Optimal Synthesis of IDK-Cascades

Sanjoy Baruah
Washington University in St. Louis
Saint Louis, USA
baruah@wustl.edu

Alan Burns
The University of York
York, UK
alan.burns@york.ac.uk

Yue Wu
Washington University in St. Louis
Saint Louis, USA
yuewu767@wustl.edu

ABSTRACT

A classifier is a software component, often based upon deep learning (DL), that categorizes each input provided to it into one of a fixed set of "classes". An IDK classifier may additionally output an "I don't know" (IDK) on certain input. Given several different IDK classifiers for the same operation, the problem is considered of using them in concert in such a manner that the average duration to successfully classify any input is minimized. Optimal algorithms are proposed for solving this problem, both as is and under an additional constraint that the operation must be completed within a specified hard deadline).

KEYWORDS

Deep Learning; Classifiers; IDK-Cascades; Hard deadlines; Optimal Synthesis

ACM Reference Format:
Sanjoy Baruah, Alan Burns, and Yue Wu. 2021. Optimal Synthesis of IDK-Cascades. In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3453417.3453425

1 INTRODUCTION

We start out with a brief description, in abstract terms, of the real-time scheduling problem that is studied in this paper; this description is followed by a discussion that motivates the problem by explaining the kinds of scenarios being modeled by it, and why these scenarios are relevant to the design, analysis, and implementation of safety-critical real-time systems.

§1. The scheduling problem considered. Suppose that there is some operation that needs to be performed, and several different *components* $C_1, C_2, \ldots, C_n$ that are each designed to perform this operation in a different manner. Component $C_i = (d_i, p_i)$ is characterized by an *execution duration* d_i and a *success probability* p_i, indicating that it takes at most d_i time units to complete execution and has a probability p_i of successfully performing the intended operation. We will execute these components until one successfully performs the operation. (We assume that when a component completes execution it becomes known whether the operation was performed successfully or not.) Our objective is to schedule the execution of the components such that the expected time taken to successfully complete the operation is minimized.

§2. Motivation for this problem. Software components that are based on deep learning and related AI technologies are increasingly being deployed for classification problems in complex resource-constrained cyber-physical systems. Such systems often require accurate predictions to be delivered in real time using limited computing resources. However, much of the recent focus in deep neural network (DNN) research has been on improving the accuracy of classification. From a real-time perspective this ongoing quest for improved accuracy has arguably gone too far, resulting in DNN designs that take large durations of time processing even simple inputs that should be relatively straightforward to classify. (For instance with regards to image classification it was shown [9] that an order-of-magnitude increase in the execution duration of DNNs has resulted in a negligible improvement in the accuracy of predictions, for a considerable fraction of the ImageNet 2012 benchmark of validation images.) Balancing the trade-off between accuracy and latency becomes important if such DNNs are to be adopted for use in CPS's that are expected to respond in a timely manner. Towards this end Wang et al. [9] observed that if one were to only use the advanced (and slower) DNNs in the more challenging cases, then one could speed up computation without impacting accuracy by combining fast DNNs with accurate ones to reduce mean latency without a loss in accuracy. This observation motivated them to explore the use of ***IDK classifiers*** [3, 7], which may be be looked upon as bringing some degree of self awareness to classifiers. An IDK-classifier is obtained from an existing ("base") classifier by attaching a computationally light-weight augmenting classifier that enables the base classifier to additionally predict an auxiliary "**I D**on't **K**now" (IDK) class depending upon the degree of uncertainty of the base model predictions. Specifically, *the IDK classifier classifies an input as being in the IDK class if the base classifier is not able to predict some actual class for that input with a level of confidence that exceeds some pre-specified threshold value* – see Figure 1. Different IDK classifiers, of varying execution duration and likelihood of outputting IDK, may be devised for a single classification problem. (It is such IDK classifiers that are modeled as the components $C_1, C_2, \ldots, C_n$ in the scheduling problem introduced at the beginning of this paper.) Wang et al. [9] proposed that several different such IDK classifiers for the same classification problem be arranged into **IDK-cascades**, which are linear sequences of IDK classifiers designed to work as follows:

(1) The first classifier in the cascade is invoked first, for any input signal that needs to be classified.
(2) If the classifier outputs a real class (i.e., not "IDK"), then the cascade terminates and characterizes the input as being of the identified class.

RTNS'2021, April 7–9, 2021, NANTES, France

ACM ISBN 978-1-4503-9001-9/21/04…$15.00
https://doi.org/10.1145/3453417.3453425

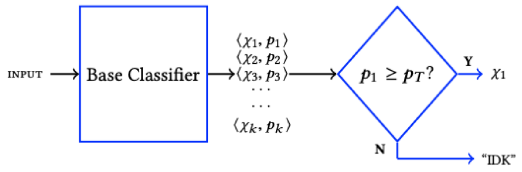


Figure 1: Obtaining an IDK classifier from a base classifier. For a given input, the base classifier outputs up to k ordered pairs $\langle \chi_i, p_i \rangle$, indicating that it believes that the input belongs to the class χ_i with probaility p_i. (It is assumed that $p_1 \geq p_2 \geq \cdots \geq p_k$, i.e., $\chi_1, \chi_2, \ldots, \chi_k$ are the k most likely classes, in order.) The threshold parameter for the IDK classifier is p_T.

(3) Else (i.e., this classifier outputs "IDK") the subsequent classifier in the cascade is invoked.
(4) This process is repeated until some classifier in the cascade outputs a real class.

It is required that all inputs be successfully classified by the cascade; hence it is assumed that the last classifier in the cascade always outputs a real class. (Wang et al. [9] propose that a human expert could be considered to be the last such classifier in the cascade – i.e., any input on which all the classifiers fail is placed before a human expert.)

Given a collection of several distinct IDK classifiers for a particular classification problem, the scheduling problem introduced at the beginning of this paper asks how they should be scheduled for execution in order to minimize the expected (i.e., average) duration taken to complete the classification operation. While that problem is concerned solely with minimizing expected duration, we will also consider a variant in which a hard deadline is also specified and the objective is to minimize the expected duration while simultaneously guaranteeing to always meet the specified deadline.

§3. Organization. The remainder of this paper is organized in the following manner. In Section 2 we define the problem we seek to solve, explain how this problem may be formulated in practice, and work through a few examples. As stated above there are two variants to our problem: (i) when there are no deadlines and (ii) when a hard deadline is specified. We develop solutions to the former in Section 3, and use these solutions to obtain solutions to the latter in Section 4. We conclude in Section 5 by placing this work in context, and briefly discussing some planned future research.

2 PROBLEM DEFINITION

In this section we formalize the problem of synthesizing IDK-cascades, justify our formalization, and work through a series of simple examples in order to develop our intuition regarding what solutions to this problem should look like.

As stated at the start of Section 1, a classifier denoted C_i may be characterized by an ordered pair (d_i, p_i) of parameters, its *execution duration* d_i and its *success probability* p_i, denoting that the classifier takes duration d_i to complete execution when invoked on an input and returns a real class (i.e., not "IDK") with probability p_i.

§. Problem Statement. Given n different components $C_1, C_2, \ldots, C_n$ for performing a certain operation, with each $C_i = (d_i, p_i)$ characterized by its execution duration d_i and its success probability p_i (as discussed above), determine which of these components should be executed, and in what order, such that the expected time taken to successfully complete the operation is minimized.

Note that for this expected duration to be finite, it is necessary that some component C_i with $p_i = 1$ be executed; henceforth in this paper we will therefore assume that some such component exists.

§. Obtaining the (d_i, p_i) values. Since our components are characterized by their d_i and p_i parameters, the values to these parameters must be known in order to have complete specifications for any given problem instance. We now discuss how these parameter values may be obtained in practice.

The execution duration d_i of a classifier is exactly that parameter of the classifier that we in the real-time computing community refer to as its worst-case execution time (WCET). WCET estimation is not the subject of this paper;[1] instead we assume that these parameter values are obtained using state-of-the-art WCET determination techniques.

In order to describe how the success probability p_i is determined, we must first better understand how IDK classifiers are obtained from regular ("base") classifiers – see Figure 1. Many DNN-based classifiers produce probabilistic outputs that encode their uncertainty in their predictions. Classifiers, for instance, produce a probability distribution over classes, and the entropy of this distribution for a given input encodes the classifier's confidence in the corresponding output. E.g., upon a particular input the output of a classifier may be the top few most likely classes to which this particular input should be mapped, along with the probability that it belongs to each of these classes. While classifiers may occasionally make incorrect predictions, they will typically rarely do so with high confidence. This is because such an outcome is heavily penalized in the objective functions used to train the DNNs.

An IDK classifier is obtained from such a base classifier by specifying some threshold probability p_T, reflecting the degree of confidence one desires in the classification. An IDK classifier

- outputs the most likely class to which the base classifier has mapped that input, if the base classifier determines that the probability that it (the input) belongs to this class is $\geq p_T$;
- outputs "IDK" otherwise (i.e., the input is not mapped into any class at a confidence $\geq p_T$).

The value to be assigned to the success probability parameter p_i of such an IDK classifier may be obtained by extensive testing: the IDK classifier is tasked with classifying a wide range of inputs, and p_i set equal to the fraction of these inputs upon which the IDK classifier outputs some class other than "IDK". (We emphasize that the success probability p_i of an IDK classifier is distinct from its threshold probability p_T: while p_T is a parameter whose value is set to reflect the desired degree of confidence one seeks from the classifier, the value of the p_i parameter is determined based on experimental evaluation of how frequently the classifier is able to provide responses at this degree of confidence.)

[1] See, e.g. [2] for a discussion on some issues that are relevant to accurate WCET analysis for DNNs.

§. Some notes. We emphasize that the problem of determining the value of p_i for the IDK classifier is distinct from the problem of ensuring that the base classifier's predictions are accurate —i.e., that the classes to which each input is mapped, and the associated probabilities, are meaningful. The latter problem is the subject of much ongoing research in the AI community, addressed as part of several initiatives (such as the Assured Autonomy program [6] of the United States Defense Advanced Research Projects Agency) that seek to develop trust-worthy machine-learning. However this problem is *not* the subject of this paper: here we assume that the base classifiers are indeed accurate in the sense that if a base classifier classifies some input as being to a particular class with a probability $\geq p_T$, then we can assume, with adequately high confidence, that this classification is the correct one.

We can also take a *mixed-criticality* [8] perspective upon IDK classifiers, by looking upon the problem of minimizing expected response time as a performance objective rather than a safety constraint. In that case the parameter p_i is not safety-critical in the sense that if an incorrect value is assigned to it then the only consequence is suboptimal performance (not a safety hazard). In contrast, accurate classification may well be safety-critical and in those cases, studied in Section 4, where a hard deadline is specified as a safety constraint, accurate (or at least, conservative) estimation of the d_i parameters –i.e., the WCETs– is also safety-critical.

§. Some simple examples. Given a collection of several distinct IDK classifiers for a particular classification problem, one should, informally speaking, construct an IDK-cascade by placing perhaps less accurate but faster classifiers earlier in the sequence in the hope that these would successfully classify the input most of the time, with more accurate classifiers that have greater execution duration being invoked in the (hopefully, rare) occasions that these earlier classifiers fail. This is illustrated in the following example.

Example 1. Suppose that we had a (regular, i.e., not IDK) classifier C_o for some classification problem that executes for no more than ten time units (represented as its execution duration parameter $d_o = 10$) and always predicts a real class – $p_0 = 1$ (and may hence be the last classifier in an IDK-cascade).

Suppose that we also have an IDK classifier C_1 with execution duration $d_1 = 5$ that returns a real class 60% of the time (and hence returns "IDK" 40% of the time) — as stated above, we represent this information by assigning the success probability parameter p_1 of this classifier the value 0.6.

Consider the IDK-cascade $[C_1; C_o]$, which executes C_1 first and subsequently executes C_0 only if C_1 fails (i.e., returns "IDK"). Since C_1 is always executed on all inputs but C_o only executes when C_1 outputs "IDK" (this is expected to happen with probability $(1-0.6)$), the expected running time of this IDK-cascade is

$$5 + (1 - 0.6) \times 10 \;=\; 5 + 4 = \mathbf{9}$$

which is smaller than 10, the running time if we were to always execute only the non-IDK classifier C_o. □

Of course, the down-side to going for the IDK-cascade rather than always executing C_o in the example above is that the *worst case* running times has increased: while we complete within five time units 60% of the time, we execute for fifteen time units the remaining 40% of the time. Whether this matters or not depends upon whether we are required to complete the classification by a specified deadline (in addition to having the objective of minimizing expected running time); in this paper we consider both possibilities.

i	d_i	p_i
0	10	1.00
1	5	0.60
2	3	0.20
3	6	0.75

Table 1: An example instance

The problem of designing the cascade gets more interesting when there are more IDK classifiers available to choose from. However even the simple case where there is only one IDK classifier available (that was illustrated in Example 1) is a bit nuanced; the reader may verify that given a non-IDK classifier C_o with execution duration d_o (and success probability $p_o = 1$) and an IDK classifier C_1 with execution duration d_1 and success probability p_1, it is preferable to use the IDK-cascade $\langle C_1; C_o \rangle$ rather than just executing the non-IDK classifier C_o in stand-alone fashion only if

$$d_1 < p_1 \cdot d_o.$$

Let us now step through a series of additional simple examples to gain further insights and help develop our intuition regarding this problem of synthesizing IDK-cascades.

Example 2. Consider again the instance from Example 1 above. Suppose that there were an additional IDK-classifier C_2 with execution duration $d_2 = 3$ that returns a real class only 20% of the time (i.e., its success probability parameter $p_2 = 0.2$) — see Table 1. It may be verified that the IDK-cascade $[C_1; C_2; C_o]$ has expected running time

$$5 + (1 - 0.6) \times 3 + (1 - 0.6) \times (1 - 0.2) \times 10 = 5 + 1.2 + 3.2 = \mathbf{9.4}$$

while the IDK-cascade $[C_2; C_1; C_o]$ has expected running time

$$3 + (1 - 0.2) \times 5 + (1 - 0.2) \times (1 - 0.6) \times 10 = 3 + 4 + 3.2 = \mathbf{10.2}$$

and the IDK-cascade $[C_2; C_o]$ has expected running time

$$3 + (1 - 0.2) \times 10 = 3 + 8 = \mathbf{11}$$

Each of these cascades has expected running time larger than that of the IDK-cascade $\langle C_1; C_o \rangle$ (which, we saw in Example 1, has an expected running time of 9); hence if minimizing expected running time is the objective then the IDK-classifier C_2 should not be used at all. □

Example 2 above illustrates one manner of synthesizing IDK-cascades optimally: simply enumerate all possible IDK-cascades and compute the expected running time for each, and choose the one with minimum expected running time. However such an approach is highly inefficient: given n IDK-classifiers (in addition to the classifier C_o that has $p_o = 1$ – i.e., always predicts the real class), the number of possible IDK-cascades is a very rapidly-growing exponential function of n. In this paper we will derive far more efficient[2] algorithms for synthesizing optimal IDK-cascades.

[2]Pseudo-polynomial if a hard deadline constraint is specified; quasi-linear running time ($\Theta(n \log n)$) without deadlines.

Example 3. Let us complete the example. Suppose we had the entire set of classifiers listed in Table 1 – this includes the classifier C_3, with execution duration $d_3 = 6$ and a success probability $p_3 = 0.75$, in addition to all those considered in Example 2 above – available to us. It is not immediately obvious by inspection of the parameters of the classifiers, what the optimal IDK-cascade should be; exhaustive enumeration would require us to consider

$$\frac{3!}{2!} + \frac{3!}{1!} + \frac{3!}{0!} = 3 + 6 + 6 = \mathbf{15}$$

different possibilities. We will see later that the IDK-cascade with minimum expected running time that can be constructed from these classifiers is $[C_3; C_1; C_0]$; this IDK-cascade has an expected running time equal to

$$6 + (1 - 0.75) \times 5 + (1 - 0.75) \times (1 - 0.6) \times 10 = 6 + 1.25 + 1 = \mathbf{8.25}$$

□

3 THERE IS NO HARD DEADLINE

Let us suppose we have the collection

$$\{C_i = (d_i, p_i)\}_{i=1}^{n}$$

of n IDK classifiers for performing a particular operation available to us, and our objective is to deploy them in order to minimize the expected duration taken to successfully complete the operation. If the amount of computational resources available to us were unbounded (or at least, there were n processors available to us), then the optimal strategy would be to execute all n IDK classifiers in parallel, each on its own dedicated processor, and stop upon the first successful (i.e., non-"IDK") completion. It is straightforward to determine what the expected duration would be using such a strategy. Let $\tilde{p_i}, \tilde{d_i}$ denote the success probability and execution duration of the classifier with the i'th smallest execution duration. Note that the overall execution duration would equal exactly $\tilde{d_i}$ if and only if the classifier with the i'th smallest execution duration is the first one to complete successfully. This happens if and only if this classifier succeeds and all of the smaller-duration classifiers fail – the probability of this happening is equal to $\tilde{p_i} \times \prod_{j=1}^{i-1}(1-\tilde{p_j})$. The expected duration is now obtained by summing over all possible values of i:

$$\sum_{i=1}^{n} \left(\prod_{j=1}^{i-1}(1 - \tilde{p_j}) \right) \tilde{p_i}\, \tilde{d_i} \tag{1}$$

In general, however, computational resources are only available in limited amounts and it is not feasible to execute all the components due to limited availability of computing capacity. In the remainder of this section we consider *uniprocessor* platforms, on which we can only execute one classifier at a time. [3] A uniprocessor schedule for our problem is then a sequence (a linear ordering) of some or all of the classifiers; during run-time we execute classifiers one at a time according to this schedule until some classifier is successful. We can use Lemma 1 below to compute the expected duration of any such schedule.

[3] We point out that this is not yet common practice: the use of DNNs in such resource-constrained platforms is currently very limited. Solving multiprocessor generalizations of this problem formulation would be interesting follow-up work – see Section 5.

Lemma 1. *Consider a uniprocessor schedule that orders n classifiers. Let $\tilde{p_k}, \tilde{d_k}$ denote the success probability and execution duration of the k'th classifier in this schedule. The expected duration of this schedule is*

$$\sum_{k=1}^{n} \left(\prod_{j=1}^{k-1}(1 - \tilde{p_j}) \right) \tilde{d_k} \tag{2}$$

Proof. Note that the first classifier will always execute; the second classifier will execute if and only if the first one fails (this happens with probability $(1 - \hat{p}_1)$); the third classifier will execute if and only if the first two both fail (this happens with probability $(1 - \hat{p}_1) \cdot (1 - \hat{p}_2)$; and so on. Hence the expected duration is

$$\hat{d}_1 + (1 - \hat{p}_1)\, \hat{d}_2 + (1 - \hat{p}_1) \cdot (1 - \hat{p}_2)\, \hat{d}_3 + \cdots + \prod_{j=1}^{n-1}(1 - \hat{p}_j)\, \hat{d}_n$$

which is represented compactly as in Expression 2. □

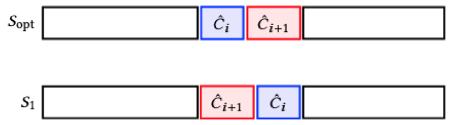
Lemma 2 below identifies an important characteristic of any schedule of minimum expected duration:

Lemma 2. *Let classifier C_j be scheduled for execution after classifier C_i in an optimal schedule (i.e., in a schedule with minimum expected duration). It must be the case that*

$$\frac{d_i}{p_i} \leq \frac{d_j}{p_j} \tag{3}$$

Proof. Below we will establish that any two adjacently scheduled classifiers C_i and C_j satisfy Expression 3 above. The lemma will then follow from the transitivity of the $\geq$ relationship on $\mathbb{R}$ (the real numbers).

Let S_{opt} denote an optimal schedule, and let $\hat{C}_i$ denote the classifier in the i'th position of this schedule for any i, $1 \leq i \leq n$. Let $\hat{p}_i, \hat{d}_i$ denote the success probability and execution duration of classifier $\hat{C}_i$. Let S_1 denote a schedule obtained from S_{opt} by swapping the classifiers in the i'th and $(i+1)$'th positions in S_{opt}:



Note that by using Expression 2 in Lemma 1, the expected duration for S_{opt} can be written in the following manner, as the sum of three terms representing respectively the outer summation of Expression 2 for $k \in \{1, \ldots, i-1\}$, $k \in \{i, i+1\}$, and $k \in \{i+2, \ldots, n\}$:

$$\begin{aligned}
&\sum_{k=1}^{i-1} \left(\prod_{j=1}^{k-1}(1 - \hat{p}_j) \right) \hat{d}_k && \text{//The first } (i-1) \text{ components} \\
+\ &\prod_{j=1}^{i-1}(1 - \hat{p}_j) \left(\hat{d}_i + (1 - \hat{p}_i)\hat{d}_{i+1} \right) && \text{//The next two components} \\
+\ &\sum_{k=i+2}^{n} \left(\prod_{j=1}^{k-1}(1 - \hat{p}_j) \right) \hat{d}_k && \text{//The remaining components}
\end{aligned} \tag{4}$$

Now let us turn our attention to the expected duration of S_1, again using Lemma 1 (Expression 2). We can also write its expected duration as the sum of three terms. Since S_1 only differs from S_{opt} in that the i'th and $(i+1)$ components are swapped, the first and last terms represent the same quantities as the first and third terms of

OptOrder($\{(d_1, p_1), (d_2, p_2), \ldots, (d_n, p_n)\}$)

1 Compute d_i/p_i for all $i, 1 \leq i \leq n$
2 Sort in non-decreasing order of d_i/p_i
3 Output the classifiers according to their position in the sorted list above, stopping at the first one that has success probability equal to 1

Figure 2: Algorithm for synthesizing a schedule with minimum expected duration

Expression 4 above, and are therefore identical. The middle term, however is now

$$\prod_{j=1}^{i-1}(1-\hat{p_j})\left(\hat{d}_{i+1} + (1-\hat{p}_{i+1})\hat{d}_i\right) \tag{5}$$

since the order of the i'th and $(i+1)$ components has been swapped.[4]

Schedule S_{opt} is, by definition, an optimal schedule. Its expected duration is therefore no larger than the expected duration of S_1, i.e., the middle term of Expression 4 is no larger than Expression 5:

$$\prod_{j=1}^{i-1}(1-\hat{p_j})\left(\hat{d}_i + (1-\hat{p_i})\hat{d}_{i+1}\right) \leq \prod_{j=1}^{i-1}(1-\hat{p_j})\left(\hat{d}_{i+1} + (1-\hat{p}_{i+1})\hat{d}_i\right)$$

Observing that the term $\prod_{j=1}^{i-1}(1-\hat{p_j})$ appears on both sides of the expression above, we have that

$$\begin{aligned}
&\left(\hat{d}_i + (1-\hat{p_i})\hat{d}_{i+1}\right) \leq \left(\hat{d}_{i+1} + (1-\hat{p}_{i+1})\hat{d}_i\right) \\
\Leftrightarrow\quad &\left(\hat{d}_i + \hat{d}_{i+1} - \hat{p_i}\hat{d}_{i+1}\right) \leq \left(\hat{d}_{i+1} + \hat{d}_i - \hat{p}_{i+1}\hat{d}_i\right) \\
\Leftrightarrow\quad &\hat{p}_{i+1}\hat{d}_i \leq \hat{p_i}\hat{d}_{i+1} \\
\Leftrightarrow\quad &\frac{\hat{d}_i}{\hat{p_i}} \leq \frac{\hat{d}_{i+1}}{\hat{p}_{i+1}}
\end{aligned}$$

and the proof is complete. □

Synthesizing an optimal schedule. Lemma 2 implies that classifiers scheduled adjacent to each other in any schedule of minimum expected duration must have the ratio of their execution duration to their success probability arranged in non-decreasing order. An algorithm for synthesizing schedules of minimum expected duration immediately suggests itself : simply determine these ratios for all the classifiers, and sort this list in non-decreasing order – see Figure 2. Note, as depicted in Figure 2, that the schedule need not enumerate the classifiers beyond the first one that has its success probability (its p_i parameter) equal to one, since it is guaranteed to complete successfully (and hence classifiers listed after it in the schedule will never execute).

Example 4. Consider again the example instance of Table 1, comprising the four classifiers C_0, C_1, C_2 and C_3. The p_i/d_i values are as follows:

i	d_i	p_i	(d_i/p_i)
0	10	1.00	10
1	5	0.60	8.33
2	3	0.20	15
3	6	0.75	8

[4]We point out that the difference between Expression 5 and the middle term of Expression 4 is that the roles of i and $(i+1)$ are swapped in the expression $(\hat{d}_x + (1-\hat{p}_x)\hat{d}_y)$.

Sorted in non-decreasing order of these d_i/p_i ratios, the four classifiers are ordered as follows:

$$[C_3,\ C_1,\ C_0,\ C_2]$$

Since $p_o = 1$, the classifiers listed after C_o in the list above will never be executed (and may hence be removed from the final schedule); the final schedule is therefore

$$[C_3,\ C_1,\ C_0]$$

as had been claimed in Example 3. □

Running time. The algorithm of Figure 2 is very efficient: its running time is dominated by the sorting step and it can hence be implemented to execute in $\Theta(n \log n)$ time.

Characterization of timing behavior. The actual execution duration of an IDK-cascade depends upon which component in the cascade first returns a real class rather than IDK, and may therefore be different on different executions. The *worst-case* execution duration of the cascade is simply the sum of the worst-case execution durations of all the components in the cascade. If the individual WCET estimates (the d_i's) are safe WCET estimates for the respective components, then $\sum_{i=1}^{n} d_i$ is a safe WCET estimate for the IDK-cascade.

A stochastic characterization of the run-time timing behavior of an IDK-cascade can also be obtained in terms of the d_i and p_i values characterizing the individual components comprising the cascade. Notice that for each i, the duration if the cascade terminates upon execution of the i'th component is equal to

$$\sum_{j=1}^{i} d_i$$

and the probability that this will happen is given by the expression

$$p_i \times \prod_{j=1}^{i-1}\left(1-p_j\right).$$

Analysis of systems comprising multiple such IDK-cascades executing upon a shared processor may be done using previously-proposed techniques (see, e.g, [4, 5]) for schedulability analysis of systems with stochastic task execution times.

4 A DEADLINE IS SPECIFIED

We now consider the variant of the problem in which a hard deadline is also specified, and the objective is to achieve the minimum expected duration subject to the constraint that the hard deadline is guaranteed to be met. Specifically, an instance is specified as

$$\left\langle \{C_i = (d_i, p_i)\}_{i=1}^{n}, D \right\rangle \tag{6}$$

where $C_1, C_2, \ldots, C_n$ are n IDK classifiers, and $D \in \mathbb{N}$ is the specified deadline. Our objective is to deploy the classifiers in order to

minimize the expected duration taken to successfully complete the operation; however, we must ensure that the operation *always* completes within a duration D. It is obvious that such a guarantee can be made for an instance if and only if there is some classifier with success probability one and execution duration $\leq D$ – instances satisfying this property are said to be ***feasible*** instances (while instances lacking this are *in*feasible).

As in the case without deadlines (Section 3), if the amount of available computational resources is unbounded then for feasible instances the optimal strategy is to execute all the classifiers in parallel,[5] each on its own dedicated processor, and stop upon the first successful completion. (The expected completion time for this schedule can be determined in a manner similar to the manner in which this was done Section 3.) In the remainder of this section we consider the uniprocessor case where we have just a single processor upon which to execute the classifiers. As for the no-deadline version of the problem (Section 3), a uniprocessor schedule is a linear sequence of some or all of the classifiers; during run-time we execute classifiers one at a time according to this schedule until some classifier is successful. The new wrinkle introduced by the presence of the added constraint of a hard deadline is that the sum of the execution durations of all the classifiers included in this linear sequence cannot exceed D (and as before, the last classifier in the sequence must have a success probability equal to one).

Some assumptions for this section. Note that Lemma 2 continues to hold regardless of the presence of deadlines: adjacent classifiers in an optimal schedule will satisfy the property that the ratio of their execution duration to their success probability (i.e., d_i/p_i) is non-decreasing. Without loss of generality, let us therefore *assume that the classifiers are indexed according to non-decreasing* d_i/p_i: for all i we have

$$\frac{d_i}{p_i} \leq \frac{d_{i+1}}{p_{i+1}}$$

This assumption can be realized for any instance by sorting, in $\Theta(n \log n)$ time. Additionally, consider the smallest i such that $p_i = 1$; as in Section 3, it can be argued that classifiers with index greater than i will never be executed in any optimal schedule. Therefore, we assume without loss of generality that n denotes the index of this component: i.e., *we assume that* $p_n = 1.0$ *and* $p_i < 1.0$ *for all* $i < n$.

Example 5. Consider an instance with the following three classifiers,[6] and a deadline $D = 10$.

i	d_i	p_i
0	2	0.4
1	4	0.8
2	6	1.0

The schedule must terminate with C_2, since $p_2 = 1.0$ (and further, neither p_0 nor p_1 is 1.0). Since $d_0 + d_1 + d_2 = 12$ which exceeds the deadline of 10, we cannot schedule all three for execution. We therefore have a choice of two schedules: $[C_0, C_2]$ or $[C_1, C_2]$.

[5]In fact, it suffices to only execute those that have duration $\leq D$.

[6]It may be verified that these three classifiers are indeed arranged in non-decreasing order of d_i/p_i.

– The expected duration of the schedule $[C_0, C_2]$ is

$$\begin{aligned} & d_o + (1 - p_o)d_2 \\ = \;& 2 + (1 - 0.4)6 = 2 + 3.6 = \mathbf{5.6} \end{aligned}$$

– The expected duration of the schedule $[C_1, C_2]$ is

$$\begin{aligned} & d_1 + (1 - p_1)d_2 \\ = \;& 4 + (1 - 0.8)6 = 4 + 1.2 = \mathbf{5.2} \end{aligned}$$

Hence the second schedule, $[C_1, C_2]$, is the optimal one □

We suspect (but have not yet proved) that the problem of synthesizing an optimal schedule of this form – i.e., of minimum expected duration that always meets a specified deadline) is an NP-hard one. Given a problem specified as in Expression 6 (and with the assumptions that the classifiers are indexed in non-decreasing d_i/p_i order, and that $p_n = 1.0$ while $p_i < 1.0$ for all $i < n$), in the remainder of this section we will apply the technique of dynamic programming [1] to determine an optimal schedule: a schedule of duration $\leq D$ in which the last classifier is the deterministic classifier C_n. We start with a definition.

Definition 1. Let $E(d, k)$ denote the minimum expected duration for the following sub-problem of the problem instance specified in Expression 6:

$$\left\langle \{C_i = (d_i, p_i)\}_{i=k}^{n}, d \right\rangle$$

That is, only the classifiers $C_k, C_{k+1}, \ldots, C_n$ are available to us and we have a deadline of d.

Using this notation, the expected duration of the optimal solution to the problem instance specified in Expression 6 is $E(D, 1)$: the deadline is D, all n classifiers $C_1, C_2, \ldots, C_n$ are available.

Let us first look at the sub-problem when only the classifier C_n is available to us, and compute the values of $E(d, n)$ for all values of d. We claim

$$E(d, n) = \begin{cases} \infty, & \text{if } d < d_n \\ d_n, & \text{otherwise (i.e., } d \geq d_n) \end{cases} \tag{7}$$

That is, if we have a deadline d and only the (deterministic) classifier C_n available, then the instance is infeasible (represented here as having an expected duration of infinity) if the deadline is smaller than the C_n's execution duration d_n. If $d \geq d_n$, then our optimal schedule comprises the classifier C_n and hence the expected duration (which in fact, equals the specified WCET) is d_n.

Now, let us assume that we have already determined the values of $E(d', k + 1)$ for all d'; we wish to compute the value of $E(d, k)$. We have

$$E(d, k) = \min\Big\{E(d, k + 1), d_k + (1 - p_k) \cdot E(d - d_k, k + 1)\Big\} \tag{8}$$

where

– The first term within the min reflects the decision to not use the classifier C_k (and hence the expected duration is equal to the minimum expected duration using only the classifiers $C_{k+1}, C_{k+2}, \ldots, C_n$).
– The second term within the min reflects the decision to use the classifier C_k. In that case
  The classifier C_k is definitely executed (since, according to Lemma 2, it precedes the remaining classifiers in the optimal schedule), and takes a duration d_k.

```
OptOrder(⟨{(d_1, p_1), (d_2, p_2), ..., (d_n, p_n)}, D⟩)
    // Input should be sorted according to d_i/p_i: d_i/p_i ≥ d_{i+1}/p_{i+1} for all i
    // It is assumed that (i) p_n = 1.0; and (ii) d_n ≤ D (else the instance is infeasible).
    // Also, assume that p_i < 1 for all i < n.
 1  E[(0, ..., D) × (1, ..., n)] of integers // Will be filled in using Dynamic Programming
    // Initializing E[d, n] for all d (Expression 7)
 2  for d = 0 to (d_n − 1)
 3      E[d, n] = ∞
 4  for d = d_n to D
 5      E[d, n] = d_n
    // Implementing the recurrence (Expression 8)
 6  for k = (n − 1) downto 1
 7      for d = 1 to D
            // Compute E[d, k] according to Expression 8
 8          E[d, k] = E[d, k + 1] // Initialize E[d, k] to first term in the "min" of Expression 8
 9          if (d ≥ d_k) // It is possible to execute C_k...
10              tmp = d_k + (1 − p_k) × E[d − d_k, k + 1] // Minimum expected duration if C_k is executed (second term
                                                             in the "min" of Expression 8)
11              if (tmp < E[d, k]) // It is better to execute C_k...
12                  E[d, k] = tmp // Update E[d, k] to second term in the "min" of Expression 8
    // Printing the optimal schedule
13  d = D
14  for k = 1 to n
15      if (E[d, k] ≠ E[d, k + 1])
            // C_k must have been selected...
16          print out C_k
17          d = d − d_k
```

Figure 3: Algorithm for synthesizing a schedule with minimum expected duration and bounded worst-case duration

C_k fails to complete successfully with a probability $(1 - p_k)$. When this happens, the remainder of the schedule is executed, and has an expected duration $E(d - d_k, k + 1)$.

The second term within the min is obtained as the sum of C_i's execution duration (d_k), and the contribution to the expected duration in the event that C_i fails to complete successfully ($(1 - p_k) \cdot E(d - d_k, k + 1)$).

We can use Equation 7 to determine $E(d, n)$ for all d. Having done so, repeated applications of Equation 8 enable to determine $E(d, n-1)$, $E(d, n-2)$,..., $E(d, 1)$, for all d, and thereby obtain the value of $E(D, 1)$ which (as mentioned earlier) is the expected duration of the optimal schedule. Furthermore, the actual optimal schedule that has this duration can be deduced by observing whether the the first or the second term in the "min" is smaller in each application of Equation 8. An algorithm for determining the optimal schedule by doing so is depicted in pseudo-code form in Figure 3.

Running time. The worst-case running time of the pseudo-code of Figure 3 is dominated by the running time of the nested **for** loops. The outer **for** loop executes n times and the inner one, D times; the overall running time is therefore $\Theta(n\,D)$ where n denotes the number of classifiers and D, the specified hard deadline.

A simple heuristic (that is not optimal). We have seen that the algorithm of Figure 3 has running time pseudo-polynomial in the representation of its input. We believe this is efficient enough for many applications: in most real-time CPS's it is unlikely that the specified deadline (the "D") will be very large. However in the unlikely event that a pseudo-polynomial running time is considered unacceptably high, we can always derive a greedy heuristic from the algorithm of Section 3, by extending it to account for deadlines as follows.

Given the classifiers indexed in non-decreasing order of their d_i/p_i ratios, we greedily schedule the classifiers in the given order while ensuring that the classifier C_n can be accommodated. That is, we consider the classifiers in index order: in considering a classifier, we schedule it if and only if doing so will leave us with $\geq d_n$ time before the deadline (so that the deterministic classifier C_n can be accommodated).

This heuristic, like the algorithm of Section 3 (Figure 2), has linear (i.e., $\Theta(n)$) running time if the classifiers are already sorted, or $\Theta(n \log n)$ if the cost of sorting must also be accounted for. It is, however, easily seen to be non-optimal: consider the following example.

Example 6. We have three classifiers $C_1 = (d_1, p_1)$, $C_2 = (d_2, p_2)$, and $C_3 = (d_3, 1.0)$, and a deadline D satisfying

$$D \geq \big(\max(d_1, d_2) + d_3\big) \text{ and } D < (d_1 + d_2 + d_3).$$

Hence exactly one of C_1 and C_2, and the deterministic classifier C_3, can be scheduled. Let us suppose that d_1/p_1 is a bit smaller than d_2/p_2, and p_1 is also a bit smaller than p_2. (E.g., $(d_1, p_1) = (10, 0.5)$ and $(d_2, p_2) = (13, 0.6)$, so that $d_1/p_1 = 20$ while $d_2/p_2 = 21\frac{2}{3}$.)

Our greedy heuristic would schedule $[C_1, C_3]$ for an expected duration of

$$d_1 + (1 - p_1)d_3$$

which, for $(d_1, p_1) = (10, 0.5)$ and $(d_2, p_2) = (13, 0.6)$, evaluates to $(10 + 0.5\, d_3)$.

Now suppose the optimal algorithm (Figure 3) were to come up with the schedule $[C_2, C_3]$, with expected duration

$$d_2 + (1 - p_2)d_3$$

— for our example of $(d_1, p_1) = (10, 0.5)$ and $(d_2, p_2) = (13, 0.6)$, this evaluates to $(13 + 0.4\, d_3)$.

Consider now the *approximation ratio* of our greedy heuristic:

$$\frac{d_1 + (1 - p_1)\, d_3}{d_2 + (1 - p_2)\, d_3}$$

It is straightforward to observe that as $d_3 \longrightarrow \infty$, this approximation ratio approaches $\frac{(1-p_1)}{(1-p_2)}$ which, for $(d_1, p_1) = (10, 0.5)$ and $(d_2, p_2) = (13, 0.6)$, evaluates to $0.5/0.4$ or **1.25**. □

There is nothing particularly intrinsic about the value **1.25** that was derived as the lower bound on the approximation ratio in Example 6 above; we leave it to the reader to verify that the technique can be generalized to show a lower bound equal to any positive number. This allows us to conclude that *the greedy heuristic may perform arbitrarily poorly in comparison to the optimal algorithm* of Figure 3.

5 CONTEXT & CONCLUSIONS

Learning-enabled components, particularly those based on Deep Neural Networks (DNNs), are being increasingly used in safety-critical real-time systems. It is imperative that the real-time scheduling theory community respond to this development by coming up with appropriate techniques to enable the pre-runtime analysis of systems that use such components.

In this work, we have adapted and applied algorithmic techniques from real-time scheduling theory to a proposed DNN use-case [9] that seeks to strike a balance between accuracy and timeliness by arranging individual DNN-based classifiers, augmented by the ability to classify inputs as belonging to an additional "IDK" class, into IDK-cascades. The intuition behind the design of IDK-cascades is simple and elegant: the earlier classifiers in a cascade should successfully classify simple-to-classify inputs, thus requiring that the later classifiers only be invoked upon truly challenging inputs. We were able to formalize this intuition and thereby develop algorithms that synthesize IDK-cascades from a given set of classifiers in an optimal manner, both when the sole objective is optimizing expected timeliness and when there is an additional hard deadline constraint.

We look upon our results in this paper as a proof of concept of the principle that real-time scheduling can contribute to better design of real-time systems that use of learning-enabled components – it behoves us, as a community, to take a closer look at such systems. A rich agenda of research in this direction can be defined. One such example: we plan to develop algorithms for solving both the problems considered in this paper (i.e., the no-deadline and the hard-deadline variants) upon multiprocessor platforms.

REFERENCES

[1] Richard Bellman. 1957. *Dynamic Programming* (1 ed.). Princeton University Press, Princeton, NJ, USA.

[2] Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. 2020. Timing isolation and improved scheduling of deep neural networks for real-time systems. *Software: Practice and Experience* 50, 9 (2020), 1760–1777. https://doi.org/10.1002/spe.2840 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2840

[3] F. Khani, M. Rinard, and P. Liang. 2016. Unanimous Prediction for 100% Precision with Application to Learning Semantic Mappings. In *Association for Computational Linguistics (ACL)*.

[4] S. Manolache, P. Eles, and Z. Peng. 2001. Memory and time-efficient schedulability analysis of task sets with stochastic execution time. In *Proceedings 13th Euromicro Conference on Real-Time Systems*. 19–26. https://doi.org/10.1109/EMRTS.2001.933991

[5] Sorin Manolache, Petru Eles, and Zebo Peng. 2004. Schedulability Analysis of Applications with Stochastic Task Execution Times. *ACM Trans. Embed. Comput. Syst.* 3, 4 (Nov. 2004), 706–735. https://doi.org/10.1145/1027794.1027797

[6] Dr. Sandeep Neema. [n.d.]. Assured Autonomy. https://www.darpa.mil/program/assured-autonomy. Accessed: 2019-03-07.

[7] T. P. Trappenberg and A. D. Back. 2000. A classification scheme for applications with ambiguous data. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, Vol. 6. 296–301 vol.6.

[8] Steve Vestal. 2007. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society Press, Tucson, AZ, 239–243.

[9] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, and Joseph E. Gonzalez. 2017. IDK Cascades: Fast Deep Learning by Learning not to Overthink. *CoRR* abs/1706.00885 (2017). arXiv:1706.00885 http://arxiv.org/abs/1706.00885