Combining Multiple Classifiers for Faster Optical Character Recognition

Kumar Chellapilla, Michael Shilman, Patrice Simard

Microsoft Research, One Microsoft Way, Redmond, WA, USA 98052
{kumarc, shilman, patrice}@microsoft.com
http://research.microsoft.com/dpu/

Abstract. Traditional approaches to combining classifiers attempt to improve classification accuracy at the cost of increased processing. They may be viewed as providing an accuracy-speed trade-off: higher accuracy for lower speed. In this paper we present a novel approach to combining multiple classifiers to solve the inverse problem of significantly improving classification speeds at the cost of slightly reduced classification accuracy. We propose a cascade architecture for combining classifiers and cast the process of building such a cascade as a search and optimization problem. We present two algorithms based on steepest-descent and dynamic programming for producing approximate solutions fast. We also present a simulated annealing algorithm and a depth-first-search algorithm for finding optimal solutions. Results on handwritten optical character recognition indicate that a) a speedup of 4-9 times is possible with no increase in error and b) speedups of up to 15 times are possible when twice as many errors can be tolerated.

1 Introduction

Given a classification problem, several approaches exist for building a classifier using machine learning. Different machine learning algorithms produce different classifiers, usually with different classification errors. Several studies have pursued the goal of finding a machine learning algorithm that can solve any classification task with the highest (generalization) accuracy. The quest for such a super classifier and a universal machine learning algorithm for training its architecture still remain elusive. However, its absence has produced several comparable alternatives. This variety has been a boon to approaches that attempt to build better classifiers through combination. Such combination approaches most commonly utilize not only the classification outputs but also the classification confidences returned by each classifier.

Recent investigations [1-6] have concentrated on combining two or more classifiers for improved classification accuracy. The intuition for why such an approach can work lies in the observation that the classification errors produced by radically different classifiers have low correlation [3]. Further, one also observes that the more confident a classifier, the more likely that it is correct and vice versa. When more than two

classifiers are available the combination alternatives become much more interesting with a greater potential for producing better qualifiers through combination.

Today's mobile electronic devices such as cell phones and digital cameras are capable of acquiring images at a sufficiently high resolution (3 MPix) to facilitate OCR of text in these images. There is a simultaneous explosion of software applications targeting these devices that can read and translate words in documents, traffic signs, restaurant menus, travel guides, etc. Given the low processing power of these devices, it is desirable to have high speed OCR systems that can be used on these machines. Though speed is a bottle neck, memory appears to be more freely available as these devices simultaneously target multimedia applications.

In this paper, we investigate the inverse problem wherein classification speed is of interest and we are willing to accept slightly reduced classification accuracy in order to achieve significant speedup. We address the scenario where a set of pre-trained classifiers are available for combination, and we wish to produce various speed-error trade-offs for several different scenarios. For simplicity we assume that retraining during combination is not an option. We present an approach to build and combine classifiers that can significantly improve classification speeds with a pre-specified maximum (usually small) drop in classification accuracy. Section 2 briefly reviews existing classifier combination approaches both for accuracy and speed. The new approach and the underlying optimization problem are presented in Section 3 along with four algorithms for building such cascades. Experimental results are presented in Section 4 and we conclude in Section 5.

2 Background

Combining classifiers for improved optical character recognition (OCR) accuracy is a well studied problem. Simple classifier fusion methods such as minimum, maximum, average, median, and majority voting have recently been studied both theoretically [1] and empirically [3]. Rather than using such simple static rules, a combining classifier can be trained to takes the outputs from two or more classifiers as input and produce a combined output that better models the class posterior probabilities or likelihoods. Such approaches based on learning have greater potential for producing larger improvements in accuracy [2]. Successful applications of classifier combinations include combining fingerprint matches, face and voice recognition and document processing [3-7]. We emphasize that all of these approaches focus on accuracy and invariably result in slower classifiers. The combined classifier is 2-20 times slower with the number of errors dropping by 18%-63% [4-5].

Combining classifiers for speed provides a different approach to improving classifiers. Such sequential combination is commonly addressed to improve not only classification accuracy, but also classification speed. Typically, the quickest classifiers are used first followed by slower (and frequently more accurate) classifiers [8]. Boosting is one such ensemble learning algorithm that sequentially adds weak classifiers to build a strong classifier. Using early stopping during sequential classifier combination

can not only produce speed benefits, but also acts as a regularization technique to improve generalization [9].

3 Combining classifiers for speed

In this paper we investigate combinations of a class of convolutional neural network classifiers [7]. These networks have two layers of convolutional nodes followed by a hidden layer and an output layer. Such a network achieved the best known error rate of 0.4% [7] for handwritten digit recognition (MNIST). It had 5 nodes in the first convolutional layer, 50 nodes in the second convolutional layer, 100 hidden nodes, and 10 output nodes (one for each digit 0-9). It can process about 250-300 chars/sec on a P4 3GHz machine. The network performs over half a million operations per classification. This network when used for recognizing documents would take over 15 seconds per page and would be too slow.

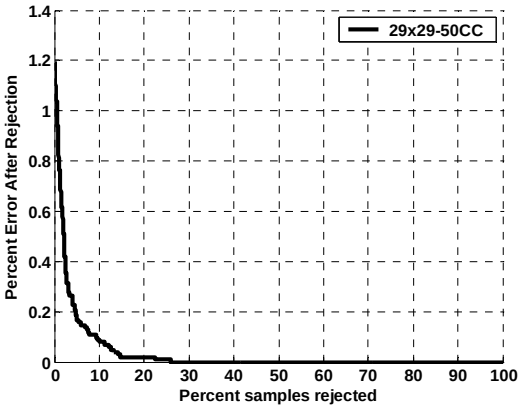

Fig. 1. Rejection curve for a convolutional neural network on MNIST data.

The factors contributing to the computation in the classifier are a) the input resolution, b) size of the convolutional layers, c) number of hidden nodes, and d) the number of output classes. The last parameter is defined by the problem and usually cannot be changed. For the remaining settings, smaller values produce faster but less accurate networks [7]. Further, it is well known that one observes a rate of diminishing returns in accuracy as complexity is increased.

The convolutional NN is trained through backpropagation using cross-entropy and learns to predict class probabilities [7]. The output for each class lies in [0,1] and can be used as a confidence measure to reject samples that would be incorrectly classified. Figure 1 presents the rejection curve for a convolutional NN with 50 hidden nodes. The rejection curve is monotonically decreasing indicating that the higher the confidence the less likely that the character will be misclassified. Even though the classifier only achieves an error rate of 1.25% on the MNIST data set, we can improve its error

rate to 0.1% or even 0% by rejecting 9% or 26% of the data, respectively. This trade-off is the key intuition behind the proposed cascade architecture.

3.1 Cascade of Classifiers

Figure 2 presents a cascade architecture for combining classifiers using a sequence of thresholds. Characters are processed by the cascade as follows: each input character image is initially presented to the first stage, S_1. If the classification output exceeds the first stage's threshold, t_1, then it is absorbed by the first stage and processing stops. If not, then the sample is rejected (by the first stage) and is passed on to the next stage. This process is repeated till the sample gets absorbed by some stage or we reach the last stage, S_M, in the cascade. The last stage, S_M, has a threshold of $t_M = 0$ and is designed to absorb all characters that reach it. The label assigned to the input character is that assigned by the absorbing stage.

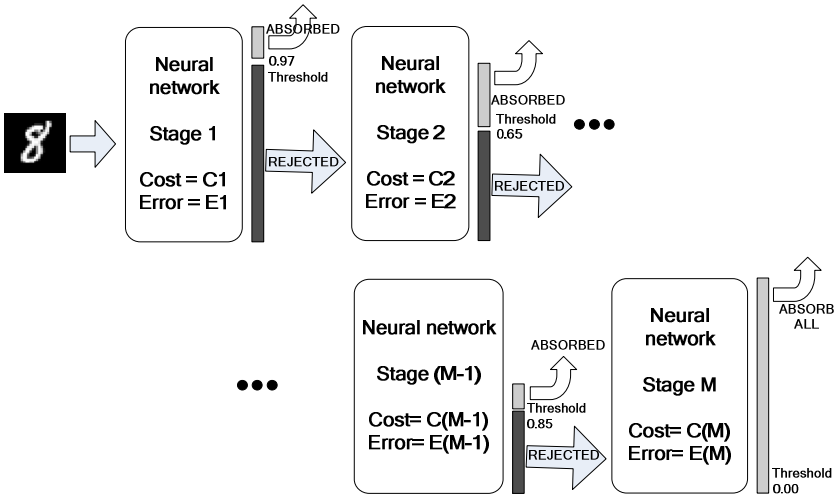

Fig. 2. Cascade of classifiers: Samples rejected at each stage are passed on to subsequent stages. Networks at the front of the cascade are fast and inaccurate while networks towards the end of the cascade are more accurate but slower.

The cascade architecture has several merits for improving processing speed. If faster less accurate nets are placed towards the front of the cascade and slower more accurate nets are placed towards the end of the cascade, one can dramatically reduce the expected processing times for input characters. The speedup and error rate of the cascade is determined by the costs, errors, and thresholds of each stage. Lower cost implies a faster stage.

3.2 Optimization problem

In this paper, we study such cascades with the stages arranged such that costs monotonically increase from left to right. Mathematically,

$$C_1 \leq C_2 \leq \ldots \leq C_M \quad (1)$$

where M is the number of stages and C_i is the cost of the i-th stage. Given the ordering, the costs are usually normalized such that $C_1 = 1.0$. Unlike the costs the errors are not expected to be monotonically decreasing as we move through the cascade (see Tables 1 and 2). Given this architecture, the goal is to find the fastest cascade with an error rate less than a pre-defined value $e_{\max}$. Though the cascade can be used to improve the error rate of the best classifier, in this paper, $e_{\max}$ will be defined to be larger than the error incurred by the best single network in the cascade. The search space of solutions is $S = \{t_1\}\times\{t_2\}\times\ldots\times\{t_M\}$, where $\{t_i\}$ is the set of all thresholds for stage i. The optimal threshold vector $T_* = [t_{1*}, t_{2*}, \ldots, t_{M*}]$ is given by

$$T_* = \arg\min \{C(T) \mid T \in S, e(T) \leq e_{\max}\} \quad (2)$$

The optimum cost is $C(T_*)$ and the corresponding speedup is $C_M/C(T_*)$. During optimization a set of input samples, $\{x_i\}$, is used to evaluate the cost, $C(T)$, and error rate, $e(T)$, of the cascade for each candidate threshold vector, T. If a stage rejects all samples (i.e., absorbs no samples) then it is considered to be pruned from the cascade and adds no cost to the cascade. Note that this problem formulation allows for some of the networks to be completely dropped from the cascade. It is easy to see that $C(T_*)$ can be no lower than C_1. At the other extreme, the maximum possible expected cost per input sample is given by

$$\max C(T) = (NC_1 + (N-1)C_2 + \ldots + (N-M)C_M)/N \quad (3)$$

For $N >> M$, we get

$$\max C(T) = C_1 + C_2 + \ldots + C_M \quad (4)$$

Four different algorithms are presented for finding solutions to this problem, namely steepest descent (SD), dynamic programming (DP), simulated annealing (SA), and depth-first-search (DFS). While the SD and DP algorithms presented here are designed for generating approximate solutions fast, the SA and DFS are designed to find the optimal solution. The SA algorithm is stochastic in nature and guarantees asymptotic convergence to the optimal solution. However, at any finite number of iterations, the best solution may only be approximate. Brief descriptions of the optimization algorithms are presented below:

Steepest descent

The algorithm is initialized with $T_0 = [1,1,\ldots,1,0]$, i.e., every stage rejects all samples except for the last stage that absorbs them. Such a solution satisfies the $e_{\max}$ constraint and has a cost $C(T_0) = C_M$. During each iteration, the change in cost ΔC_i, $i = 1,2,\ldots,M$ and the change in error Δe_i, $i = 1,2,\ldots,M$ are computed by lowering each threshold t_i to the next possible value while keeping all other thresholds the same. Note that due to

the monotonic increase in costs over the cascade $\Delta C_i < 0$, $i = 1,2,\ldots,M$. If the error decreases for any i (i.e., $\Delta e_i < 0$), the best such i with the lowest Δe_i is selected for update. If all $\Delta e_i > 0$, the i with the lowest cost change per unit error change = $-\Delta C_i/\Delta e_i$ is selected for update. The selected threshold is updated to the next lower value and the process is iterated. Search is terminated when the best possible update puts the error above $e_{\max}$. The steepest descent algorithm is sensitive to local optima and is used only as a baseline for comparing algorithms. The algorithm is simple and very fast. Each update takes at most $O(M)$ evaluations and there are at most MN evaluations. Due to the incremental updates to the thresholds during successive evaluations, the cost and error evaluation can be done very efficiently by remembering which samples were absorbed at each of the stages and which ones are affected by the threshold update. The total running time is bounded by $O(M^2N)$.

Dynamic programming

The dynamic programming algorithm presented here builds a cascade by iteratively adding new stages. It starts with a two stage cascade containing only the first and last stages, S_1, and S_M, respectively. Note that such a two stage cascade has at most N possible threshold vectors. Each threshold vector represents a unique solution with a different second last stage threshold. Let these be represented as N paths of length one, each ending at a unique threshold. Each of these N paths is evaluated. Now consider inserting stage S_2 between S_1, and S_M. Each of the existing N paths can be extended in N possible ways through S_2. All such N^2 extensions are evaluated. For each threshold in S_2, the best path extension (among the N^2 possible extensions) is chosen and retained. This results in N paths of length 2 each passing through a different threshold in S_2 and representing a different cascade with three stages. This process of adding a stage is repeated $M - 2$ times to obtain a set of N paths representing cascades with M stages. The best path among these remaining N paths is picked as the final solution. The algorithm is not guaranteed to find the optimal solution because only N paths are retained during each iteration. The running time is $O(MN^2)$.

Simulated annealing

A simulated annealing algorithm is presented that simultaneously optimizes all thresholds in the cascade of M stages. As in the case of steepest descent, the initial solution is T_0. At any given temperature, λ, each threshold, t_i, is updated to a neighbor that is η = round($G(0, \lambda)$) steps away, where $G(0, \lambda)$ is a zero mean Gaussian random variable with standard deviation λ. Note that η can be positive or negative. Any thresholds that fall outside the valid limits (threshold indices: 1–N or threshold values 0–1) are reset to the limit violated. The initial temperature was set to N and the Metropolis algorithm was used to accept better solutions. Further, any solutions that didn't satisfy the $e_{\max}$ criterion were also rejected during the updates. The temperature was continuously annealed down to 1.0 with a maximum of one million evaluations (= E). The running time is $O(EM)$.

Depth first search

The above three algorithms do not guarantee finding the optimal solution after a finite amount of computation. A simple depth-first-search was used to search through all possible threshold settings. Every possible cascade with 2 to M stages can be represented as a node in a tree of maximum depth $M - 2$. Nodes at depth d represent cascades of length $d - 2$. The running time is $O(N^{M-1})$, but extensive pruning is possible.

3.3 Experiments

Experiments were conducted with the MNIST and FUGU character datasets. The MNIST dataset consists of 60,000 hand written digits uniformly distributed over 0–9. The FUGU dataset contains a natural distribution of 925,702 Japanese kanji characters with up to 3 strokes (258 classes). For each dataset, 80% of the samples were used for training, 10% were used for validation (to determine when to stop training) and the remaining 10% were used for testing. The validation samples were also used to optimize the thresholds.

A total of 18 MNIST classifiers and 12 FUGU classifiers of varying sizes were trained. The parameters being varied were the input image size (5x5, 7x7, 9x9, 11x11, and 29x29), the number of convolution layers (2 layers or 0 layers), and the number of hidden nodes (50, 100, 200, and 300). Cascades of such trained networks were optimized using the above algorithms. Approximate running times for SD, DP, and SA, on these two problems were 0.25, 45, and 250 seconds, respectively. Due to the computationally intensive nature of DFS, DFS experiments were conducted only on a toy problem consisting of 4 MNIST classifiers and 5000 samples. The quality of the optimal solutions found using DFS are compared with the other algorithms.

Table 1. MNIST Results: Costs and Errors.

SN	Input	Arch.	Cost	Train%	Test%
1	5x5	0,0,50,10	1.00	26.52	27.48
2	5x5	0,0,100,10	1.22	25.06	26.81
3	5x5	0,0,200,10	1.72	25.36	26.76
4	5x5	0,0,300,10	2.24	25.57	27.07
5	7x7	0,0,50,10	1.04	8.82	9.06
6	7x7	0,0,100,10	1.35	8.17	8.38
7	7x7	0,0,200,10	1.99	8.29	8.82
8	7x7	0,0,300,10	2.61	8.04	8.36
9	9x9	0,0,50,10	1.14	4.49	4.65
10	9x9	0,0,100,10	1.53	4.13	4.20
11	9x9	0,0,200,10	2.36	4.03	4.01
12	9x9	0,0,300,10	3.17	3.78	3.89
13	11x11	0,0,50,10	1.26	3.63	3.72
14	11x11	0,0,300,10	3.84	2.35	2.27
15	29x29	0,0,50,10	3.39	3.43	3.36
16	29x29	0,0,300,10	16.3	1.82	1.81
17	29x29	5,50,50,10	23.0	1.20	1.25
18	29x29	5,50,300,10	41.6	1.10	1.19

4 Results

Training results for MNIST are presented in Table 1. The Train% is the error on the validation set used for stopping training and the Test% is the error on the test set. The best network (stage 18) had a validation error rate of 1.1% and a test error of 1.19%. On the other hand, the fastest network which was 41.6 times faster had an error rate of 26.52%! The validation set was used to optimize the 18-thresholds in the cascade for 11 values of e_{max} ranging from 1.1% (no extra error) to 2.2% (double the error). Figure 3 presents the speedup results obtained using SD, DP, and SA on the MNIST dataset. The speedup rapidly increases with increasing e_{max}. It is quite remarkable that even with no change in error, a 4x speedup is possible using the cascade architecture. The speedup quickly rises above 5x for an e_{max} of 1.2% and reaches 15x for an e_{max} of 2.2%. The curve for DP is wavy with certain lower error settings producing faster solutions. This is because DP only retains the best N paths during each iteration.

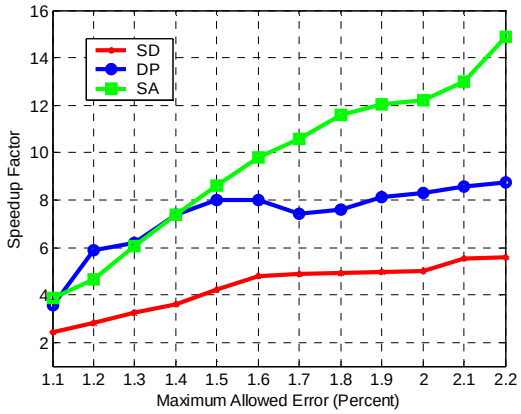

Figure 3. Speedup factor on MNIST.

Table 2 presents training results for FUGU. The best network (stage 120 has an error rate of 5.79% on the validation set. The error on the test set was 5.85%. On the other hand, the fastest network which is 17.61 times faster had an error rate of 52.20%. The validation set was used to optimize the 12-thresholds in the cascade for 11 values of e_{max} ranging from 5.79% (no extra error) to 11.58% (twice the error). Figure 4 presents the speedup results obtained using SD, DP, and SA on the FUGU dataset. As in the case of MNIST, the speedup rapidly increases with increasing e_{max}. It is interesting to note that using the cascade architecture, an almost 10x speedup is possible with no change in error. The speedup quickly rises to 12x for an emax of 8.95% (55% more errors) and reaches 14x for an emax of 11.58% (twice the number of errors). Over an order of magnitude speedup was possible with little or no change in error rate. We conjecture that the reason for higher speedup factors with FUGU is the natural distribution of characters and an overall larger error rate. Further, the lower slope on the FUGU dataset is due to the lower cost range among constituent networks.

At double the error rate, we come close to the maximum possible speedup factor of 17.61.

Table 3 presents results on the toy problem using the four algorithms. DFS finds optimal solutions within a few minutes. On the toy problem, the quality of solutions found using DP and SA closely match those found using DFS. However, further experiments are necessary to determine how well these results generalize to larger problems with more stages and larger validation sets.

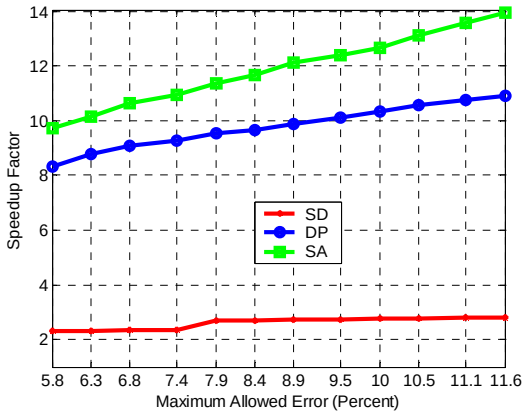

Figure 4. Speedup factor on FUGU.

Table 2. FUGU Results: Costs and Errors

SN	Input	Arch.	Cost	Train%	Test%
1	5x5	0,0,50,258	1.00	52.20	52.02
2	5x5	0,0,300,258	1.06	49.48	49.47
3	7x7	0,0,50,258	1.09	28.56	28.29
4	7x7	0,0,300,258	1.12	20.57	20.53
5	9x9	0,0,50,258	1.96	18.53	18.44
6	9x9	0,0,300,258	2.86	11.91	11.99
7	11x11	0,0,50,258	3.03	14.39	14.27
8	11x11	0,0,300,258	3.25	9.41	9.43
9	29x29	0,0,50,258	3.48	11.88	11.61
10	29x29	0,0,300,258	8.12	7.29	7.23
11	29x29	5,50,50,258	9.37	7.54	7.56
12	29x29	5,50,300,258	17.61	5.79	5.85

Table 3. Speedup comparisons against optimal speedup on a toy problem (4 classifiers, 5000 samples)

e_{max}	DFS (optimal)	SA	DP	SA
12.36%	5.1219	1.0250	5.1102	5.0493
15%	7.9754	1.0699	7.9556	7.9193
20%	20.5	1.6314	20.5	20.5

5 Conclusion

A cascade architecture for combining classifiers was presented along with four algorithms for optimization. Results on character recognition show that significant speed-ups can be obtained with little or no change in the error rate. The input to our optimizer is a set of rejection curves computed by potential classifiers (of any type) and the output is a set of thresholds (which potentially eliminate the useless classifiers). The optimizer is called off-line and its output yields a near speed-optimal combination classifier, ready for deployment. Future work will address scaling this approach to much larger data sets, larger cascade sizes, and re-training constituent classifiers in light of the role played by them in the cascade.

References

1. Ludmila IK, "A Theoretical Study on Six Classifier Fusion Strategies," IEEE Trans. On Pattern Analysis and Machine Intelligence, v. 24, No. 2, pp. 281-286, Feb. 2002.
2. RPW. Duin, "The Combining Classifier: To Train or Not to Train?" ICPR (2), pp. 765-770, 2002.
3. J Kittler, M Hatef, RPW Duin, and J Matas, "On Combining Classifiers," IEEE Trans. On Pat. Analysis and Machine Intel., Vol. 20, No. 3, Mar. 1998. Mar. 1998.
4. L Prevost, C Michel-Sendis, A Moises, L Oudot, and M Milgram, "Combining model-based and discriminative classifiers: application to handwritten character recognition," ICDAR'03. 2003.
5. H Hao, CL Liu, and H Sako, "Confidence evaluation for combining diverse classifiers," ICDAR'03, pp. 760-765, 3-6 Aug. 2003.
6. U. Bhattacharya and B. B. Chaudhuri, "A Majority Voting Scheme for Multiresolution Recognition of Handprinted Numerals," ICDAR'03, pp. 16-20, 3-6 Aug. 2003.
7. PY Simard, D Steinkraus, and J Platt, (2003) "Best Practice for Convolutional Neural Networks Applied to Visual Document Analysis," in ICDAR'03, pp. 958-962.
8. S Marinai, M Gori, G Soda, "Artificial Neural Networks for Document Analysis and Recognition," IEEE TPAMI, Vol. 27, No. 1, pp. 23-35.
9. T Zhang and B Yu, "Boosting with early stopping: Convergence and consistency," Annals of Statistics. vol. 33, no. 4, 1538–1579, 2005.