaults to the majority class (always predicts a *0* symbol), achieving 97.80% of accuracy (88 errors, the total number of correct email addresses in the dataset input) with loss of 2,760 (the total length of the correct emails in the dataset input). The FIMT-DD obtains a worse accuracy (91.61% with loss 4,119.70).

ESHT clearly outperforms the two best configurations obtained for ESN, with only 10 errors and a cumulative loss of 180 (compared to around 60 errors and cumulative error around 1,000 in ESN). In order to better understand the results shown in Table 2 for ESN and ESHT, the left plot in Figure 12 shows the *cumulative loss* evolution with the number of samples in the input. After eight repetitions the ESN failed to get right all the 11 emails in the same block (observe how the *cumulative loss* continues to grow with the number of samples). The ESHT clearly outperforms the ESN with only 500 samples; after this number of samples, the plot shows a constant loss for the ESHT between 500 and 1,000 samples (this is an effect of the plot scale, in this range the loss grows, but after this it stays constant).

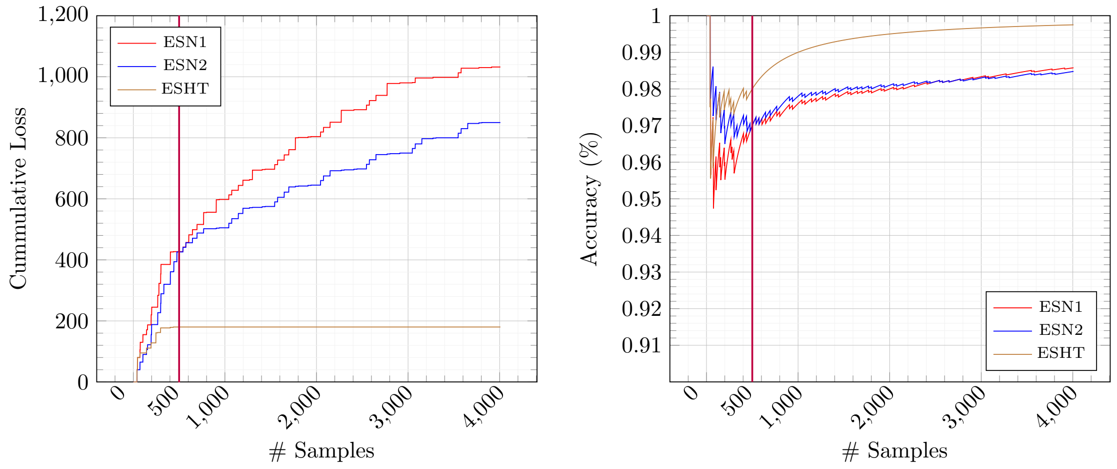


Figure 12: *Cumulative loss* (left) and *accuracy* (right) evolution for *emailFilter*

The right plot in Figure 12 shows the evolution of the accuracy of ESN and ESHT with the number of samples in the input. Observe that the three curves start with 100% accuracy; this is due to the fact that the first label is zero, and all tests started biased to zero.

5. Conclusions and Future Work

This paper proposes a novel architecture, the Echo State Hoeffding Tree (ESHT), to learn temporal dependencies in data streams in real-time. The proposal is based on the combination of the reservoir in the Echo State Network (ESN) and a FIMT-DD regressor tree. The paper also evaluates the proposed architecture with a proof-of-concept implementation and three string-based input sequences generated by functions typically implemented by a programmer. ESHT is able to learn faster than standard ESN, and requires less hyper-parameters to be tuned (only two). The hyper-parameters required by the ESHT have a more predictable effect on the final accuracy than the hyper-parameters in typical neural networks (such as learning rate or momentum). The comparison with a standard feed-forward neural network and the FIMT-DD itself shows the ability to capture the temporal dependencies in data streams that these other two architectures are unable to capture.

The paper also shows the ability of the ESHT to learn functions typically implemented by programmers, opening the door to explore the possibilities of Learning Functions instead of programming them.

The current proof-of-concept implementation of our architecture limits the number of samples that we can use in our tests (due to the large execution time). We are currently reimplementing the proposed ESHT in order to be able to use much larger input sequences and properly study the effects of the initial state vanishing in long runs. Controlling the time-window stored in ESL is an interesting feature for data-streams, specially in scenarios where drifting is present.

Acknowledgments

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2015-65316-P project, by the Generalitat de Catalunya (contract 2014-SGR-1051), by the Universitat Politècnica de Catalunya through an FPI/UPC scholarship and by NVIDIA through the UPC/BSC GPU Center of Excellence.

References

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

Albert Bifet, Geoffrey Holmes, and Bernhard Pfahringer. Leveraging bagging for evolving data streams. *ECML PKDD'10*, pages 135–150, 2010.

Léon Bottou. Online algorithms and stochastic approximations. In *Online Learning and Neural Networks*, pages 9–42. Cambridge University Press, 1998.

Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2000.

João Gama. Accurate decision trees for mining high-speed data streams. In *In Proc. SIGKDD*, pages 523–528. ACM Press, 2003.

Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(5):855–868, May 2009. ISSN 0162-8828. doi: 10.1109/TPAMI.2008.137.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *The IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, December 2015.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

Elena Ikonomovska, João Gama, and Sašo Džeroski. Learning model trees from evolving data streams. *Data Mining and Knowledge Discovery*, 23(1):128–168, 2010. ISSN 1573-756X.

Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks - with an erratum note. Technical report, German National Research Center for Information Technology, 2001. URL http://www.faculty.jacobs-university.de/hjaeger/pubs/EchoStatesTechRep.pdf.

Herbert Jaeger. Long Short-Term Memory in Echo State Networks: Details of a Simulation Study. Technical Report 27, Jacobs University, Bremen, Germany, February 2012. URL http://minds.jacobs-university.de/pubs.

Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004. ISSN 0036-8075.

K. Lang. 20 newsgroups data set, 2008. URL http://people.csail.mit.edu/jrennie/20Newsgroups/. Last accessed: October 2016.

Mantas Lukoševičius. A Practical Guide to Applying Echo State Networks. In *Neural Networks: Tricks of the Trade*, volume 7700 of *LNCS*, chapter 27, pages 659–686. Springer Berlin Heidelberg, 2012.

Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Comput.*, 14(11):2531–2560, November 2002. ISSN 0899-7667.

Diego Marrón, Jesse Read, Albert Bifet, and Nacho Navarro. Data stream classification using random feature functions and novel method combinations. *Journal of Systems and Software*, 2016. ISSN 0164-1212. doi: 10.1016/j.jss.2016.06.009.

James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 1033–1040, 2011.

Michael C. Mozer. A focused backpropagation algorithm for temporal pattern recognition. In Yves Chauvin and David E. Rumelhart, editors, *Backpropagation*, pages 137–169. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995. ISBN 0-8058-1259-8.

Mustafa C. Ozturk and José Carlos Príncipe. An associative memory readout for ESNs with applications to dynamical pattern recognition. *Neural Networks*, 20(3):377–390, 2007.

A. J. Robinson and F. Fallside. The Utility Driven Dynamic Error Propagation Network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. In James A. Anderson and Edward Rosenfeld, editors, *Neurocomputing: Foundations of Research*, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6.

Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, pages 338–342, 2014.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

Mark D. Skowronski and John G. Harris. Automatic speech recognition using a predictive echo state network classifier. *Neural Networks*, 20(3):414 – 423, 2007. ISSN 0893-6080.

P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, pages 339 – 356, 1988.

Izzet B. Yildiz, Herbert Jaeger, and Stefan J. Kiebel. Re-visiting the echo state property. *Neural Networks*, 35:1 – 9, 2012. ISSN 0893-6080.