Scalable-effort Classifiers for Energy-efficient Machine Learning

ABSTRACT

Supervised machine-learning algorithms are used to solve classification problems across the entire spectrum of computing platforms, from data centers to wearable devices, and place significant demand on their computational capabilities. In this paper, we propose *scalable-effort classifiers*, a new approach to optimizing the energy efficiency of supervised machine-learning classifiers. We observe that the inherent classification difficulty varies widely across inputs in real-world datasets; only a small fraction of the inputs truly require the full computational effort of the classifier, while the large majority can be classified correctly with very low effort. Yet, state-of-the-art classification algorithms expend equal effort on all inputs, irrespective of their complexity. To address this inefficiency, we propose a systematic approach to design scalable-effort classifier that dynamically adjust their computational effort depending on the difficulty of the input data, while maintaining the same level of accuracy. Our approach utilizes a chain of classifiers with increasing levels of complexity (and accuracy). Scalable effort execution is achieved by modulating the number of stages used for classifying a given input. Every stage in the chain is constructed using an ensemble of biased classifiers, which is trained to detect a single class more accurately. The degree of consensus between the biased classifiers' outputs is used to decide whether classification can be terminated at the current stage or not. Our methodology thus allows us to transform any given classification algorithm into a scalable-effort chain. We build scalable-effort versions of 8 popular recognition applications using 3 different classification algorithms. Our experiments demonstrate that scalable-effort classifiers yield 2.79× reduction in average OPS per input, which translates to 2.3× and 1.5× improvement in energy and runtime over well-optimized hardware and software implementations, respectively.

1. INTRODUCTION

For many computational systems, all inputs are not created equal. Consider the simple example of 8-bit multiplication; intuitively, computing the product of 02h and 01h should be easier than multiplying 19h and 72h. Similarly, compressing a picture that contains just the blue sky should take less effort than one that contains a busy street. Ideally, to improve both speed and energy efficiency, algorithms should expend effort (computational time and energy) that is commensurate to the difficulty of the inputs. Unfortunately, for most applications, discriminating easy inputs from hard ones at runtime is challenging. Thus, hardware or software implementations tend to expend constant computational effort as determined by worst-case inputs or a representative set of inputs. In this paper, we focus on a specific, important class of algorithms - machine learning classifiers - and show how they can be constructed to scale their computational effort depending on the difficulty of the input data, leading to faster and more energy-efficient implementations.

Machine-learning algorithms are used to solve an ever-increasing range of classification problems in recognition, vision, search, analytics and inference across the entire spectrum of computing platforms [1]. Machine learning algorithms operate in two phases: training and testing. In training, decision models are constructed based on a labeled training data set. In testing, the learnt model is applied to classify new input instances. The intuition behind our approach is as follows. During the training phase, instead of building one complex decision model, we construct a cascade or series of models with progressively increasing complexity. During testing, depending on the difficulty of an input instance, the number of decision models applied to it is varied, thereby achieving scalability in time and energy.

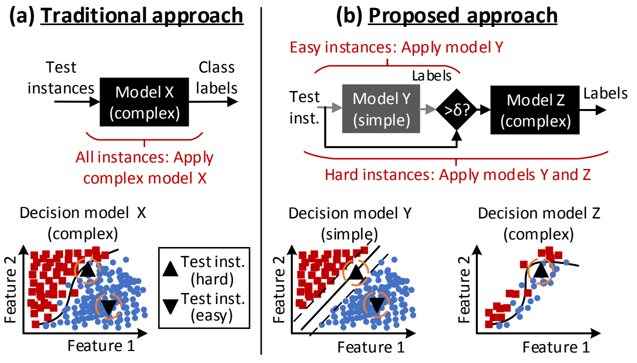


Figure 1: (a) Traditional approach: learn one (complex) model; apply to all instances. (b) Proposed approach: learn multiple models; apply one or more depending on difficulty of input.

Fig. 1 illustrates our methodology through a specific machine learning algorithm namely a binary support-vector machine (SVM) classifier. In the traditional approach shown in Fig. 1(a), input training examples are used to build a decision boundary (model X) that separates data into two categories or classes. At test time, data instances are assigned to one class or the other depending on their location relative to the decision boundary. The computational effort (in terms of energy and time) to process every test instance depends on the complexity of the decision boundary, *e.g.*, non-linear boundaries typically cost more than linear ones. In the example of Fig. 1(a), a single model (X) clearly needs to use a non-linear boundary in order to separate the classes with high accuracy. However, this leads to high computational effort for not only the hard test data instances (points close to the decision boundary) but also the easy test data instances (points far removed from the decision boundary). In contrast, Fig. 1(b) shows our approach, where we create multiple decision models (Y and Z) with varying levels of complexity. In the simpler model (Y), two different linear decision boundaries (dashed lines) are used to classify the easier training instances, while leaving a subset of training instances unclassified. The complex model (Z) is employed only for instances that cannot be classified by the simpler model. This approach can save time and energy, since all data instances need not be processed by the more complex non-linear decision model.

The amount of computational time and energy saved depends on the application at hand. Fortunately, in many useful applications, lots of test data is easy. For instance, while detecting movement using a security camera, most video frames contain only static objects. We quantify this intuition for the popular MNIST handwriting recognition dataset [2]. Fig. 2 shows statistics from the dataset (inset shows some representative hard and easy instances). Observe that only 5-30% of the data is close to the decision boundary.

We generalize the approach described above for any machine

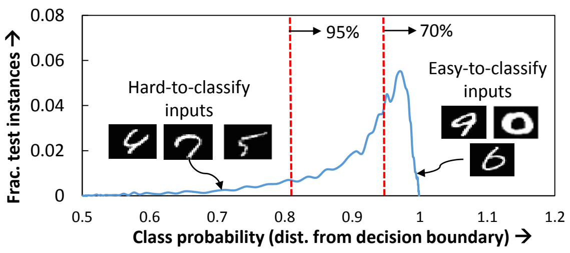


Figure 2: MNIST dataset: only 5-30% of instances are hard.

learning classifier by constructing a cascaded series of classification stages with progressively increasing complexity and accuracy. We also show how to construct simpler models for each stage by using an ensemble of biased classifiers.

How do we determine the difficulty of an instance at runtime? Besides model partitioning, this is another challenge that we address in the paper. We determine the hardness of each test data instance implicitly. The top portion of Fig. 1(b) illustrates our approach. We process test instances through the decision models in a sequence starting from the simplest model. After the application of every model, we estimate the confidence level of the produced output (*i.e.*, the class probability or classification margin). Constructing each model as an ensemble of biased classifiers further facilitates this, since their consensus may be used to indicate the confidence of classification. If the confidence is above a threshold, we accept the output class label produced by the current model and terminate the classification process. Simpler data instances get processed through only the initial few (simpler) models, while harder instances need to go through more models. Thus, our approach provides an inbuilt method to scale computational effort at test time.

In summary, we make the following contributions:

- Given any machine learning classifier, we propose a systematic approach to construct a scalable-effort version thereof by cascading classification stages of growing accuracy and complexity. The scalable-effort classifier has accuracy comparable to the original one, while being faster and more energy efficient.

- To construct the stages of the scalable effort classifier, we propose ensembles of biased classifiers and a consensus operation that determines the confidence level in the class labels produced by the classification stage.

- We present an algorithm to train a scalable effort classifier that trades off the number of stages, complexity of each stage, and fraction of inputs classified by each stage, to optimize the overall computational effort spent in classification.

- Across a benchmark suite of eight applications that utilize 3 classification algorithms, we show that scalable effort classifiers provide 1.5× average reduction in runtime. Through hardware implementations in a 45nm SOI process, we also demonstrate an average of 2.3× reduction in energy.

The rest of the paper is organized as follows. In Section 2, we present related work. In Section 3, we describe our approach to the construction of scalable-effort classifiers. In Section 4, we describe a methodology to construct such classifiers. In Section 5, we describe our evaluation methodology and benchmarks. We present experimental results in Section 6 and conclude in Section 7.

2. RELATED WORK

Most previous efforts in building input-aware computational systems have considered application-specific solutions [3, 4]. Generalizing such approaches to arbitrary circuits, or even classes of applications as we attempt, is non-trivial.

In a broad sense, approximate computing [5–11], which exploits the resilience of applications to approximate or inexact execution of their underlying computations, also leverages the fact computational effort can be scaled at different levels of abstraction [5, 6]. However, these techniques usually provide an explicit tradeoff between efficiency and quality of results. In contrast, our approach provides energy savings, while maintaining classification accuracy. Thus, these existing methods are complementary to the concept of scalable-effort classifiers.

On the algorithmic front, using multiple classifiers for increasing learning accuracy is an active area of research [12]. However, using them to reduce energy and runtime has only received limited attention. The closest related approach is the method of cascaded classification [13]; the Viola-Jones algorithm used for face detection is a classic example [14]. It comprises a 21-stage cascade of simple detectors that operate on multiple patches of an image. At each stage, if an image patch matches a particular pattern, it is passed on to the next stage for classification; if not, it is rejected early in the chain. Thus, cascaded classifiers provide a limited form of one-class scalability (*i.e.*, for instances belonging to the non-face class). For an image to be deemed belonging the face-class, however, it has to pass through all stages, resulting in fixed computational effort [15]. Another recent attempt employs a tree of simple classification models [16], which again exhibits scalable effort in a limited form.

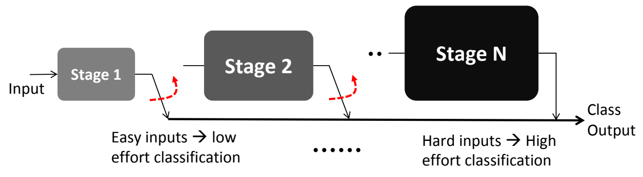
The methodology that we propose in this paper builds upon the concept of cascading classifiers. Unlike existing work, which applies to only faces, our methodology is generic and applicable to any given classification algorithm and dataset. Further, unlike existing work, which allows only early rejects, our approach allows early class labeling (including multi-class labels) at any stage along the scalable-effort chain. Aditionally, we explore new insights into the micro-architecture of the classifier, including the associated energy-accuracy trade-offs.

3. SCALABLE-EFFORT CLASSIFIERS

In this section, we present our structured approach to design scalable effort classifiers. Fig. 3 shows the conceptual view of a scalable-effort classifier. Given any classification algorithm, different models are learnt using the same algorithm and training data. These models are then connected in a sequence such that the initial stages are computationally efficient but have lower classification accuracies, while the later ones have both higher complexities and accuracies. Further, each stage in the cascade is also designed to implicitly assess the hardness of the input. During test time, data is processed through each stage, starting from the simplest model, to produce a class label. The stage also produces a confidence value associated with the class label. This value determines whether the input is passed on to the next stage or not. Thus, class labels are produced earlier in the chain for easy instances and later for the hard ones. If an instance reaches the final stage, the output label is used irrespective of the confidence value. Next, we present more details on how each stage of Fig. 3 is designed.

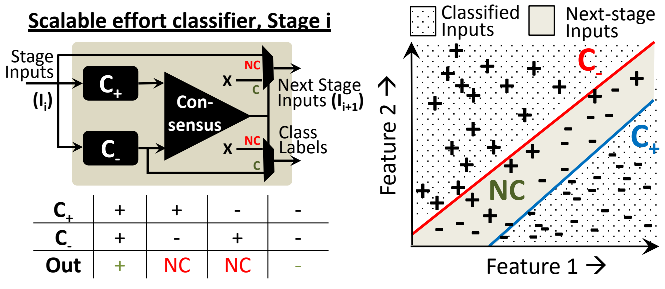
3.1 Structure of Classifier Stages

First, we consider the case of a binary classification algorithm with



Input
Stage 1
Stage 2
Stage N
Class Output
Easy inputs → low effort classification
Hard inputs → High effort classification

Figure 3: A scalable-effort classifier comprises a sequence of decision models, which grow progressively complex.



C_+	+	+	-	-
C_-	+	-	+	-
Out	+	NC	NC	-

(a) (b)

Figure 4: (a) Each stage comprises of two biased classifiers and a consensus module. (b) The stage produces labels or next-stage inputs depending on the consensus.

two possible class outcomes + and -. In such a scenario, each stage comprises two biased classifiers at the core; biased classifiers are those that are trained to detect one particular class with high accuracy. For instance, if a classifier is biased towards class + (denoted by C_+), it frequently mispredicts inputs from class - but seldom from class +. Besides the biased classifiers, the stage also contains a consensus module, which determines the confidence value of the class label assigned to every test instance. Fig. 4(a) shows the block diagram of a classifier stage. The consensus module of the i^{th} stage makes use of the output from the two biased classifiers to produce either the class labels or inputs to the next stage (denoted by I_{i+1}). This decision is made based on the following two criteria:

1. If the biased classifiers predict the same class *i.e.*, ++ or - -, then the corresponding label *i.e.*, + or - is produced as output.
2. If the biased classifiers produce no consensus (NC) *i.e.*, +- or -+, the input is deemed to be difficult to classify by the stage and the next-stage inputs are produced.

To better understand how each stage functions, consider the example shown for a binary SVM in Fig. 4(b). The two biased classifiers used (*i.e.*, C_+ and C_-) are linear SVMs, which are computationally efficient. Observe how the decision boundaries for the two classifiers are located such that they do not misclassify instances from the class towards which they are biased. For all input test instances that lie in the hatched region, both biased classifiers provide identical class labels (*i.e.*, consensus). However, there is no consensus on input instances that lie in the grayed-out region. Test instances in this latter region are thus passed on as inputs to the next stage.

Since the end-to-end scalable-effort classifier comprises many such individual stages in a sequence, the following two factors determine its the overall runtime and accuracy: (1) the number of connected stages and (2) the fraction of training data that is processed by each stage. These factors present an intertwined tradeoff in the design of the scalable-effort classifier, which we explain next.

3.2 Runtime and Accuracy Optimization

The consensus operation and method of biasing component classifiers in each stage directly control the number of stages and fraction of training data processed by each stage. In this section, we describe their design. In order to better understand the implications of these parameters, we first provide some mathematical insight that goes into the selection of each classifier stage.

For every stage i, with cost γ_i per instance, let I_i be the fraction of inputs that reach stage that stage. If γ_{i+1} is the cost per instance of the next stage, then the following condition should be satisfied to admit stage i into the sequence:

$$\gamma_i \cdot (I_i - I_{i+1}) + \gamma_{i+1} \cdot I_{i+1} < \gamma_{i+1} \cdot I_i \qquad (1)$$

The left-hand side in the above equation represents the cost when the stage is present, which is given by the sum of the costs incurred due to the fraction of inputs that the stage classifies (*i.e.* $I_i - I_{i+1}$) and the costs incurred by the next stage due to the fraction of inputs that the stage does not classify (*i.e.*, I_{i+1}). This cost should be lower than the cost that would be incurred if all I_i instances were processed by the next stage [*i.e.*, the right-hand side of Eq. (1)].

In the SVM example presented in the previous section, we described how identical labels from the two biased classifiers imply consensus and contradicting labels mean NC. In reality, however, the component classifiers produce labels based on the class probabilities associated with the labels. This allows us to design a slightly different consensus measure (or confidence value) called the consensus threshold, which controls the number of instances processed by a stage. Further, the cost associated with a stage can be modulated depending on the method of biasing the component classifiers. We describe these two design parameters next.

3.2.1 *Choice of Consensus Threshold*

Since the biased classifiers produce class probabilities, we combine the component classifier outputs over a continuum to either relax or tighten the consensus operation. To help us achieve this flexibility, we employ a tweakable parameter called the *consensus threshold* (denoted by δ). Fig. 5 illustrates the impact of different choices of δ for a stage; for larger values, the fraction of the input examples classified by a stage diminishes and vice versa. For negative values of δ, inputs can be labeled by a stage even if the classifiers disagree on the individual class assignments provided their confidence in the contradictory predictions is jointly greater than δ. Thus, δ directly controls the fraction of inputs classified by a stage. To achieve computational efficiency, we optimize for the value of δ at training time such that it minimizes the total number of misclassifications.

3.2.2 *Biasing the Component Classifiers*

In Eq. (1), while δ controls the number of input instances processed by a stage, the method of biasing the component classifiers controls the computational cost. Observe that the total cost of each stage is the sum of the costs associated with the two biased classifiers. The following options are available to design the biased component classifiers:

- **Asymmetric weighting:** We bias classifiers by assigning misclassification penalties to training instances depending on the associated class labels. For instance, while buliding C_+, we assign higher weights to instances from the + class, which encourages them to be classified correctly at the cost of misclassifying instances from the - class.
- **Resampling and sub-sampling:** To bias a classifier towards a particular class, we generate additional examples in that class by adding some uniform noise to the existing instances or sub-sampling instances from the opposite class. This provides a way of implicitly weighting the instances.
- **Tweaking algorithmic knobs:** Many classification algorithms provide parameters that control their complexity and bias. Some examples include changing the kernel function from a linear to a non-linear function in an SVM and the number of neurons and layers in a neural-network.

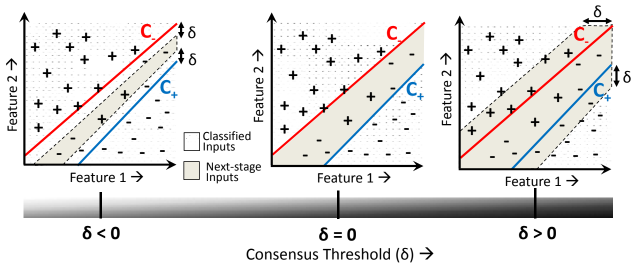
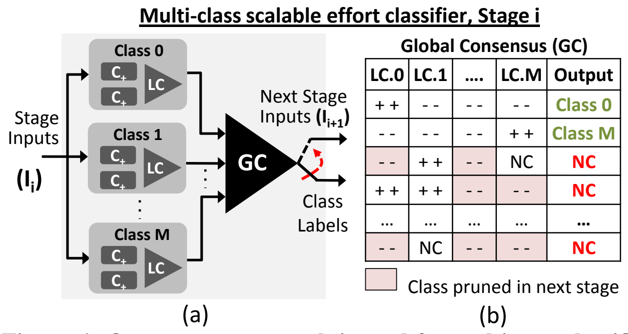


Figure 5: δ controls the fraction of inputs classified by a stage.


Multi-class scalable effort classifier, Stage i
Stage Inputs
$(\mathbf{I_i})$
Class 0
Class 1
Class M
C_+
LC
GC
Next Stage Inputs $(\mathbf{I_{i+1}})$
Class Labels
Global Consensus (GC)
Class pruned in next stage
(a)
(b)

LC.0	LC.1	....	LC.M	Output
+ +	- -	- -	- -	Class 0
- -	- -	- -	+ +	Class M
▨ - -	+ +	▨ - -	NC	NC
+ +	+ +	▨ - -	▨ - -	NC
...	...	...	...	...
▨ - -	NC	▨ - -	▨ - -	NC

▨ Class pruned in next stage

Figure 6: One *vs.* rest approach is used for multi-way classification. GC can prune some classes in the next stage.

Most algorithmic knobs significantly impact the classifier complexity but provide less control over biasing. Thus, we make limited use of this approach. On the other hand, the first two approaches above are complementary and provide much more control over biasing. Thus, we employ them jointly in our design.

3.3 Multi-way Scalable-effort Classifiers

We extend our approach to multi-class problems by employing a well-known strategy called one *vs.* rest classification, which reduces the computation to multiple binary classifications. The strategy involves training one classifier per class, with samples from that class regarded as positive (*i.e.*, +) while the rest as negative (*i.e.*, -). At test time, the highest confidence values across multiple such one *vs.* rest classifiers determines the final class assignment.

Fig. 6(a) shows the design of a stage of a scalable-effort multi-way classifier. It comprises several binary classification units, each containing a pair of biased classifiers and a local consensus (LC) module similar to the one shown in Fig. 4(a). It also contains a global consensus (GC) module, which aggregates outputs from all LC modules in the stage. The functionality of GC is illustrated in Fig. 6(b). If there is positive consensus (*i.e.*, ++) in exactly one LC module, then the GC outputs a class label corresponding to the consenting binary-classification unit. If more than one LC module provides consensus, then the next stage is invoked.

Another feature of multi-way scalable-effort classifiers is *class pruning*, *i.e.*, even if a stage does not classify a given input, it can eliminate some of the classes from consideration in the next stage. Specifically, if there is no consensus in the GC module and if the LC output shows negative consensus (*i.e.*, - -) then binary classification units corresponding to that particular class need to be evaluated in subsequent stages. Thus, only class that produce positive consensus or NC are retained down the chain. This early class pruning leads to increased computational efficiency.

4. DESIGN METHODOLOGY

In this section, we describe the procedure for training and testing scalable-effort classifiers.

4.1 Training Scalable-effort Classifiers

Algorithm 1 shows the pseudocode for training. The process takes the original classification algorithm C_{orig}, training data D_{tr}, and number of classes M as input. It produces a scalable-effort version of the classifier C_{se} as output, which includes the biased classifiers $C_{+/-}$ and consensus thresholds δ for each stage.

First, we train C_{orig} on D_{tr} and obtain its cost γ_{orig} (line 1). Then, we iteratively train each stage of the scalable-effort classifier C_{stg} (lines 2-22). The algorithm terminates if a stage does not improve the overall gain G_{stg} beyond a certain threshold ϵ (line 3). Next, we describe the steps involved in designing each stage of C_{se}.

To compute C_{stg}, we initialize G_{stg} and complexity parameter λ_{stg} to $+\infty$ and $-\infty$, respectively (line 2). Then, we obtain $C_{+/-}$ (line 5). We follow-up by assigning the smallest value of δ that yields an accuracy of ~100% on D_{tr} to be the consensus threshold for the stage δ_{stg} (line 6). Once, we determine $C_{+/-}$ and δ_{stg} for all classes, we proceed to estimate the number of inputs classified by the stage ΔI_{stg} by iterating over D_{tr} (line 9-17). During this time, we compute LC and GC values for each instance in D_{tr} (lines 10-11). For any instance, if global consensus is achieved (line 12), we remove it from D_{tr} for subsequent stages and increment ΔI_{stg} by one (line 13). If not, we add a fractional value to ΔI_{stg}, which is proportional to the number of classes eliminated from consideration by the stage (line 15). After all instances in D_{tr} are exhausted, we compute G_{stg} as the difference between the improvement in efficiency for the inputs it classifies and the penalty it imposes on inputs that it passes on to the next stage (line 18). We admit the stage C_{stg} to the scalable-effort classifier chain C_{se} only if G_{stg} exceeds ϵ (line 19). Since instances that are classified by the stage are removed from D_{tr} used for subsequent stages, one or more classes may be exhausted. In this case, we terminate the construction of additional stages (line 20) and proceed to append the final stage (line 23). The complexity of the classifier is increased for subsequent stages (line 21).

Algorithm 1 Methodology to train scalable-effort classifiers

```
Input: Original classifier $C_{orig}$, training dataset $D_{tr}$, # classes M
Output: Scalable-effort classifier $C_{se}$ (incl. $\delta$ and $C_{+/-}$ $\forall$ stages)
 1: Train $C_{orig}$ using $D_{tr}$ and obtain classifier cost $\gamma_{orig}$
 2: initialize stage gain $G_{stg} = +\infty$, complexity param. $\lambda_{stg} = -\infty$,
    and allClassesPresent = true
 3: while ($G_{stg} > \epsilon$ and allClassesPresent) do
 4:   for currentClass :=1 to M do              // evaluate stage $C_{stg}$
 5:     Train $C_{+/-}$ biased towards currentClass using $D_{tr}$ and $\lambda_{stg}$
 6:     $\delta_{stg} \leftarrow$ minimum $\delta$ s.t. training accuracy = 100%
 7:   end for
 8:   initialize # input instances to stage $I_{stg}$ = # instances in $D_{tr}$
      and # instances classified by stage $\Delta I_{stg} = 0$
 9:   for each trainInstance $\in D_{tr}$ do       // compute $\Delta I_{stg}$ for $C_{stg}$
10:     Compute local consensus LC $\forall$ M classes
11:     Compute global consensus GC
12:     if GC $\leftarrow$ true then
13:       remove trainInstance from $\in D_{tr}$ and $\Delta I_{stg} \leftarrow \Delta I_{stg} + 1$
14:     else
15:       $\Delta I_{stg} \leftarrow \Delta I_{stg}$ + # negative LCs / M
16:     end if
17:   end for
18:   $G_{stg} = (\gamma_{orig} - \gamma_{stg}) \cdot \Delta I_{stg} - \gamma_{stg} \cdot (I_{stg} - \Delta I_{stg})$
19:   if $G_{stg} > \epsilon$ then admit stage $C_{stg}$ into $C_{se}$
20:   if any class is absent in $D_{tr}$ then allClassesPresent $\leftarrow$ false
21:   $\lambda_{stg}$ + +        // increase classifier complexity for next stage
22: end while
23: append $C_{orig}$ as the final stage of $C_{se}$
```

4.2 Testing Scalable-effort Classifiers

Algorithm 2 shows the pseudocode for testing. Given a test instance i_{test}, the process obtains the class label L_{test} for it using C_{se}. First, the list of possible outcomes is initialized to the set of all class labels (line 1). Each stage C_{stg} is invoked iteratively (lines 2-15) until the instance is classified (lines 2). In the worst case, C_{orig} is employed in the final stage to produce a class label (lines 3-4). In all other cases, the following steps are carried out. At each active stage, $C_{+/-}$ are invoked to obtain an estimate of LC (line 6) and GC (line 7). If global consensus is achieved, *i.e.*, one LC output is positive and the rest are negative (lines 8-10), then the instance is predicted to belong to the class with the highest LC value (line 9). If not, the list of active classes is pruned by removing the classes for which LC is negative (line 11). Subsequent stages are then invoked with the reduced set of possible outcomes (line 14).

In summary, C_{se} implicity distinguishes between inputs that are easy and hard to classify. Thus, it improves the overall efficiency of any given data-driven classification algorithm. Next, we describe our experimental setup, which helps us evaluate the performance of scalable-effort classifiers.

Algorithm 2 Methodology to test scalable-effort classifiers

```
Input: Test instance $i_{test}$, scalable-effort classifier $C_{se}$, # stages $N_{se}$
      in $C_{se}$, and # possible classes M
Output: Class label $L_{test}$
 1: initialize possibleClassesList = {1,2,...,M}, currentStage = 1,
    and instanceClassified = false
 2: while instanceClassified = false do
 3:   if currentStage = $N_{se}$ then                    // apply $C_{se}$ to $i_{test}$
 4:     $L_{test} \leftarrow C_{se}$ [$i_{test}$]; instanceClassified ← true
 5:   else
 6:     Compute local consensus LC ∀ M classes
 7:     Compute global consensus GC
 8:     if GC ← true then              // global consensus achieved
 9:       $L_{test} \leftarrow$ label ∈ max (LC); instanceClassified ← true
10:     else
11:       ∀ LC = -1, delete labels from possibleClassesList
12:     end if
13:   end if
14:   currentStage ← currentStage + 1
15: end while
```

5. EXPERIMENTAL METHODOLOGY

Using scalable-effort classification, we demonstrate improvements in both software runtime and hardware energy consumption.

Application benchmarks. Table 1 shows the benchmarks and datasets that we use in our experiments. We evaluated 8 applications with over 9000 features and up to 10 classes. Between them, they utilize three common supervised machine-learning algorithms, namely the SVM, neural networks, and decision trees (J48 algo.).

Table 1: Application benchmarks used in our experiments

Algorithm	Application	Dataset [17]	Features / Classes
Support-vector machines	Handwriting reco.	MNIST [2]	784 / 10
	Human activity reco.	Smartphones	561 / 6
	Eye detection	YUV faces	512 / 2
	Text classification	Reuters	9947 / 2
Neural networks	Enzyme classification	Protein	356 / 3
	Census data analysis	Adult	114 / 2
Decision trees-J48	Game prediction	Connect-4	42 / 3
	Census data analysis	Adult	114 / 2

Energy and runtime evaluation. We implemented scalable-effort versions of each of the above in C#. We also integrated WEKA, a machine-learning toolkit, as a backend to our software [18]. This helped us rapidly train and evaluate different component classifiers. We measured runtime for the applications using performance counters on a commodity Intel Core i5 notebook with a 2.5 GHz processor and 8 GB of RAM. For the energy measurements, we implemented each classifier as an accelerator at the register-transfer logic (RTL) level to the many-core machine-learning architecture described in [11]. We used Synopsys design compiler to synthesize the integrated design to a 45 nm SOI process from IBM. Finally, we used Synopsys power compiler to estimate the energy consumption at the gate level.

6. RESULTS

In this section, we present experimental results that demonstrate the benefits of our approach.

6.1 Energy and Runtime Improvement

Fig. 7 shows the normalized improvement in efficiency with scalable effort classifiers designed to yield the same classification accuracy as the single-stage classifier (which forms the baseline) for all applications. We quantify efficiency in terms of three metrics: (i) average number of operations (or computations) per input (OPS), (ii) energy of hardware implementation, and (iii) execution time of software. We observe that scalable effort classifiers provide between 1.2×-9.8× (geometric mean: 2.79×) improvement in average OPS/input compared to the baseline. Note that the benefits vary depending on the fraction of hard-to-classify inputs in the dataset and the complexity of the classifier stages. For instance, the CONNECT application, in which we obtain the least improvement, filters only 25% of the inputs, while the complexity of the stages amount to 10% of the original classifier. On the other extreme, the EYES application classifies 90% of the inputs at a cost of 0.2% of the baseline. In the case of hardware and software implementations, the reduction in OPS/input translates on an average to 2.3× and 1.5× improvement in energy and runtime, respectively. While still substantial, due to the control and memory overheads involved, the benefits in energy and runtime for some applications are lower than that in OPS/input. In particular, the impact of implementation overheads is pronounced in the case of applications with smaller feature sizes and datasets.

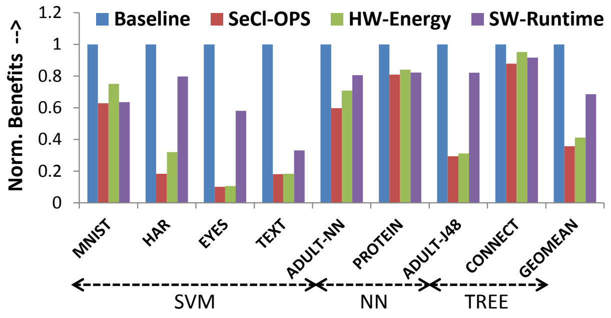


Figure 7: Improvement in average OPS/input, energy, and runtime for different applications compared against the baseline.

6.2 Impact of Hard Inputs on Efficiency

In this section, we examine the impact of hard-to-classify inputs on the overall efficiency of the scalable-effort classifiers. Towards this end, we identify inputs that are closer to the decision boundary of the original classifier and vary their proportion in the test dataset. Fig. 8(a) shows the normalized OPS required for different fractions of hardware inputs for three applications. Naturally, as the fraction of hard inputs increases, the benefits of scalable-effort execution are lowered. In fact, when the fraction increases beyond a certain level, scalable-effort classifiers become inefficient depending on the application and the complexity of the classifier stages. Fig. 8(b) shows the normalized complexity of the corresponding classifier stages. In the case of EYES, where the stage complexity is only 0.2%, scalable-effort design is desirable even when more than 99% of the inputs are hard [dashed vertical line in Fig. 8(a)]. As the stage complexity increases to 10% and 26%, as in the case of CONNECT and ADULT-NN, the break-even point occurs earlier at 85% and 72% of hard inputs respectively. The default fraction of hard inputs in these applications are also marked in Fig. 8(a), which corresponds to the benefits reported in Sec. 6.1.

6.3 Optimizing the No. of Classifier Stages

Choosing the right number of stages is critical to the efficiency of the scalable-effort architecture. We study the impact of this choice by varying the number of stages for the ADULT-J48 application.

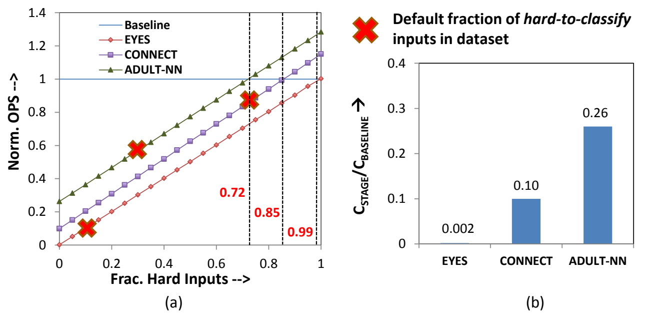


Figure 8: (a) Benefits are lowered when there are more hard-to-classify inputs. (b) Classifier complexity varies across apps.

The normalized OPS of the overall classifier split amongst each stage is shown in Fig. 9(a) – the original classifier has a stage count of one. When the number of stages is increased to 2, we see a large drop in total OPS, as the stage adds a small overhead, while significantly reducing the OPS contributed by the final stage. As we add a 3rd stage, we observe only a slight improvement. Though the stage decreases the number of final stage OPS, its added complexity nearly balances the reduction. Ultimately, the 4th stage is unfavorable as it increases the overall OPS. To gain additional insight, consider the normalized stage complexity and the fraction of inputs classified at each stage shown for the 4-stage classifier in Fig. 9(b). As expected, stage 1 is quite simple (5% complexity) and classifies a disproportionately large number (53%) of inputs. Stage 2 is balanced, with 24% complexity and 29% classification rate. The trade-off is reversed for the third stage, whose complexity is 53%, but classifies only 18% of additional inputs. This behavior is also reflected in the *gain* of each stage quantified by our design methodology in Fig. 9(b).

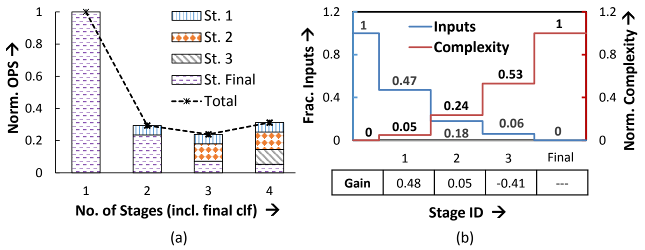


Figure 9: Normalized reduction in OPS with different number of classifier stages for ADULT-J48 application

6.4 Efficiency-Accuracy Tradeoff using δ

The consensus threshold δ provides us a powerful knob to trade accuracy for efficiency. Fig. 10 shows the variation in the normalized energy and accuracy of the scalable-effort classifier with different values of δ for two applications. In the case of MNIST, when $\delta = 0$, the accuracy is $\sim 5\%$ lower than the baseline, with over $10\times$ improvement in energy. To reach the same level of accuracy, δ should be increased at the cost of higher energy consumption, since now more inputs will reach the final classifier stage. Decreasing δ improves efficiency, but further degrades accuracy. For the EYES application, we find that even when $\delta = 0$, the accuracy is on par with the original classifier and has $3.5\times$ lower energy. Therefore, if we lower δ to -0.5, energy efficiency increases to $9\times$ with minimal loss in accuracy. Further decreasing δ still leads to only a slight degradation in accuracy; 3% lower for $30\times$ energy improvement for $\delta = -1$. Thus, the efficiency and accuracy of scalable-effort classifiers is a strong function of δ, which can be easily adjusted at runtime to an appropriate value.

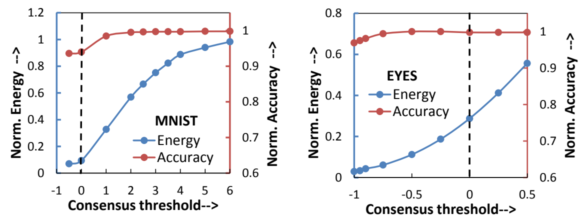


Figure 10: Energy *v.s.* accuracy trade-off by modulating consensus threshold

7. CONCLUSION

Supervised machine-learning algorithms play a central role in the realization of various prominent applications and place significant demand on the computational capabilities of modern computing platforms. In this work, we identify a new opportunity to optimize machine learning classifiers by exploiting the significant variability in the inherent classification difficulty of its inputs. Based on the above insight, we propose the concept of scalable effort classifiers, or classifiers that dynamically scale their effort to match the complexity of the input being classified. We develop a systematic methodology to design scalable effort versions for any given classifier and training dataset. We achieve this by combining biased versions of the classifier that progressively grow in complexity and accuracy. The sclable-effort classifier is equipped to implicitly modulate the number of stages used for classification based on the input, thereby achieving scalable effort execution. To quantify the potential of scalable effort classification, we build scalable effort versions of 8 recognition and computer vision applications, utilizing 3 popular machine learning classifiers. Our experiments demonstrate 2.79X reduction in average OPS per input, which translates to 2.3X and 1.5X improvement in energy and runtime over well-optimized hardware and software implementations of the applications.

8. REFERENCES

[1] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Intel Tech. Magazine*, 9(2):1–10, Feb. 2005.
[2] L. Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Proc. Magazine*, 29(6):141–142, Nov. 2012.
[3] Y.-T. Lin *et al.* Low-power variable-length fast fourier transform processor. *Proc. IEEE Computers and Digital Techniques*, 152(4):499–506, Jul. 2005.
[4] H. Kaul *et al.* A 1.45 GHz 52-to-162 GFLOPS/W variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm CMOS. In *Int. Solid-State Circuits Conf. Dig. Tech. Papers*, pages 182–184, Feb. 2012.
[5] V. Chippa *et al.* Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Design Autom. Conf.*, pages 555 –560, 2010.
[6] V. Chippa *et al.* Dynamic effort scaling: Managing the quality-efficiency tradeoff. In *Design Autom. Conf.*, pages 603–608, June 2011.
[7] H. Esmaeilzadeh *et al.* Architecture support for disciplined approximate programming. In *Proc. Int. Conf Architectural Support for Programming Languages and Operating Systems*, pages 301–312, Mar. 2012.
[8] S. Sidiroglou-Douskos et. al. Managing performance vs. accuracy trade-offs with loop perforation. In *Proc. ACM SIGSOFT symposium '11*, pages 124–134.
[9] S. Narayanan *et al.* Scalable stochastic processors. In *Proc. Design Automation and Test in Europe*, pages 335–338, 2010.
[10] K. Palem *et. al.* Sustaining moore's law in embedded computing through probabilistic and approximate design: Retrospects and prospects. In *Proc. CASES*, CASES '09, 2009.
[11] S. Venkataramani et al. Quality programmable vector processors for approximate computing. In *Proc. MICRO*, pages 1–12, 2013.
[12] R. E. Schapire. The boosting approach to machine learning: An overview. *Lect. Notes in Statistics: Nonlinear Estim. and Classification*, 171:149–171, 2003.
[13] J. Gama and P. Brazdil. Cascade generalization. *J. Machine Learning*, 41(3):315–343, Dec. 2000.
[14] P. Viola *et al.* Rapid object detection using a boosted cascade of simple features. In *Proc. Conf. Comput. Vision & Pattern Reco.*, pages 511–518, Dec. 2001.
[15] D. Hefenbrock *et al.* Accelerating Viola-Jones face detection to FPGA-level using GPUs. In *Symp. Field-Programmable Custom Computing Machines*, pages 11–18, May. 2010.
[16] C. Zhang and P. Viola. Multiple-instance pruning for learning efficient cascade detectors. In *Proc. Neural Info. Processing Syst.*, pages 1681–1888, Dec. 2008.
[17] K. Bache and M. Lichman. UCI machine learning repository, 2013.
[18] M. Hall *et al.* The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, Jun. 2009.