Faster Classification of Time-Series Input Streams

Kunal Agrawal
Washington University in Saint Louis, MO, USA

Sanjoy Baruah
Washington University in Saint Louis, MO, USA

Zhishan Guo
North Carolina State University, Raleigh, NC, USA

Jing Li
New Jersey Institute of Technology, Newark, NJ, USA

Federico Reghenzani
Politecnico di Milano, Italy

Kecheng Yang
Texas State University, San Marcos, TX, USA

Jinhao Zhao
Washington University in Saint Louis, MO, USA

Abstract

Deep learning–based classifiers are widely used for perception in autonomous Cyber-Physical Systems (CPS's). However, such classifiers rarely offer guarantees of perfect accuracy while being optimized for efficiency. To support safety-critical perception, ensembles of multiple different classifiers working in concert are typically used. Since CPS's interact with the physical world continuously, it is not unreasonable to expect dependencies among successive inputs in a stream of sensor data. Prior work introduced a classification technique that leverages these inter-input dependencies to reduce the average time to successful classification using classifier ensembles. In this paper, we propose generalizations to this classification technique, both in the improved generation of classifier cascades and the modeling of temporal dependencies. We demonstrate, through theoretical analysis and numerical evaluation, that our approach achieves further reductions in average classification latency compared to the prior methods.

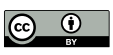
2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases Classification, Deep Learning, Sensor data streams, IDK classifiers

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2025.13

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15429421`
Software (ECRTS 2025 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.11.1.4`

Funding *Kunal Agrawal*: NSF Grant No. CCF-2106699, CCF-2107280, and PPoSS-2216971.
Sanjoy Baruah: NSF Grant No. CNS-2141256 and CPS-2229290.
Zhishan Guo: NSF Grant No. CMMI-2246672.
Jing Li: NSF Grant No. CNS-2340171, DOE ASCR Award Number DE-SC0024424.
Federico Reghenzani: NGI Enrichers Transatlantic Fellowship Programme, National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data and Quantum Computing.
Kecheng Yang: NSF Grant No. CNS-2104181.



1 Introduction

In order to obtain an understanding of the physical world within which they are operating, autonomous Cyber-Physical Systems (CPS's) repeatedly sense their operating environment and attempt to classify sensed signals as representing objects from one of a pre-defined set of classes. Such classification is commonly done using classifiers that are based upon Deep Learning and related AI techniques. These classifiers need to make accurate predictions in real time, even when implemented upon edge devices with limited computational capabilities. However, most classical machine learning techniques emphasize accuracy over efficiency of implementation: classical techniques are highly accurate but can be quite time-consuming even on very simple inputs. Classifier *cascades*, in which very resource-efficient classifiers of limited accuracy first attempt to classify each input with more powerful (but less efficient) classifiers being used only upon inputs on which these efficient ones fail, have been proposed as a means of balancing the contrasting needs of efficiency and accuracy. Such classifier cascades may be constructed using *IDK classifiers* [15]. An IDK classifier is obtained from some base classifier in the following manner: if the base classifier cannot make a decision with confidence exceeding a predefined threshold, it outputs a placeholder class labeled as IDK, meaning "I Don't Know." Multiple IDK classifiers can be trained for a given classification problem, each offering varying execution times and likelihoods of producing a definitive class rather than IDK. Wang et al. [15] proposed organizing such collections of IDK classifiers into an *IDK cascade*: a linear arrangement of (some or all) the IDK classifiers that specifies the order in which they are to be called upon any input until a non-IDK classification is obtained. For applications where every input must ultimately receive a real classification, a *deterministic* classifier, which always produces a real class, is included as the final classifier of the IDK cascade – this deterministic classifier being unable to classify an input constitutes a system fault, potentially triggering recovery mechanisms.

Temporal dependences in input streams. Most mobile perception pipelines require that streams of input values that are obtained by sensors each be classified. It is reasonable to hypothesize some dependence among successive inputs in such time-series readings from a single sensor source. Consider, for instance, a stream of frames recorded by a camera in an autonomous vehicle, where each designated Region of Interest (RoI) is tracked across the frames in the stream. If a specific classifier accurately identifies a RoI in one frame, it is likely to be capable of classifying the same RoI in the subsequent several frames as well, until perhaps eventually failing because the object has moved too far away and needs a more sophisticated DL model for accurate recognition. While prior work studied the problem of constructing IDK cascades [6, 5, 3], the potential inter-input dependencies were not exploited when using IDK classifier cascades, which always started by invoking the first classifier in the cascade and moving through the cascade until a non-IDK classification was returned. Agrawal et al. [2] addressed this shortcoming in the prior state of the art by conducting a methodical study of the phenomenon of temporal dependence in time-series input streams. Specifically, they (i) characterized and formally defined a particular form of dependence, which quantifies the probability that the exactly same set of classifiers in the IDK cascade return real (i.e., non-IDK) classes on consecutive inputs; (ii) proposed a schema for learning the degree of dependence that may be present in a particular input stream; and (iii) presented and evaluated algorithms that are capable of exploiting the potential presence of such dependences to speed up the average duration to successful classification.

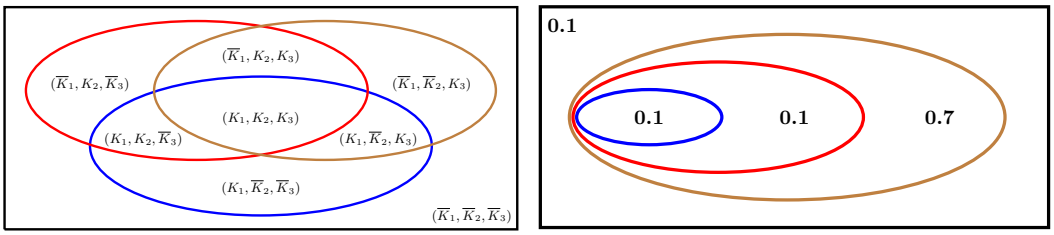
This work. We build upon and extend the findings of Agrawal et al. [2], a seminal work in the field of embedded edge AI that, to the best of our knowledge, was the first to explore the exploitation of temporal dependencies in time-series input streams. In this paper, we aim to further improve classification performance in such settings. Our specific ***contributions*** are as follows:

1. We derived *an improved algorithm* for exploiting the form of temporal dependence. The improvement comes from the conceptual realization that simply commencing classification from a different classifier than the one at the start of the cascade is not optimal – one can achieve further reduction in average classification duration by additionally skipping classifiers within the cascade based on the classification of the previous input.
2. For this improved algorithm to be practically useful, we needed a technical breakthrough: since skipping classifiers within the cascade is essentially equivalent to synthesizing a new cascade from the ones present in the original cascade, we needed a speedier cascade-synthesis algorithm than the previously-proposed algorithm [4] with $\Theta(n^2)$ complexity, in order to be able to use it during online classification time for each and every continuous input. We achieved a *significant improvement in efficiency* by developing a linear-time $\Theta(n)$ algorithm for synthesizing cascades.
3. Additionally, we formulated *a more general notion of dependence* for time-series input streams, which more accurately reflects the kinds of temporal dependence that is likely to be encountered in embedded systems. We extended our improved algorithm to effectively exploit this more general form of dependence as well. Further extending the notion of dependence to a logical endpoint, *a completely general characterization* of dependence via Markov Decision Processes is also discussed.
4. We performed *experimental evaluation* of our improved algorithm with both the original and the more general notion of temporal dependence.

Organization. The remainder of this manuscript is organized as follows. In Section 2, we describe the model of IDK classifiers and cascades commonly used in the embedded systems community. In Section 3, we delve into the issue of temporal dependence in sensed data: we briefly summarize the main aspects of the research that we are building on (Section 3.1), and use this to motivate our proposed modifications (Section 3.2). As stated above, we needed to make a significant improvement in runtime efficiency to a previously-proposed algorithm [4] for synthesizing cascades in order for our proposed modifications to be practically realizable – this is described in Section 4. Section 5 presents our new runtime classification algorithm, and Section 6 reports on our experimental evaluation of this new algorithm as compared to the prior state-of-the-art. In Section 7, we briefly discuss a further (and very general) extension of the model of dependence that we have considered in this paper – an area to explore in greater depth in future research. Finally, we conclude in Section 8.

2 A Real-Time Model and Practical Considerations for IDK Cascades

In this section, we briefly describe the formal model for representing IDK classifiers that was defined in Abdelzaher et al. [1], and is now commonly used in the real-time scheduling literature. We suppose that there are multiple IDK classifiers denoted $K_1, K_2, K_3, \ldots, K_{n-1}$, as well as a deterministic classifier K_n, that have all been trained to solve the same classification problem. A probabilistic characterization of the classification capabilities of the different classifiers is provided – this may be visually represented in a Venn diagram with each region corresponding to some combination of the individual classifiers returning either a real class or IDK for an input (see Figure 1 (a) for an example with three classifiers). Each classifier K_i is also characterized by a worst case execution time (WCET) C_i.



(a) (b)

Figure 1 (From [2]) **(a)**: The probability space for three IDK classifiers and one deterministic classifier. The blue, red, and brown ellipses respectively denote the regions of the probability space where the classifiers K_1, K_2, and K_3 are successful (i.e., do not output IDK). The enclosing rectangle denotes the region in which the deterministic classifier is successful (i.e., all inputs). Each of the disjoint regions partitioned by the ellipses is labeled with a 3-tuple, with K_i ($\overline{K}_i$, respectively) denoting that the IDK classifier K_i returns a real class (resp. IDK) in this region. **(b)**: An example Venn diagram for *contained classfiers*. The numbers denote the associated probabilities.

Given such a collection of classifiers, an algorithm was derived [1] for constructing IDK cascades that minimize the expected duration required to achieve successful classification while optionally adhering to a specified latency constraint; this algorithm has a worst-case running time of $\mathcal{O}(4^n)$ where n denotes the number of IDK classifiers. More efficient algorithms with $\mathcal{O}(n^2)$ worst-case running time have been obtained [4] for the special case where the IDK classifiers satisfy the *containment* property[1]: the set of ***contained classifiers*** can be strictly ordered from least to most powerful, such that any input successfully classified by a particular classifier in the set is also successfully classified by all more powerful classifiers (see Figure 1 (b) for a Venn diagram representation).

Practical Considerations for IDK Cascades. In the broader machine learning community, IDK cascades are part of adaptive inference and dynamic neural networks that dynamically adjust effort (e.g., skip layers, select among multiple sub-models, or adapt the input resolution) based on input difficulty [11, 8, 9]. The idea of cascades is also related to anytime prediction – the computation can be truncated as soon as sufficient confidence is achieved. While IDK cascades and anytime prediction are orthogonal and incomparable to each other in terms of resource efficiency, real-time analyses for IDK cascades may be extended to anytime prediction. For instance, early-exit networks can be considered as an internal realization of the contained IDK cascade, where a single deep network is augmented with multiple intermediate classifiers (exits) so that inference can "exit" early if an earlier layer's prediction is confident [12, 10]. Here, the cost of skipping intermediate classifiers needs a modified analysis. Techniques on model compression, such as model quantization, pruning, and knowledge distillation, can also be combined with cascades. For example, more aggressively quantized models can serve as faster yet less accurate classifiers in the contained IDK cascade. These techniques all aim to balance accuracy and efficiency, a trade-off at the heart of the IDK cascade design philosophy. Recent work [14] has demonstrated the effectiveness of IDK cascades in various domains, including autonomous systems, real-time surveillance, and industrial automation, further supporting the relevance and practicality of IDK cascades.

[1] Containment was called 'full dependence' in [4]. Agrawal et al. [2] proposed the change to 'containment' since 'full dependence' may lead to confusion with the notion of temporal dependences amongst inputs.

From an implementation perspective, IDK cascades can be executed such that the follow-up model in the cascade is pre-loaded onto the computing devices (e.g., GPU) during the running of the current model to reduce runtime overhead. Depending on the available computation capacity of the application domain (e.g., real-time monitoring using IoT micro-controllers, object classification on autonomous vehicles, traffic control or smart manufacturing on the edge, city-wide surveillance system on the cloud), the number of models in such cascades can range from a dozen to several hundred.

In many application domains, like perception for autonomous driving, models are frequently pre-trained on the same or highly similar datasets. In such settings, larger and deeper models tend to consistently outperform smaller ones across most input types, giving rise to contained IDK cascades, where every input classified by a simpler model is also correctly classified by all more complex models in the cascade. Similarly, techniques like model compression and anytime prediction also naturally result in contained classifiers. Even in cases where this containment property is not strictly satisfied, the discrepancy is often minor, making it practical to approximate real-world cascades as contained with negligible loss in performance.

3 Temporal Dependences in Time-Series Input Streams

As stated in Section 2, earlier cascade-synthesis algorithms [1, 4] have been shown to be optimal for minimizing expected duration to successful classification. These optimality results, however, only hold under the assumption that there is no dependence between different inputs that are to be classified: each input is assumed to have been drawn independently from the underlying probability distribution. Focusing on the case of contained classifiers, Agrawal et al. [2] sought to exploit the possibility that some dependence between successive inputs is likely to be present. They formalized a quantitative metric of dependence in time-series input streams (Definition 1 below) in order to further reduce the average classification time.

Let $\vec{x} \stackrel{\text{def}}{=} \langle x_1, x_2, x_3 \ldots \rangle$ denote a stream of inputs to be classified (i.e., the inputs $x_1, x_2, x_3, \ldots$ must each be classified, one at a time in increasing order of the subscript) by a particular IDK cascade $\mathcal{K}$. Two inputs in $\vec{x}$ are defined to be ***equivalent inputs*** if and only if exactly the same set of classifiers in $\mathcal{K}$ would return real (i.e., non-IDK) classes upon both inputs. The defined dependence parameter λ of input stream $\vec{x}$ with respect to $\mathcal{K}$ is a quantitative metric of the likelihood that successive inputs are equivalent inputs.

▶ **Definition 1** (dependence parameter λ)**.** *Time-series input stream $\vec{x} \stackrel{def}{=} \langle x_1, x_2, x_3 \ldots \rangle$ that is to be classified by cascade $\mathcal{K}$ has dependence parameter λ, $0 \leq \lambda \leq 1$, if and only if for each $t > 1$ the input x_t is*

- *with probability λ, equivalent to the input x_{t-1}; and*
- *with probability $(1 - \lambda)$, drawn at random from the underlying probability distribution.*

Thus, a small value of λ indicates little dependence between successive inputs (streams where each input is independently drawn from the underlying distribution have $\lambda = 0$) and larger λ denotes greater dependence (when $\lambda = 1$ all the inputs are equivalent inputs – they are successfully classified by exactly the same set of classifiers in the cascade $\mathcal{K}$).

Note that since temporal dependence is primarily a property of an input stream, having a single dependence (λ) parameter to model its impact upon all the classifiers in the cascade is an oversimplification. Greater modeling accuracy (and a consequent additional reduction in classification duration) can be achieved by having one parameter to model the effect of the stream's temporal dependence on each classifier. In Section 5.1, we proposed this more general model and adapted our algorithm to be compatible with this generalized model.

K_i	K_1	K_2	K_3	K_4
C_i	1	11	100	200
P_i	0.1	0.2	0.9	1.0

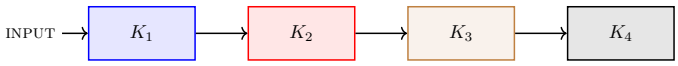


Figure 2 An example cascade $\mathcal{K} = \langle K_1, K_2, K_3, K_4 \rangle$ of contained classifiers. K_4 is the deterministic classifier. C_i denotes the WCET of K_i, and P_i the probability that K_i will return a real (i.e., non-IDK) classification on an input drawn at random from the underlying probability distribution.

Some Notations. In the remainder of this paper, we will often let $\mathcal{K} \stackrel{\text{def}}{=} \langle K_1, K_2, \ldots, K_N \rangle$ denote the ***static cascade*** – the optimal contained IDK cascade *in the absence of dependencies between successive inputs*, where the classifiers are listed in the order they are to be called, with K_N denoting the deterministic classifier.[2] Such an optimal IDK cascade can be constructed from all available classifiers using algorithms provided by Baruah et al. [4]. We let C_i denote K_i's WCET, and P_i the probability that K_i returns a true class (rather than IDK) upon a randomly-drawn input. (Note that $P_N = 1$, and that $C_{i-1} < C_i$ and $P_{i-1} < P_i$ for each $i, 1 < i \leq N$, in any optimal cascade.) An example cascade for the case $N = 4$ (i.e., constituting three IDK classifiers and one deterministic classifier) is depicted in Figure 2 – this optimal cascade was synthesized from the collection of contained classifiers depicted in Figure 1 (b) – as can be seen, all four classifiers appear in this optimal cascade.

Suppose that during the classification of a particular input stream $\vec{x}$ that has dependence parameter λ, the classifier K_b is the first classifier in the cascade $\mathcal{K}$ to successfully classify the input x_{t-1} (in Section 3.1, we will refer to K_b as the ***boundary classifier*** for x_{t-1}). By definition of the dependence parameter (Definition 1), there is a probability λ that K_b will also be the first classifier to successfully classify x_t due to dependence, and a probability $(1 - \lambda)$ that x_t will be drawn independently from the underlying probability distribution. The probability $\widehat{P}_i$ that each K_i will successfully classify x_t, is therefore as follows:

$$\widehat{P}_i = \begin{cases} (1-\lambda)P_i, & i < b \\ \lambda + (1-\lambda)P_i, & i \geq b \end{cases} \tag{1}$$

Problem Considered. We assume that a stream of inputs arrives (e.g., from a sensor) at the cascade with successive inputs arriving exactly T time units apart. Each input must be classified before the next input arrives: i.e., there is a hard deadline of time-duration T for successful classification. The *performance objective* is to reduce the average duration to successful classification over all the inputs in the input stream, subject to the hard deadline being met for each and every input. Once the classification is obtained for an input, the identified class is immediately reported; *the lapsed duration between the input's arrival instant and this classification instant constitutes the response time for this input.* After the classification is completed, the remainder of the time-interval may be used by the runtime algorithm for performing additional "exploratory" computations that are aimed at reducing the average duration for classification of future inputs.

[2] Note the change in notation from Section 2 – there the subscripts were just for distinguishing amongst n different classifiers, whereas the subscript now denotes the position of the classifier in the optimal cascade (which comprises N classifiers).

3.1 The Current State-of-the-art Classification Algorithm [2]

Agrawal et al. [2] present an algorithm for using the static cascade assuming the value of λ is known[3]. During an INITIALIZATION PHASE prior to processing the input stream $\vec{x}$, this algorithm (i) computes a "***skip factor***" which is a positive integer k, $0 \leq k \leq N$, as a function of the value of the dependence parameter λ characterizing $\vec{x}$; and (ii) designates the classifier K_1 as the boundary classifier for the classification of a hypothetical input x_0. After this initialization phase, the algorithm proceeds to process each input in $\vec{x}$ in order, where classification is followed by an exploration step:

1. CLASSIFICATION. Let b denote the index of the boundary classifier K_b for x_{t-1} (i.e., the earliest classifier is that can successfully classified x_{t-1}). The attempt to classify x_t begins at the classifier $K_{\max(1,b-k)}$, and proceeds down the cascade until a successful classification is obtained.
2. EXPLORATION. Immediately when a successful classification for x_t is obtained, it is reported. The reminder of the interval prior to the arrival of the next input is used to execute additional classifiers in order to determine the boundary classifier for x_t, unless the boundary is already identified during classification.

The rationale for the design of this algorithm is motivated by considering the boundary cases: when $\lambda = 0$ (no dependence) and $\lambda = 1$ (full dependence).

- If $\lambda = 0$, then each input is independent and hence one should attempt to classify the input x_t starting at classifier K_1, working one's way down the cascade until a successful (i.e., non-IDK) classification is obtained. By the optimality of the static cascade in the absence of input dependencies, doing so minimizes the expected classification duration.
- If $\lambda = 1$, then there is no point in attempting to classify x_t with classifiers before K_b – they're all guaranteed to fail since x_t and x_{t-1} are equivalent inputs and thus successfully classified by exactly the same classifiers. Instead, using K_b to classify x_t is optimal.

Thus, for the two extreme values of the dependence parameter, $\lambda = 0$ and $\lambda = 1$, the optimal strategy is to either start at the beginning of the cascade (when $\lambda = 0$), or at the boundary classifier itself (when $\lambda = 1$), respectively. For values of λ that lie between these two extremes (i.e., $0 < \lambda < 1$), they adopt the reasonable strategy of interpolating between these optimal decisions for the two extremes and skipping backwards a constant number of classifiers k (the skip factor) from the boundary classifier, where the value of k depends upon the value of λ: the smaller the value of λ, the larger the value of k.

For stream-classification algorithms based on the skip factor approach (i.e., using cascades that start processing each input a skip factor prior to the boundary classifier for the previous input), their algorithm [2] for computing the skip factor as a function of λ is optimal: for a given value of λ, this algorithm computes the skip factor that will result in minimum average classification time for a time series input stream with dependence parameter λ.

3.2 Can the Skip Factor Approach be Improved?

In effect, the skip factor approach truncates the cascade for the purposes of classifying x_t, by snipping off a segment at the front of the cascade. While the skip factor can be optimally computed, it behooves us to wonder whether one can do even better than the skip factor approach; the following example suggests the answer is 'YES.'

[3] They describe [2] how the value of λ can be *learned* via runtime observations by a direct and straightforward application of the standard learning technique of Maximum Likelihood Estimation (MLE) [13].

▶ **Example 2.** Consider an input stream $\vec{x}$ that is to be classified by the example cascade of Figure 2, with a dependence parameter $\lambda = 0.75$. Let us suppose that the boundary classifier for the input x_{t-1} is K_3. The updated probabilities of successful classification of x_t by the different classifiers, computed according to Equation 1, are

$$\widehat{P_1} = (1-\lambda) \times P_1 = 0.25 \times 0.1 = \mathbf{0.025}$$
$$\widehat{P_2} = (1-\lambda) \times P_2 = 0.25 \times 0.2 = \mathbf{0.050}$$
$$\widehat{P_3} = \lambda + (1-\lambda) P_3 = 0.75 + 0.25 \times 0.9 = \mathbf{0.975}$$
$$\widehat{P_4} = \lambda + (1-\lambda) P_4 = 0.75 + 0.25 \times 1.0 = \mathbf{1}$$

As the skip factor is bounded by the length of the cascade $N = 4$, i.e., $k \in \{0, 1, 2, 3, 4\}$, let us consider every possibility.

1. For $k \in \{2, 3, 4\}$, $\max(1, 3-k) = 1$. Hence the skip factor algorithm would attempt to classify x_t starting at classifier K_1, and the expected duration to successful classification is
$$\begin{aligned} & C_1 + (1-\widehat{P_1}) \times C_2 + (1-\widehat{P_2}) \times C_3 + (1-\widehat{P_3}) \times C_4 \\ = \; & 1 + 0.975 \times 11 + 0.95 \times 100 + 0.025 \times 200 = \mathbf{111.725} \end{aligned}$$
2. For $k = 1$, we have $\max(1, 3-1) = 2$. Hence the skip factor algorithm would attempt to classify x_t starting at classifier K_2 and the expected duration to successful classification is
$$\begin{aligned} & C_2 + (1-\widehat{P_2}) \times C_3 + (1-\widehat{P_3}) \times C_4 \\ = \; & 11 + 0.95 \times 100 + 0.025 \times 200 = \mathbf{111.0} \end{aligned}$$
3. For $k = 0$, $\max(1, 3-0) = 3$. Hence the skip factor algorithm would attempt to classify x_t starting at classifier K_3 itself and the expected duration to successful classification is
$$C_3 + (1-\widehat{P_3}) \times C_4 = 100 + 0.025 \times 200 = \mathbf{105.0}$$

If, however, as an alternative to the skip-factor approach, we were to instead attempt to classify x_t using the classifiers in the order K_1 followed by K_3 and finally K_4 (i.e., we used the cascade $\langle K_1, K_3, K_4 \rangle$), the expected duration to successful classification is

$$C_1 + (1-\widehat{P_1}) \times C_3 + (1-\widehat{P_3}) \times C_4 = 1 + 0.975 \times 100 + 0.025 \times 200 = \mathbf{103.5}$$

which is smaller than the expected duration to successful classification that can possibly be obtained by any skip factor algorithm, regardless of what skip factor it computes. (We reemphasize that $\langle K_1, K_3, K_4 \rangle$ is not a contiguous sub-cascade of $\mathcal{K}$, and so is never considered by the skip factor approach.) ◀

Our Proposed Approach. The skip-factor approach deals with dependence in time-series input streams by skipping upstream from the boundary classifier a certain number of classifiers k, the precise value of k depending upon the degree of dependence. Our *conceptual realization*, illustrated in Example 2, is that this approach can be improved upon by additionally skipping some of the classifiers within the cascade. This realization immediately suggests the following runtime strategy for classifying the inputs in input stream $\vec{x}$ with dependence parameter λ: once x_{t-1} has been classified and the boundary classifier K_b identified, we

1. update the probability values – i.e., compute the $\widehat{P_i}$ values as dictated by Equation 1;
2. synthesize a *new* cascade using these $\widehat{P_i}$ values and the classifiers that are present in the cascade $\mathcal{K}$, that minimizes the expected duration to successful classification;
3. use this new cascade to classify the input x_t; and
4. perform exploration, if necessary, to identify the new boundary classifier.

Note that the new cascade synthesized in Step 2 above will only be used for classifying the input x_t (after which the process will be repeated – a new boundary classifier identified, the $\widehat{P}_i$ values recomputed according to Equation 1, and a new cascade synthesized for classifying x_{t+1}). In other words, this cascade is only used for classifying a single input that is drawn at random from the distribution $\{\widehat{P}_i\}$; therefore, the algorithms proposed by Baruah et al. [4] are optimal for this purpose.

However, these cascade-synthesis algorithms have running time $\Theta(n^2)$ where n is the number of available IDK cascades when no hard deadline is specified, and $\Theta(n^2 D)$ if a hard deadline D must be guaranteed, and are therefore likely to be too inefficient for repeated use prior to classifying each input in the input stream. For the strategy outlined above to be applicable in practice, we need faster cascade-synthesis algorithms. In Section 4 below we meet this need and improve upon the $\Theta(n^2)$ algorithm: we derive a linear-time (i.e., $\Theta(n)$ time) algorithm for synthesizing optimal cascades, which is efficient enough for runtime use as envisioned above. Then in Section 5, we flesh out the details of the strategy outlined above, and describe in detail the runtime algorithm that is used for classifying all the inputs in a time-series input stream in a manner that further reduces (as experimentally demonstrated in Section 6) the average duration to successful classification.

4 A Linear-Time Cascade-Synthesis Algorithm

Example 2 revealed that in classifying the inputs in input stream $\vec{x}$, one can sometimes achieve a reduction in expected duration to successful classification by choosing to use a subset of the classifiers in the cascade that are *not* contiguous in the cascade upon inputs – in the example, we saw that $\langle K_1, K_3, K_4 \rangle$ has smaller expected duration to successful classification than any contiguous subset. But which subset of the classifiers in the cascade should one use? While this question is easily solved via the algorithms proposed by Baruah et al. [4], those algorithms are computationally expensive (with $\Theta(n^2)$ runtime complexity) and may not be practical to execute between each pair of input arrivals during stream classification time. In this section, we derive an entirely different cascade synthesis algorithm that has linear (i.e., $\Theta(n)$) runtime complexity; for the values of n arising in practice, this algorithm is fast enough for runtime use during classification of time-series input streams.

4.1 The Current State-of-the-art Cascade-Synthesis Algorithm [4]

Let us first examine the state of the art. Given a contained collection of IDK classifiers $K_1, K_2, \ldots, K_{n-1}$ and a deterministic classifier K_n, an algorithm with running time $\Theta(n^2)$ was derived [4] for synthesizing a cascade that minimizes the expected duration to successful classification for an input that is randomly drawn from the underlying distribution. Without loss of generality, assume that the classifiers are indexed in increasing order of probability of successful classification: $P_i < P_{i+1}$ (and hence $C_i < C_{i+1}$ – if not, we can ignore K_i since it will definitely not appear in an optimal cascade). Note that for the deterministic classifier K_n, we have $P_n = 1.0$. For notational convenience, let us define a hypothetical dummy classifier K_0 with $C_0 = 0$ and $P_0 = 0.0$.

The algorithm proposed by Baruah et al. [4] is a classical dynamic program: it successively determines the best cascade that can be built using $K_1, K_2, \ldots, K_i$ for $i = 1, 2, \ldots, n$, where the notion of "best" is formalized in the following definition:

▶ **Definition 3** (optimal sub-sequence S_i; optimal expected duration f_i)**.** *The optimal sub-sequence S_i is the cascade comprising classifiers from $\langle K_1, K_2, \ldots, K_i \rangle$ of minimum expected duration, with K_i being the last classifier in the cascade. Let f_i denote the expected duration of this optimal sub-sequence S_i.*

Algorithm 1 The cascade-synthesis algorithm proposed by Baruah et al. [4].

Input: $K_1, K_2, \ldots, K_n$, with each $K_i = (C_i, P_i)$
Output: The expected duration f_n of the optimal cascade S_n

```
1 for i ← 1 to n do                                          //Computing f_i
2     tmp ← 0
3     for h ← 1 to i − 1 do                  //Would K_h be a better predecessor to K_i?
4         if (f_h + (1 − P_h) × C_i) < (f_tmp + (1 − P_tmp) × C_i) then
5             tmp ← h
6         f_i ← f_tmp + (1 − P_tmp) × C_i
7 return f_n
```

Using the terminology introduced in Definition 3, the aim of determining an optimal IDK cascade reduces to that of determining the optimal sub-sequence S_n. The $\Theta(n^2)$ algorithm [4] achieves this by inductively determining the optimal sub-sequences $S_1, S_2, \ldots, S_n$ in order. To determine S_i for $i \geq 1$, they exploit the fact that optimal sub-sequences satisfy the *optimal sub-structure property* [7, page 379]: optimal solutions to any problem instance incorporate optimal solutions to sub-instances. Initially, we have:

$$S_0 = \langle\ \rangle \text{ and } f_0 = 0$$

Let K_h denote the classifier immediately preceding K_i in the optimal sub-sequence S_i. By the optimal sub-structure property, it follows that S_i is the concatenation of K_i to the end of the optimal sub-sequence S_h (i.e., $S_i = (S_h||K_i)$). We therefore have:

$$f_i = \min_{0 \leq h < i} \{f_h + (1 - P_h) \cdot C_i\} \tag{2}$$

and S_i is the concatenation of K_i to the end of S_ℓ where $\ell = \arg\min_{0 \leq h < i} \{f_h + (1 - P_h) \cdot C_i\}$. The $\Theta(n^2)$ algorithm, which we depict in pseudocode form in Algorithm 1, computes f_i according to Equation 2 in increasing order of i; in computing f_i, it iterates through $h = 0, 1, \ldots, i-1$, and is thus a pair of nested for-loops with overall running time $\sum_{i=1}^{n} i = \boldsymbol{\Theta(n^2)}$.

4.2 Our New And Improved Algorithm

In this section, we derive a $\Theta(n)$ algorithm for computing f_n (and thereby S_n). We do so by computing $f_1, f_2, \ldots, f_i$ in increasing order of i according to Algorithm 2. In determining f_i for a particular i we do not, however, simply iterate through all potential values of h. Rather, we reimplement the work done by the inner loop of the pseudocode of Algorithm 1 in such a manner that each value of 'h' in this inner loop is only considered a constant number of times throughout all n iterations of the outer loop (and hence the total running time of the algorithm falls from $\Theta(n^2)$ to $\Theta(n)$). To understand how we achieve this, let us take a closer look at the manner in which the quantities on the RHS of Equation 2, over which the 'min' is being determined, relate to each other.

▶ **Lemma 4.** *Let integers j_1 and j_2 satisfy $0 \leq j_1 < j_2 < n$. When plotted as functions of x, the lines $y_{j_1}(x) = (f_{j_1} + (1 - P_{j_1}) \cdot x)$ and $y_{j_2}(x) = (f_{j_2} + (1 - P_{j_2}) \cdot x)$ intersect. Let x_2 be the intersection, i.e., the solution of $f_{j_1} + (1 - P_{j_1}) \cdot x = f_{j_2} + (1 - P_{j_2}) \cdot x$, then*

$$\min\{y_{j_1}(x), y_{j_2}(x)\} = \begin{cases} f_{j_1} + (1 - P_{j_1}) \cdot x, & \text{if } x \in [0, x_2] \\ f_{j_2} + (1 - P_{j_2}) \cdot x, & \text{else } x \in [x_2, \infty)) \end{cases}$$

(Please see Figure 3 for visual examples.)

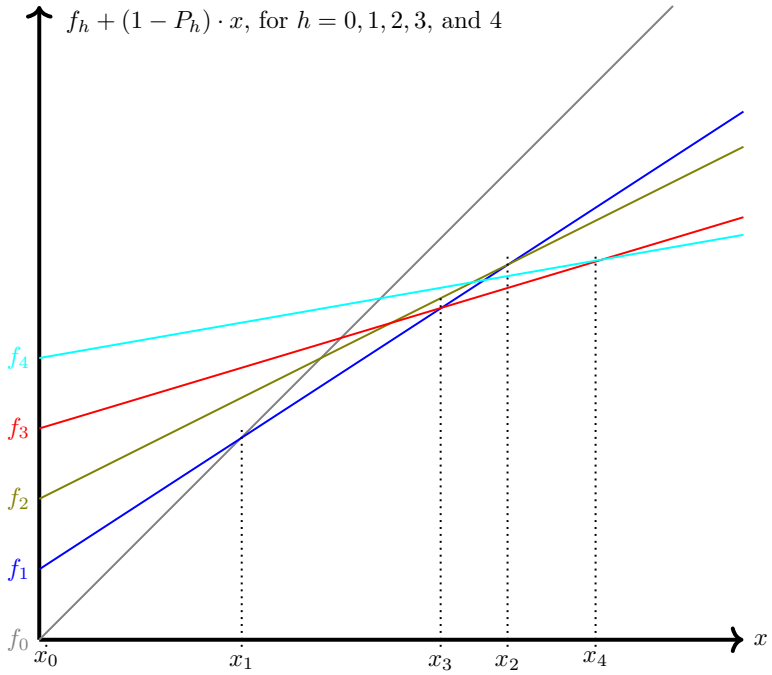


Figure 3 Plots of $(f_h + (1 - P_h) \times x)$ as a function of the value of x, for $h = 0, 1, 2, 3$, and 4. The value of x_k denotes the value of x after which the plot $(f_k + (1 - P_k) \times x)$ is below all prior plots (i.e., all plots $(f_h + (1 - P_h) \times x)$ for $h < k$).

Proof. We make the following observations.

1. $f_{j_1} < f_{j_2}$.
   To show this, we will show that $f_{j+1} > f_j$ for all j. This follows since by definition,
   $$f_j = \min_{h<j} \{f_h + (1 - P_h) \cdot C_j\}$$
   $$\text{and } f_{j+1} = \min_{h \le j} \{f_h + (1 - P_h) \cdot C_{j+1}\}$$
   Since $C_{j+1} > C_j$, for each $h < j$ we have
   $$f_h + (1 - P_h) \cdot C_{j+1} > f_h + (1 - P_h) \cdot C_i$$
   That is, each of the first $(j-1)$ terms in the RHS for f_{j+1} is larger than the corresponding term in f_j. It remains to consider the last term in f_{j+1}: $f_j + (1 - P_j) \cdot C_{j+1}$. Since this equals f_j plus a positive term, it, too, is clearly $> f_j$.
2. $(1 - P_{j_1}) > (1 - P_{j_2})$ (since $P_{j_2} > P_{j_1}$).

From the above, it follows that the plot for $(f_{j_1} + (1 - P_{j_1}) \times x)$ has a smaller y-intercept and larger slope than the plot for $(f_{j_2} + (1 - P_{j_2}) \times x)$; consequently, they intersect at x_2. When $x \le x_2$, $(f_{j_1} + (1 - P_{j_1}) \times x)$ is smaller; when $x > x_2$, $(f_{j_2} + (1 - P_{j_2}) \times x)$ is smaller. ◀

The significance of Lemma 4 is highlighted in the following example.

▶ **Example 5.** Let us suppose that we have completed computing f_0 (which equals 0), f_1 (which equals C_1), f_2, f_3, and f_4 on a particular example instance (as well as the corresponding optimal cascades $S_0 = \langle\,\rangle$, $S_1 = \langle K_1 \rangle$, S_2, S_3, and S_4), and seek to determine the optimal cascade S_5 ending in K_5 and its expected duration f_5. From Lemma 4, we can obtain a graph like the one depicted in Figure 3. For any value of C_5 (plotted on the x axis), this graph reveals the value of $h < 5$ that minimizes $\big(f_h + (1 - P_h) \times C_5\big)$. The $x_0 (= 0), x_1, x_2, x_3$

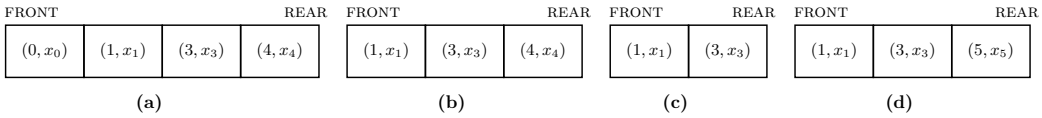


Figure 4 The dequeue of Example 5 at various stages.

and x_4 values depicted on the x-axis are significant: each x_k denotes the value of x after which the plot $\big(f_k + (1 - P_k) \times x\big)$ is below all prior plots (and is hence potentially the one that determines the 'min' for the RHS of Equation 2). Our algorithm stores each such x_k as an ordered pair (k, x_k). From this graph (and recalling that S_0, the optimal cascade ending in the hypothetical dummy classifier K_0, is the empty cascade $\langle\ \rangle$), we can conclude that

$$S_5 = \begin{cases} (S_0||K_5) = (\langle\ \rangle||K_5) = \langle K_5\rangle, & \text{if } C_5 \in [x_0, x_1) \\ (S_1||K_5), & \text{if } C_5 \in [x_1, x_3) \\ (S_3||K_5), & \text{if } C_5 \in [x_3, x_4) \\ (S_4||K_5), & \text{otherwise, (i.e., } C_5 \in [x_4, \infty)) \end{cases}$$

We make two observations regarding this example:

1. x_2 has no role to play in deciding S_5 – its role is subsumed by x_3 (since $x_3 < x_2$, meaning that the line $(f_3 - (1 - P_3) \times x)$ lies below the line $(f_2 - (1 - P_2) \times x)$ for $x \geq x_3$).
2. Since $C_5 > C_4$, only the part of the graph to the right of $(x = C_4)$ is meaningful in determining S_5. (Suppose, for instance, it had been the case that $x_1 \leq C_4 < x_3$. Then x_0 and x_1 have no role to play in deciding S_5, either; all that matters is whether $C_5 \leq x_3$, $x_3 \leq C_5 < x_4$, or $x_4 \leq C_5$.)

Based on these observations, we point out that the ordered list $[(0, 0), (1, x_1), (3, x_3), (4, x_4)]$ of four x_k values (each represented as an ordered pair as mentioned above) contains all the information that is needed to compute S_5 and f_5; if it had additionally been the case that $x_3 \leq C_4 < x_4$, then only the two ordered pairs $[(3, x_3), (4, x_4)]$ would be the ones to matter. Our algorithm will maintain such an ordered list of ordered pairs sorted in increasing order of x_k, from which S_5 and f_5 is computed in the following manner:

1. Determine the contiguous pair of ordered pairs (ℓ, x_ℓ) and $(\ell', x_{\ell'})$ in the list satisfying $x_\ell \leq C_5 < x_{\ell'}$. (If C_5 is larger than the x_k for the last ordered pair in the list, then let (ℓ, C_ℓ) denote this last ordered pair).
2. As can be seen from Figure 3, this ℓ is the value of h that minimizes the RHS of Equation 2 for computing f_5.
3. Therefore, $S_5 = (S_\ell||K_5)$ and $f_5 = f_\ell + (1 - P_\ell) \cdot C_5$.

The main innovation in our algorithm is that it is able to *use and update this list in an efficient* manner. It does so by maintaining it as a double-ended queue (deque) – see Figure 4 (a). Given the deque of Figure 4 (a) and supposing that the value of C_5 is $> x_1$ and $< x_3$, we now step through the steps taken by our algorithm to compute S_5 and f_5 and update the deque.

1. Our algorithm starts out comparing C_5 with x_1, the x_k value of the second entry of the deque. Since $C_5 > x_1$, the algorithm concludes that all future C_i's (i.e., all C_i's for $i > 5$) will also be $> x_1$ and hence the first entry in the deque, $(0, 0)$, is no longer relevant; this element is therefore removed (a $\Theta(1)$ operation), yielding the deque of Figure 4 (b).
2. The step above is repeated: C_5 is compared with x_3, the x_k value of the new second entry of the deque. Since $C_5 < x_3$, we conclude that in computing f_5 using Equation 2, the first ordered pair in the dequeue, $(1, x_1)$, determines the 'min' value in the RHS of Equation 2.
3. f_5 is therefore computed as per Equation 2: $f_5 = f_1 + (1 - P_1) \times C_5$.

Algorithm 2 `OptBuild` – an optimal and efficient cascade-synthesis algorithm.

```
1  OptBuild(C_1, C_2, ..., C_n;  P_1, P_2, ..., P_n)
   Input: The execution durations C_1, C_2, ..., C_n and probabilities P_1, P_2, ..., P_n
   Output: The optimal cascade S_n and its expected duration f_n
2  Q = [(0,0)] Prev[0] = ⊥; f_0 = 0
3  for i ← 1 to n do
4      while ((len(Q) > 1) ∧ (Q[2][2] ≤ C_i)) do //Q[2] is the second-from-front ordered
         pair in Q; Q[2][2] is its x_k component
5          Q.pop_front()          //Not needed any more, since C_j > C_i for all j > i
6      Let (h, x_h) ← Q.front()
7      Prev[i] ← h                                                    // S_i = (S_h||K_i)
8      f_i ← f_h + (1 − P_h) · C_i
9      repeat
10         Let (h, x_h) ← Q.back()
11         Compute x_i such that f_h + (1 − P_h) · x_i = f_i + (1 − P_i) · x_i   //Where the lines
             for h and i intersect
12         if x_i ≤ x_h then
13             Q.pop_rear()      //(h, x_h) is subsumed by (i, x_i) (which will be added -
                 Line 15 below)
14     until (x_i > x_h);
15     Q.push_rear((i, x_i))
16 return the optimal cascade and f_n
```

4. Conceptually speaking, a new line with y-offset equal to f_5 and slope $(1 - P_5)$ is now added to Figure 3 for the purposes of computing $S_6, S_7, \ldots, S_n$. Hence the point x_5 must be determined, beyond which the plot $(f_5 + (1 - P_5) \times x)$ is beneath all prior plots.
5. Recall how the fact that $x_3 < x_2$ meant that $(f_2 + (1 - P_2) \times x)$ could never be the line defining the 'min' for f_5. We wish to identify any of the x_k's in the deque that are rendered similarly incapable, due to being smaller than x_5, of defining the 'min' for values of f_i for $i > 5$. Starting with the ordered pair at the rear of the deque (in our case, this is $(4, x_4)$), our algorithm determines the value of x for which the new line $(f_5 + (1 - P_5) \times x)$ intersects with the line $(f_4 + (1 - P_4) \times x)$. Suppose this value of x is $< x_4$. This implies that $x_5 < x_4$, and x_4 will consequently never define the 'min' on the RHS of Equation 2 when computing f_i for future values of i that are > 5. Hence the ordered pair $(4, x_4)$ is deleted from the rear of the deque – again, a $\Theta(1)$ operation. This results in the deque of Figure 4 (c).
6. The process is now repeated on the new ordered pair at the back of the deque, $(3, x_3)$: determine the value of x for which $(f_5 + (1 - P_5) \times x)$ intersects with the line $(f_3 + (1 - P_3) \times x)$. Let us suppose that this intersection occurs to the right of x_3 and is thus equal to x_5 (which recall, is defined as the value of x after which the plot $(f_5 + (1 - P_5) \times x)$ is below all the earlier plots). The ordered pair $(3, x_3)$ remains relevant and cannot be removed from the deque.

   Once we have identified an ordered pair at the end of the deque that should not be removed, since the deque is sorted in increasing order of the x_k parameters it follows that any ordered pairs that were present before this last ordered pair in the deque, too, remain relevant and hence should not be deleted.
7. Finally, our algorithm adds the ordered pair $(5, x_5)$ to the rear of the deque in a $\Theta(1)$ operation, and terminates. The resulting deque is as depicted in Figure 4 (d). ◀

Algorithm 2 lists the pseudocode that implements the algorithm illustrated in Example 5 above. We now formally prove the correctness of Algorithm 2, based on the following lemma.

▶ **Lemma 6.** *Let (q_j, x_j) be the j-th element of Q (i.e., $(q_j, x_j) \leftarrow Q[j]$). Let $y_h(x) = f_h + (1 - P_h) \cdot x$ be functions of x, where integer $h \in [0, n]$. Let $x_{j+1} = \infty$, if (q_j, x_j) is the last element of Q. At the beginning of each iteration i of Algorithm 2, Q has the property:*

$\min_{h\in[0,i)} y_h(x) = y_{q_j}(x) = f_{q_j} + (1 - P_{q_j}) \cdot x$, *for* $x_j \le x \le x_{j+1}$,

which implies that $f_{q_j} + (1 - P_{q_j}) \cdot x_{j+1} = f_{q_{j+1}} + (1 - P_{q_{j+1}}) \cdot x_{j+1}$

Proof. We prove by induction. When $i = 1$, the queue only has one element, so it is true.

Now suppose iteration i satisfies this property of $\min_{h\in[0,i)} y_h(x) = y_{q_j}(x)$ for $x_j \le x \le x_{j+1}$, where $Q = \{(q_1, x_1), \ldots, (q_l, x_l)\}$ with l elements, we only need to prove it for iteration $i + 1$. In other words, we need to prove that by the end of iteration i where Q is updated to $Q' = \{(q'_1, x'_1), \ldots, (q'_k, x'_k)\}$ with k elements, the property of $\min_{h\in[0,i]} y_h(x) = y_{q'_j}(x)$ for $x'_j \le x \le x'_{j+1}$ holds.

Now consider the property by looking at the updates to Q during the loop body of iteration i. Clearly, removing the front elements from Q (in lines 4-5) or the rear elements (in lines 9-14) does not affect the property of $\min_{h\in[0,i)} y_h(x) = y_{q'_j}(x)$ for $x'_j \le x \le x'_{j+1}$ where (q'_j, x'_j) is the element remained in Q.

However, we need to show that adding the new function $y_i(x)$ in iteration i does not change the property for the elements remained in Q. In other words, $\min_{h\in[0,i]} y_h(x) = \min_{h\in[0,i)} y_h(x) = y_{q'_j}(x)$ for $x'_j \le x \le x'_{j+1}$ where $j \in [1, k)$. Additionally, we need to analyze whether the added element satisfies the property.

Line 15 shows that (i, x_i) is the only element added (to the rear of updated Q) in iteration i. Therefore, we know $q'_k = i$ and $x'_k = x_i$ and (q'_{k-1}, x'_{k-1}) is the last element remained in the updated Q before adding $(q'_k, x'_k) = (i, x_i)$. From line 11 with $h = q'_{k-1}$, we have

$$f_{q'_{k-1}} + (1 - P_{q'_{k-1}})x_i = f_h + (1 - P_h)x_i = f_i + (1 - P_i)x_i = f_{q'_k} + (1 - P_{q'_k})x_i$$

From Lemma 4, we have

$$\min\left\{y_{q'_{k-1}}(x), y_{q'_k}(x)\right\} = \begin{cases} f_{q'_{k-1}} + (1 - P_{q'_{k-1}})x, & \text{if } x \in [0, x_i] \\ f_{q'_k} + (1 - P_{q'_k})x, & \text{else } x \in [x_i, \infty) \end{cases}$$

Therefore, for $x'_{k-1} \le x < x'_k = x_i$ and $y_{q'_k} = y_i(x)$, we have $\min_{h\in[0,i)} y_h(x) = y_{q'_{k-1}}(x) < y_i(x)$. Thus, the property $\min_{h\in[0,i]} y_h(x) = y_{q'_{k-1}}(x)$ holds for $x'_{k-1} \le x < x'_k$.

Similarly, for any element (q'_j, x'_j) before (q'_{k-1}, x'_{k-1}) with $x'_j \le x \le x'_{j+1}$, we have $\min_{h\in[0,i)} y_h(x) = y_{q'_j}(x) < y_{q'_{k-1}}(x) < y_i(x)$. Thus, the property $\min_{h\in[0,i]} y_h(x) = y_{q'_j}(x)$ holds for $x'_j \le x < x'_{j+1}$. In summary, we have proved that the property holds for any elements remained in the updated Q' after adding the new function $y_i(x)$.

Now, we consider the property of the added element $(q'_k, x'_k) = (i, x_i)$ with $x_i \le x < \infty$. Since $\min\left\{y_{q'_{k-1}}(x), y_{q'_k}(x)\right\} = y_{q'_k} = y_i(x)$ for $x \in [x_i, \infty)$, we have $\min_{h\in[0,i]} y_h(x) = y_{q'_k}$. Thus, the new element also satisfy the property. This completes the induction. ◀

Finally, we discuss the time complexity of the algorithm.

▶ **Lemma 7.** *Algorithm 2 is $\mathcal{O}(n)$ in time.*

Proof. *Sketch:* First, observe that for each $i, 1 \le i \le n$ of the **for**, an ordered pair (i, x_i) is pushed into Q exactly once. Second, the nested loops runs for at most $(n + 1)$ times in total. This is because each iteration of the two **while** loops (lines 4-5 and 9-14) contains one *pop* operation from Q. Moreover, the popped elements never come back, which means there can be at most $n + 1$ *pop* operations. This limits the total number of nested iterations, despite contained in the **for** loop. ◀

Algorithm 3 Runtime classification algorithm.

Input: A cascade $\langle K_1, K_2, \ldots, K_N \rangle$ synthesized by the algorithm of [4]
Output: A classification for each input in time-series input stream $\vec{x}$

```
1 b ← 1                                          //Initialize the boundary classifier as K_1
2 for each input x_t do
3     for i ← 1 to N do
4         Determine P̂_i according to Equation 1
5     Let cascade S ← OptBuild(C_1, C_2, ..., C_N; P̂_1, P̂_2, ..., P̂_N)   //OptBuild is defined
        in Algorithm 2
6     Attempt to classify x_t with the classifiers in S in order, until a non-IDK class is obtained
7     return this class
8     Perform exploration to identify the new boundary classifier K_b
9     Update the value of b
```

5 Runtime Classification

When classifying the inputs in a time-series input stream $\vec{x}$ that is characterized by a dependence parameter λ, our linear-time cascade-synthesis algorithm of Section 4 permits us to rebuild a cascade for each input in a time-series input stream, thereby achieving lower average duration to successful classification when compared to the state-of-the-art algorithm in [2]. Our runtime algorithm for doing this is listed in pseudocode form in Algorithm 3. We emphasize a few points about this algorithm.

1. The initial cascade (that forms the input to Algorithm 3) is synthesized by the algorithm in [4], which does not make use of the dependence parameter λ that is specified for $\vec{x}$. This algorithm has running time $\Theta(n^2 D)$ where n is the number of IDK cascades available to us, and D the hard deadline specifying the duration within which each input must be classified (this is the inter-arrival duration T between successive inputs).
2. However in the calls to `OptBuild( )` in Line 5, only those classifiers are considered for potential inclusion in the cascade, that were already in the cascade that was synthesized prior to runtime by the algorithm in [4]. Since the entire cascade synthesized prior to runtime by the algorithm in [4] can execute to complete within the specified hard deadline, the cascade S returned by `OptBuild( )` in Line 5 is guaranteed to also meet the deadline.
3. During the exploration phase (Line 8), all N classifiers that were in the cascade that was synthesized prior to runtime by the algorithm in [4] should be considered, not just those that were included in the cascade S that was obtained in Line 5. This is because the boundary classifier is defined over all the classifiers in the cascade that was synthesized prior to runtime by the algorithm in [4]; it is possible that some classifier present in the cascade synthesized prior to runtime by the algorithm in [4] but not included in S is the one with minimum execution duration to successfully classify x_t (and thus the boundary classifier).

5.1 Generalizing the Definition of Temporal Independence

In Definition 1, Agrawal et al. [2] use a single dependence parameter λ to characterize the dependence of a time-series input stream $\vec{x}$ with respect to an IDK classifier cascade $\mathcal{K}$ that is tasked with classifying the inputs in $\vec{x}$. But looking upon the dependence parameter as an attribute of just the input stream $\vec{x}$ (and not also the classifier cascade $\mathcal{K}$) may be a more accurate reflection of reality: inter-input dependences characterizes an input stream, and

each IDK classifier may be differently impacted by this intrinsic inter-input dependence. We therefore also consider a more general model than the one in [2], in which each IDK classifier K_k is characterized by its own dependence parameter λ_k (with the same interpretation as in Definition 1: if classifier K_k is the classifier of minimum WCET to successfully classify x_{t-1}, then there is a λ_k probability that it will also be the classifier of minimum WCET to successfully classify x_{t-1}, while with probability $(1-\lambda_k)$ input x_t is drawn at random from the underlying probability distribution).[4] Our classification algorithm of Algorithm 3 is easily modified to be applicable to this more general and realistic model of inter-input dependence. Equation 1 clearly generalizes as in Equation 3 below, with λ_b playing the role that λ had in Equation 1:

$$\widehat{P}_i = \begin{cases} (1-\lambda_b)P_i, & i < b \\ \lambda_b + (1-\lambda_b)P_i, & i \geq b \end{cases} \tag{3}$$

In the pseudocode of Algorithm 3, the use of Equation 1 in Line 4 should be replaced with Equation 3; everything else remains as is.

Determining the λ_k values. The MLE-based learning techniques that were proposed in [2] (see footnote 3) for learning the value of the single parameter λ based on observing the input stream, remain applicable for learning the values of the multiple parameters $\lambda_1, \lambda_2, \ldots, \lambda_N$.

6 Experimental Evaluation

In this section we discuss some experiments that we have conducted in order to characterize (i) the *efficiency* of the new cascade-synthesis algorithm we had obtained in Section 4; and (ii) the *effectiveness* (i.e., the improved performance in terms of reduced average execution duration to successful classification over the state-of-the-art approach) of our proposed runtime algorithm. All the code used in these simulation experiments is open source and available online[5]. Exact reproducibility (with the exception of overhead evaluation that is, by its very nature, subject to variation on different runs) is ensured via the use of constant seeds for the random number generators.

Workload generation. We have developed a synthetic workload generator that allows us to experimentally explore various parts of the space of possible parameter values that characterize contained collections of IDK classifiers.[6] The workloads generated for the experiments described in the remainder of this section are generated with the following parameter values:

- The WCET parameters (C_i's) of individual classifiers are integers each drawn uniformly at random over $[0, 100]$.

[4] To appreciate why this model of different λ_k values for different IDK classifiers is more realistic, recall that the Venn diagram representation of the probability space for contained collections of classifiers is a set of concentric ellipses (as in, e.g., Figure 1 (b)). A classifier K_b being the boundary classifier for a particular input x_{t-1} implies that K_b returned a real class but the ellipse immediately within K_b's ellipse that is in the cascade returned IDK on x_{t-1}. While one hesitates to put too much physical interpretation to the actual areas in the probability space, it would not be unreasonable to expect that the characteristics of this space (e.g., the probability measure in the region between the ellipses) may have some role to play in deciding whether K_b will also be the boundary classifier for the next input x_t.

[5] `https://doi.org/10.5281/zenodo.15429421`

[6] We emphasize that the goal here is not one of generating workloads that are particularly faithful to real-world use cases, but to allow us to explore the space of possible system instances by varying different parameters.

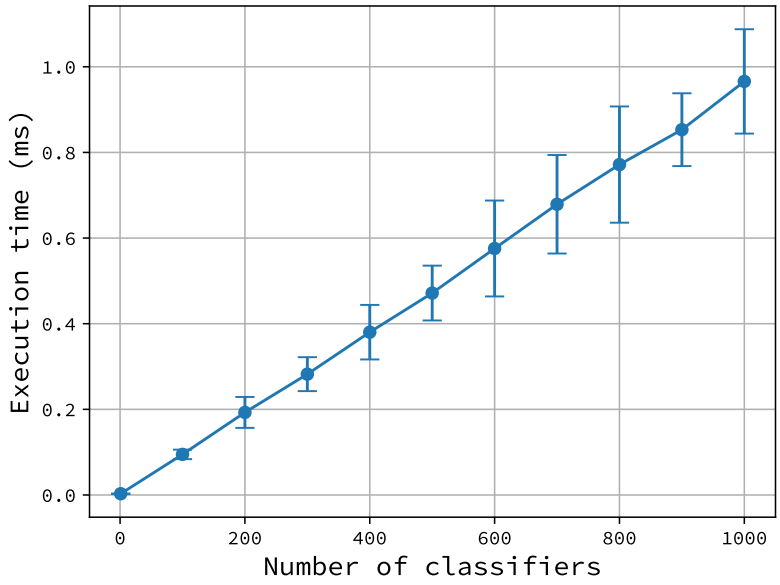


Figure 5 Overhead evaluation of the optimized algorithm.

- The probability P_i that classifier K_i will be successful in classifying a single input drawn independently at random from the underlying probability distribution is a real number drawn uniformly at random over $[0, 1]$.
- For the efficiency experiments, the number of classifiers in the contained collections of IDK classifiers from which the optimal cascade is constructed took on the values $[1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]$; for the effectiveness experiments, the number of classifiers N in the cascade is an integer drawn uniformly at random over $[1, 10]$.

6.1 Evaluating Efficiency of the Cascade-Synthesis Algorithm of Section 4

We performed overhead evaluation experiments to verify the runtime efficiency of the cascade-synthesis algorithm that was derived in Section 4. We separately considered the number of classifiers in the contained collections of IDK classifier from which the optimal cascade is constructed, to be $[1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]$; for each number of classifiers, we generated 10 000 random contained collections of IDK classifiers upon which we used the cascade-synthesis algorithm of Section 4 to synthesize an optimal cascade. The computing platform was a desktop computer equipped with an Intel i7-6700K. (We emphasize that the goal here was not to measure and report the exact absolute execution times but rather to verify the linearity of the runtime complexity with reasonable constants.) As evident from Figure 5, the runtime does indeed scale *linearly* with the number of classifiers in the cascade, and the cascade-synthesis algorithm completes in less than 1 ms to execute for up to 1000 classifiers.

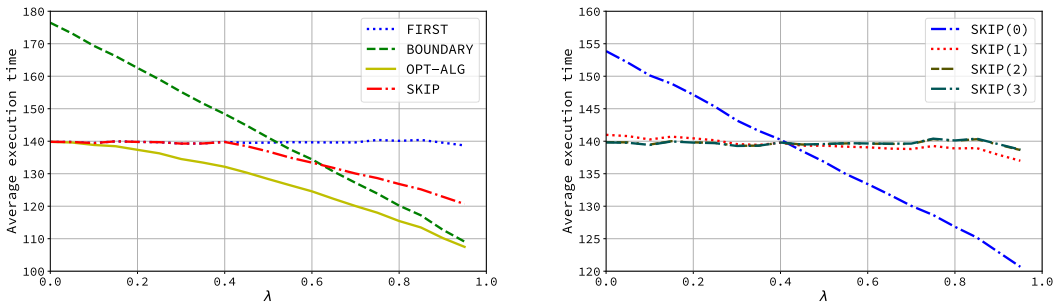


Figure 6 Comparisons of average performance (i.e., expected classification time of input using IDK cascades) under a single λ setting. [left:] comparison of our approach with the state-of-the-art algorithms and [right:] Performances of `SKIP` under various skip factors.

6.2 Evaluating Effectiveness of the Runtime Classification Algorithm

We compare the effectiveness of the runtime classification algorithm (Algorithm 3) versus prior algorithms for both the case when temporal dependence is characterized by a single λ parameter (Section 6.2.1), and when it is characterized by a potentially different λ_i parameter per IDK classifier K_i in the cascade (Section 6.2.2). In either case, we compared the average classification duration of the classification algorithm of Algorithm 3 with the average classification duration of several other classification algorithms:

- `FIRST`: Always start from the first classifier of the offline-optimized cascade. As discussed in Section 3.1, this strategy has no "sense" of inter-input dependence (λ), but is optimal upon time-series input streams with no inter-input dependence ($\lambda = 0$), where each input is drawn independently from the underlying probability distribution.
- `BOUNDARY`: Always start from the boundary classifier K_b for the previous input. It is obvious (and also explained in Section 3.1) that this strategy is optimal upon time-series input streams exhibiting full dependence ($\lambda = 1$) since consecutive inputs will share the exact same boundary classifier.
- `SKIP`: The state-of-the-art "skip-factor" algorithm [2, Algorithm 2] that we have described in Section 3.1. In our experiments, we give this algorithm an advantage by assuming that it always uses the optimal skip factor – the one that minimizes the average duration to successful classification – although in practice there is some loss of performance due to the fact that learning the value of λ takes some time, and may not be entirely accurate.

In the remainder of this section, when discussing our findings we will refer to the runtime classification algorithm of Algorithm 3 as `OPT-ALG`.

6.2.1 Single lambda for all classifiers

We run an exploration of the λ parameter in the interval $[0, 1)$ with a 0.05 step. For each lambda, 10 000 scenarios are created randomly picking values for the number of classifiers ($[1, 10]$), C_i (range $[0, 100]$), and P_i (range $[0, 1]$); and each scenario is run for 1000 inputs. For `FIRST` and `SKIP`, we use the algorithm of [4] to first synthesize an optimal cascade, while for `OPT-ALG`, the optimal cascade is re-synthesized using Algorithm `OptBuild` (Algorithm 2) after each input is classified and the new boundary classifier identified. Our findings are depicted in Figure 6.

The results show that our approach outperforms the basic approaches (`FIRST`, `BOUNDARY`) and the state-of-the-art algorithm (`SKIP`) for all values of $\lambda \in [0, 1)$. As expected, for values of λ towards 1, `OPT-ALG` performs similarly to `BOUNDARY`, because picking the same classifier

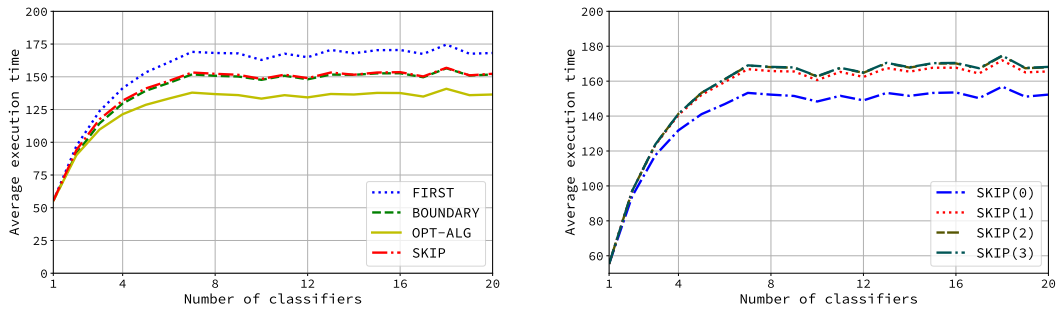


Figure 7 Expected classification time comparisons under a multiple λ_i setting; [left:] comparison of our approach with with the state-of-the-art algorithms and [right:] Performances of `SKIP` under various skip factors.

that classified the previous input is frequently the best choice. Similarly when λ is small, `OPT-ALG` performs similarly to `FIRST`, because starting from the first classifier becomes the best choice. When λ is unknown, the proposed `OPT-ALG` has clear advantages over any other approaches. In particular, the reduction in the execution time of `OPT-ALG` is $0 - 22.5\%$ compared to `FIRST` (average 8.8%), $1.5 - 20.8\%$ compared to `BOUNDARY` (average 10.1%), and $0.1 - 10.9\%$ compared to `SKIP` (average 5.3%).

For the prior state-of-the-art algorithm (`SKIP`), the right subfigure of Figure 6 is reporting its performance under various skip factors. Since the best choice of skip factor may depend on λ (and is unknown), we only report its best-seen performance for each λ (which is a lower envelope of all lines in the right subfigure) in the left subfigure as the performance of `SKIP` to ensure a reasonably fair comparison over [2, Algorithm 2].

Also, note that when λ is large, not only the proposed `OPT-ALG` but also simple heuristics such as `BOUNDARY` is outperforming the state-of-the-art approach `SKIP`. Specifically, the only difference between `BOUNDARY` and `SKIP(0)` is whether to "optimize" the cascade *a priori*. It turns out that such offline optimization without any knowledge/modeling of λ may be hurting the average performance, at least for simple heuristics. This also verifies `OPT-ALG`'s advantage in the linear-time optimization – after each round, we can "afford" to optimize the cascade dynamically according to the current best possible estimations of distributions and the λ value.

Note that standard deviation is not displayed in these plots because it is intrinsically high: the number of classifiers and the WCET interval are wide to test very different scenarios, which also brings highly-spread final execution times, thus the implications are unclear by comparing the standard deviations. For reference, we observed that the standard deviation of `OPT-ALG` was lower than the standard deviation of `FIRST`, `BOUNDARY`, and `SKIP` for all the respective experiments. This suggests `OPT-ALG` is a more *robust* and *stable* algorithm overall.

6.2.2 Multiple lambda case

In this scenario, we run the simulation by using multiple random values of λ_i, one for each classifier of the cascade. We explored how the average execution time changes when the number of classifiers increases. The other parameters of the simulation are identical to the case in the previous Section 6.2.1.

The results are displayed in Figure 7. Also in the multiple lambda case, `OPT-ALG` outperforms all the other approaches. In particular, the reduction in the execution time of `OPT-ALG` is up to 19.7% compared to `FIRST` (average 16.6%), up to 10.1% compared to `BOUNDARY` (average 8.2%), and up to 10.7% compared to `SKIP` (average 9.0%).

7 A Further Generalization: Markov Processes

The λ parameter of IDK cascades is intended to model, and thereby exploit, the temporal dependence amongst inputs in a time-series input stream. Another factor that could potentially be exploited to further speed up the average classification duration is *performance correlation across the different IDK classifiers*. Some performance correlation across different ML models, whereby different models produce similar results on certain kinds of data, is often due to factors like training set similarity (where models learn from nearly identical data distributions), and data structure (such as spatial or temporal patterns), which guides models to rely on the same prominent features. Additionally, model capacity and complexity, shared optimization techniques and loss functions, and regularization practices (like early stopping) can lead models to converge on similar solutions. Data preprocessing consistency and starting from the same pretrained models also align performance, as do similar bias-variance trade-offs that suit a dataset's complexity. Together, these factors encourage different models to learn, generalize, and perform in similar ways, resulting in natural performance correlation across machine learning algorithms.

We conjecture that a *Markov Process* is well suited for representing both the performance correlation of models and the temporal inter-input dependence that tends to be present in the data-streams generated by CPS perception tasks. In future work, we intend to study the capabilities and limits of modeling both temporal dependencies and classifier correlations via Markov Processes; here we provide a brief overview presenting the essential idea.

A Markov Process is a stochastic process describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. For both model correlation and inter-input temporal dependence, we can consider this *state* to be the identity of the boundary classifier (i.e., the least powerful classifier in the contained collection of classifiers that is able to successfully identify the current input). We can then have an $(N \times N)$ transition matrix Ψ describe the probabilities of moving from one state to another for successive inputs. Specifically, in a transition matrix, each entry Ψ_{ij}, $(0 \leq \Psi_{ij} \leq 1)$, represents the probability of transitioning from K_i being the boundary classifier for one input to K_j being the boundary classifier for the subsequent input. The sum of the probabilities $\sum_j \Psi_{ij} = 1$ in each row (i) is 1, as they represent the total probability of moving from a particular state to any other state.

In this general formulation as a Markov Process, the single-parameter model and our generalization in Section 5.1 become special cases where state transition probabilities are correlated. When moving from the single-parameter model to the one with N parameters, one needs to learn more information during runtime via on-line learning. This problem is further exacerbated if the Markov Process model is used: $(N \times N)$ different parameter values must now be learned. While this can be accomplished by directly applying the Maximum Likelihood Estimation (MLE) techniques used by Agrawal et al. [2], learning more parameter values at adequately high confidence levels requires a lot more runtime observations since a lot more information is sought to be learned.

8 Conclusions

The accurate and rapid classification of streams of inputs that are captured by sensors is an essential enabler for perception for autonomous CPS's. In this paper, we focused on how to better leverage temporal dependencies in input streams to accelerate classification without sacrificing accuracy when using IDK cascades. A key insight was that simply starting classification from a later classifier in the cascade is suboptimal; improvements can be achieved

by selectively skipping classifiers altogether. Realizing this required a technical breakthrough: we developed a linear-time cascade synthesis algorithm, significantly faster than the state-of-the-art algorithm, enabling dynamic restructuring of cascades at runtime. We also introduced a more refined temporal dependence model, assigning individual parameters to each classifier rather than a single global parameter. This improved modeling accuracy and further reduced classification latency. Experiments on synthetic datasets confirmed the performance gains of our approach. Finally, we proposed a generalized modeling framework based on Markov Processes to jointly capture temporal dependencies and classifier correlations, showing our earlier models as special cases. We leave deeper exploration of this framework to future work.

References

1 Tarek Abdelzaher, Kunal Agrawal, Sanjoy Baruah, Alan Burns, Robert I. Davis, Zhishan Guo, and Yigong Hu. Scheduling IDK classifiers with arbitrary dependences to minimize the expected time to successful classification. *Real-Time Systems*, March 2023. doi:10.1007/s11241-023-09395-0.

2 Kunal Agrawal, Sanjoy Baruah, Alan Burns, and Jinhao Zhao. IDK cascades for time-series input streams. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, York, UK, 2024. IEEE Computer Society Press.

3 Sanjoy Baruah, Iain Bate, Alan Burns, and Robert Davis. Optimal synthesis of fault-tolerant IDK cascades for real-time classification. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2024)*. IEEE, 2024. doi:10.1109/RTAS61025.2024.00011.

4 Sanjoy Baruah, Alan Burns, Robert Davis, and Yue Wu. Optimally ordering IDK classifiers subject to deadlines. *Real Time Syst.*, 2022. doi:10.1007/s11241-022-09383-w.

5 Sanjoy Baruah, Alan Burns, and Robert I. Davis. Optimal synthesis of robust IDK classifier cascades. In Claire Pagetti and Alessandro Biondi, editors, *2023 International Conference on Embedded Software, EMSOFT 2023, Hamburg, Germany, September 2023*. ACM, 2023. doi:10.1145/3609129.

6 Sanjoy Baruah, Alan Burns, and Yue Wu. Optimal synthesis of IDK-cascades. In *Proceedings of the Twenty-Ninth International Conference on Real-Time and Network Systems*, RTNS '21, New York, NY, USA, 2021. ACM. doi:10.1145/3453417.3453425.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

8 Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(11):7436–7456, 2021. doi:10.1109/TPAMI.2021.3117837.

9 Yigong Hu, Shengzhong Liu, Tarek Abdelzaher, Maggie Wigness, and Philip David. On exploring image resizing for optimizing criticality-based machine perception. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 169–178, 2021. doi:10.1109/RTCSA52859.2021.00027.

10 Metod Jazbec, James Allingham, Dan Zhang, and Eric Nalisnick. Towards anytime classification in early-exit architectures by enforcing conditional monotonicity. *Advances in Neural Information Processing Systems*, 36:56138–56168, 2023.

11 Wittawat Jitkrittum, Neha Gupta, Aditya K Menon, Harikrishna Narasimhan, Ankit Rawat, and Sanjiv Kumar. When does confidence-based cascade deferral suffice? *Advances in Neural Information Processing Systems*, 36:9891–9906, 2023.

12 Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. Shallow-deep networks: Understanding and mitigating network overthinking. In *International conference on machine learning*, pages 3301–3310. PMLR, 2019. URL: http://proceedings.mlr.press/v97/kaya19a.html.

13 In Jae Myung. Tutorial on maximum likelihood estimation. *Journal of Mathematical Psychology*, 47(1):90–100, 2003. doi:10.1016/S0022-2496(02)00028-7.

14 Ishrak Jahan Ratul, Zhishan Guo, and Kecheng Yang. Cascading idk classifiers to accelerate object recognition while preserving accuracy. In *49th IEEE International Conference on Computers, Software, and Applications (COMPSAC)*. IEEE, 2025.

15 Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, and Joseph E. Gonzalez. IDK cascades: Fast deep learning by learning not to overthink. *CoRR*, abs/1706.00885, 2017. `arXiv:1706.00885`.