Effective Compressed Multi-function Convolutional Neural Network Model Selection Using A New Compensatory Genetic Algorithm-Based Method

Luna M. Zhang

BigBear, Inc.

Abstract

An important research problem is how to automatically and efficiently find the best Convolutional Neural Network (CNN) with both high classification performance and compact architecture with high training and prediction speeds, small power usage, and small memory size for Artificial Intelligence of Things (AIoT) applications. A"Single-function CNN" (SCNN) using one activation function, such as Google's Inception-V4 using ReLU, may not always be optimal for different image classification problems. A "Multi-function CNN" (MCNN), which uses different activation functions for different neurons, can outperform a SCNN. A "Compressed Multi-function CNN" (CMCNN) is made from a larger MCNN such that the number of convolutional layers is reduced. A new compensatory algorithm using a new genetic algorithm (GA) is created to find the best CMCNN with an ideal compensation between performance and architecture size. Simulations using the CIFAR10 dataset showed that the new compensatory algorithm could find CMCNNs that could outperform non-compressed MCNNs in terms of classification performance (F1-score), speed, power usage, and memory usage. Effective, fast, power-efficient, and memory-efficient CMCNNs based on other popular CNN architectures, such as a ResNet, will be developed and optimized for users to solve important AIoT application problems.

Introduction

Recently, deep learning techniques have been effectively used in various applications in computer vision, healthcare, etc. Convolutional Neural Networks (CNNs) are powerful techniques for image classification for various important real-world applications (LeCun et al. 2015; Krizhevsky et al. 2012; He et al. 2016; Esteva et al. 2017; Szegedy et al. 2015; Szegedy et al. 2017). Some examples are GoogLeNet, ResNets, DenseNets, Dual Path Networks, and Inception-V4 networks. Traditional CNNs tend to use the same activation function (typically ReLU) for all convolutional layers. For example, both ResNets (He et al. 2016) and the very deep Inception-v4 network (Szegedy et al. 2017) use the ReLU for their activation functions. However, Compared to smaller networks, larger networks would need more memory, longer training times (less power-efficient), longer prediction times, and more data in order to perform and generalize well. Therefore, it would be advantageous to try to make CNNs smaller in size for real-world applications. For example, it is useful to build a compact deep neural network (DNN) on an Internet-of-Things (IoT) device to ensure high DNN model inference accuracy, low inference latency, high throughput, and low power consumption (Zhang et al. 2019). A new bi-directional co-design approach was developed to optimize both DNN models and their deployment configurations on IoT devices on FPGAs to achieve high accuracy, high throughput and high energy efficiency (Zhang et al. 2019). The architecture-aware optimization framework was implemented to efficiently run compressed DNN to outperform all the state-of-the-art dense DNN execution frameworks such as TensorFlow Lite (Niu et al. 2019). A new context-adaptive binary arithmetic coder was made to compress deep neural networks with very high compression ratios across many different network architectures and datasets (Wiedemann et al. 2019).

Neural architecture search (NAS) is an important subfield of automated machine learning (AutoML). Much work has already been done in NAS and AutoML. For instance, a NAS method with optimal arithmetic bit length allocation and neural network pruning was able to search for the configurations with a computational complexity budget while maximizing the accuracy (Zur et al. 2019). A single-path NAS method, a novel differentiable NAS method for designing device-efficient ConvNets, was up to 5,000x faster compared to prior methods (Stamoulis et al. 2019). A new instance-ware NAS framework can search for a distribution of architectures in a multiple-objective NAS setting to get either higher accuracy with same latency or significant latency reduction without compromising accuracy against MobileNetV (Cheng et al. 2019). The NAS ensemble algorithm called AdaNAS could improve the performance of NAS models while having a similar parameter count as single large model (Macko et al. 2019). In particular, evolutionary approaches, such as Genetic Algorithms (GAs), have been used for NAS. For example, GA was used to optimize the number of layers and the number of neurons of each layer of a CNN that used one activation function (Sun et al. 2018; You et al. 2015). Also, a multi-objective GA

AAAI-2020 Workshop on Artificial Intelligence of Things.

was used for NAS by proposing a new algorithm called NSGA-Net (Lu et al. 2018). The results for both examples showed that using GA could be useful for optimizing a CNN's architecture and performance at the same time. The new evolutionary-neural hybrid agents outperformed both neural and evolutionary agents for text classification and image classification benchmarks (Maziarz et al. 2019).

Deep learning models like CNNs, such as Inception-V4, have a lot of parameters and very long training times. Some work was done in pruning CNNs to be smaller (Molchanov et al. 2016).

A MCNN using different activation functions can outperform a SCNN using one activation function (Zhang, 2018). In this paper, we newly designed a new GA to automatically optimize CMCNNs and select the best CMCNN that takes into account both the image classification performance and architecture size. The goal is to find the best CMCNN with high performance, small architecture size, and fast prediction time. Such an effective, fast, power-efficient, and memory-efficient CMCNN is useful for practical AIoT applications, such as AI chips, and mobile biomedical imaging.

Compressed Multi-function CNNs

It is useful to make CNN architecture smaller (i.e. removing layers) and making it "multi-function" can lead to better performance, faster speeds in training and prediction, lower energy consumption, and lower memory usage for a particular image classification problem. Let this new CNN be denoted as "Compressed Multi-function Inception-V4" (CMI).

Let "CMI_i" and "CI_i" mean that a CMI and a compressed Inception-V4 with ReLU have i Inception-A block(s), i + 1 Inception-B block(s), and i Inception-C block(s) for i = 1, 2, and 3. For CMI_1, the number of activation functions is 58, CMI_2 has 85 activation functions, and CMI_3 has 112. In Tables 1 and 2, "MI" and "I" mean that a MI and the original Inception-V4 with ReLU have 4 Inception-A, 7 Inception-B, and 3 Inception-C blocks. Stratified 3-fold cross validation was used to evaluate and compare the three CMI models, the MI model, the three compressed Inception-V4 models with ReLU, and the original Inception-V4 using multi-class classification metrics (i.e. training F1-score ($F1_{train}$), validation F1-scores ($F1_{valid}$), training times (T_{train}) in seconds, and classification testing times (T_{test}) in seconds. An activation function set {ReLU, SIG, TANH, ELU} was used to build all of the MCNNs. Each activation function is randomly chosen from this set for each convolutional block (CB). Table 1 shows the number of CBs, model sizes, and numbers of activation functions used for CMI_1, CMI_2, CMI_3, and MI.

Table 1: Comparison between Different Compressed Multi-function Inception-V4 Architectures and the Multi-function Inception-V4 Architecture

Model:	CMI_1	CMI_2	CMI_3	MI
No. CBs	58	85	112	149
Model Size (MB)	129	190	252	323
No. Functions	58	85	112	149

Table 2: Performance Comparison between Different Compressed Multi-function Inception-V4 Architectures and the Original Inception-V4 Architecture (stratified 3-fold cross validation, 120 epochs)

Model:	CMI_1	CMI_2	CMI_3	MI	I
$F1_{train}$	0.78	0.85	**0.86**	0.84	0.70
$F1_{test}$	0.76	0.82	**0.86**	0.83	0.68
$T_{train}(s)$	2229	3081	3978	5082	4815
$T_{test}(s)$	1.24	1.54	1.88	2.41	2.31

New compensatory CMCNN model selection algorithms using GA

CNN model selection with multi-objective optimization

We consider a four-objective optimization problem: maximizing performance, maximizing speed, minimizing energy usage, and minimizing memory usage by optimizing CMCNN's hyperparameters (i.e. numbers of layers, numbers of neurons in layers, and activation functions of neurons).

The energy E is directly proportional to the execution time T (Li 2008). If the number of convolutional layers of a CMCNN is reduced, then the execution time and energy consumption are also reduced. In addition, a CNN model size (i.e., memory usage) is reduced too. Thus, the four-objective optimization problem becomes a two-objective optimization problem: maximizing performance and minimizing the number of convolutional layers of a CMCNN.

A simple optimization function (the fitness function for GA) is defined as $\alpha = wF1_{train} + (1-w)(1-S)$ where w for $0 \leq w \leq 1$ is a weight, and S is a metric related to a property of a CMCNN architecture, such as a ratio of the model sizes between a CMCNN and a popular CNN. A user can choose a value for w.

New compensatory CMCNN model selection algorithms using GA

A MCNN has n convolutional layers; each convolutional layer is followed by $f \in \{f_1, f_2, ..., f_m\}$ where f_i is an activation function for $i = 1, 2, ..., m$. There are $m^n - m$ different MCNNs and m different traditional CNNs that use the same activation function for all neurons. Since there are too many different MCNNs, it is inefficient and unnecessary to test each one. Thus, we develop a new GA. A MCNN's activation functions can be represented by a string $[g_1 g_2 g_3 ..... g_n]$ where $g_i \in \{f_1, f_2, ..., f_m\}$ for $i = 1, 2, ..., n$. The new GA method has two operations: a newly defined mutation and a traditional crossover. For a given string $[a_1 a_2 a_3 ..... a_n]$ where $a_i \in \{f_1, f_2, ..., f_m\}$ for $i = 1, 2, ..., n$, a new mutation operation with a mutation point j for $j \in \{1, 2, 3, ..., n\}$ results in $[a_1 a_2 ... a_{j-1} c_j a_{j+1} ... a_n]$ where $c_j \in \{f_1, f_2, ..., f_m\}$ and $c_j \neq a_j$. For two parent strings $[a_1 a_2 a_3 ..... a_n]$ where $a_i \in \{f_1, f_2, ..., f_m\}$ and $[b_1 b_2 b_3 ..... b_n]$ where $b_i \in \{f_1, f_2, ..., f_m\}$ for $i = 1, 2, ..., n$, a crossover operation with a crossover point k for $k \in \{2, 3, ..., n-1\}$ results in two new child strings

$[a_1a_2...a_kb_{k+1}b_{k+2}...b_n]$ and $[b_1b_2...b_ka_{k+1}a_{k+2}...a_n]$. The new GA method is shown in Algorithm 1 where F is a fitness function.

Algorithm 1 A New GA for Optimizing MCNN Models

Input: initial population of N MCNN models where N is an even number, training data

Output: the best MCNN model

1: Create a population of an even number of MCNN models with randomly generated activation functions where each function has equal probability of being chosen.
2: Train all MCNN models in the population.
3: Compute F for each trained MCNN model in the population. Keep the top MCNN with the highest F.
4: Select pairs of MCNNs' function strings.
5: Perform crossover on each pair of MCNNs' function strings to generate new MCNNs' function strings.
6: Perform the newly defined mutation operations on the new MCNNs' activation function strings based on the mutation probability.
7: Create a new population that includes the newly created MCNNs' activation function strings and the parents.
8: Train all MCNN models in the new population.
9: Compute F for each trained MCNN model in the new population.
10: Repeat steps 4 to 9 for a generation until the maximum number of generations is reached.

A new CMCNN model selection method using a new GA is shown in Algorithm 2. The new GA using the new mutation operator for optimizing MCNN models is shown in Algorithm 2.

Algorithm 2 Compensatory CMCNN Model Selection Using Algorithm 1

Input: initial population of N MCNN models where N is an even number, training data

Output: the best CMCNN model

1: For each compressed architecture, create an initial population (N CMCNNs) for an even positive integer N.
2: For each compressed architecture, run Algorithm 1 to find the best CMCNN with the highest F. Calculate α for this best CMCNN.
3: Choose the overall best compensatory CMCNN with the highest α among all the best CMCNNs of different compressed architectures.
4: Use the overall best compensatory CMCNN for a real-world application.

Simulation results and performance analysis

Let "CLs" mean convolutional layers. Let "CM^n" mean that a CMCNN has n CLs with n activation functions. Let "CM^n_{GA}" mean that a CMCNN optimized by the new GA has n CLs with n activation functions. Let "M^m" mean that a non-compressed MCNN has m CLs with m activation functions for $m > n$. Then we define the model size ratio as S = (number of CLs)/m. A fitness function F is defined by $\alpha = wF1_{train} + (1-w)(1-S)$ for $w = 0.7$ for simulations. An activation function set {ReLU, SIG, TANH, ELU} was used; each function was randomly chosen from this set for each CL. CM^4_{GA}, CM^6_{GA}, and CM^8_{GA} were compared with M^{10}. Samples of the CIFAR10 data were used (Krizhevsky 2009). The population of the new GA has four randomly generated CNNs. The mutation probability is 1. Table 3 shows the numbers of convolutional layers, model sizes (in KB), S, and average training times (T_{train}) (in seconds) of one CNN model of CM^4_{GA}, CM^4, CM^6_{GA}, CM^6, CM^8_{GA}, CM^8, M^{10}_{GA}, and M^{10} for 25000 training data. $T_{predict}(s)$ is the total prediction time for 7000 testing data. Table 3 shows that CMCNNs use less memory, less training times (i.e., less power usage), less prediction times, and smaller numbers of CLs than M^{10} (a MCNN). CM^k_{GA} has longer training time than CM^k for $k = 4, 6, 8$, and M^{10}_{GA} has longer training time than M^{10}.

Table 3: Properties of different models

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
CLs	4	4	6	6	8	8	10	10
Size	454	454	507	507	648	648	815	815
S	0.4	0.4	0.6	0.6	0.8	0.8	1.0	1.0
T_{train}	7371	1256	9126	1472	9849	1578	10391	1652
$T_{predict}$	3.68	3.68	4.04	4.04	4.23	4.23	4.46	4.46

Simulation results

Algorithm 2 was implemented. For Tables 4-18, the performance results (training F1-score ($F1_{train}$), testing F1-score ($F1_{test}$), training α (Fit_{train}), and testing α (Fit_{test})) of the best models (those with the largest $F1_{train}$) were recorded. The largest value for each row is bolded in Tables 4-18. The new GA used many generations of training with new mutation operations and crossover operations. To better evaluate the new algorithm, two CIFAR10 data partitions were made based on the original 50,000 CIFAR10 training data (D[1], D[2], ..., D[50000]) and 10,000 CIFAR10 testing data (T[1], T[2], ..., T[10000]). The first data partition method generated N_{train} training data (D[1], D[2], ..., D[N_{train}]) and N_{test} testing data (T[1], T[2], ..., T[N_{test}]). The second data partition method generated K_{train} training data (D[$50001-K_{train}$], D[$50002-K_{train}$], ..., D[50000]) and K_{test} testing data (T[$10001-K_{test}$], T[$10002-K_{test}$], ..., T[10000]).

Simulation results using the first data partition method Simulation results using the first data partition method are shown in Tables 4-13.

Table 4: Model comparisons (N_{train} = 20000, N_{test} = 5000, 15 epochs, 5 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.852**	0.826	0.829	0.810	0.806	0.778	0.770	0.752
$F1_{test}$	0.737	0.707	**0.747**	0.742	0.746	0.714	0.717	0.714
Fit_{train}	**0.776**	0.758	0.700	0.687	0.624	0.605	0.539	0.526
Fit_{test}	**0.696**	0.675	0.643	0.639	0.582	0.560	0.502	0.500

Table 5: Model comparisons ($N_{train} = 25000$, $N_{test} = 6000$, 15 epochs, 5 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.846**	0.799	0.829	0.813	0.805	0.792	0.732	0.684
$F1_{test}$	0.734	0.711	**0.757**	0.741	0.743	0.730	0.700	0.654
Fit_{train}	**0.772**	0.739	0.700	0.689	0.624	0.614	0.512	0.479
Fit_{test}	**0.694**	0.678	0.650	0.639	0.580	0.571	0.490	0.458

Table 6: Model comparisons ($N_{train} = 12500$, $N_{test} = 5000$, 10 epochs, 10 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.842**	0.831	0.804	0.775	0.770	0.691	0.689	0.657
$F1_{test}$	0.697	0.685	**0.712**	0.690	0.693	0.637	0.640	0.615
Fit_{train}	**0.769**	0.761	0.683	0.663	0.599	0.544	0.482	0.460
Fit_{test}	**0.668**	0.660	0.618	0.603	0.545	0.506	0.448	0.431

Table 7: Model comparisons ($N_{train} = 15000$, $N_{test} = 6000$, 20 epochs, 10 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.899**	0.895	0.865	0.839	0.801	0.801	0.785	0.739
$F1_{test}$	0.694	0.695	**0.740**	0.722	0.709	0.709	0.710	0.681
Fit_{train}	**0.809**	0.807	0.726	0.707	0.621	0.621	0.550	0.517
Fit_{test}	0.666	**0.667**	0.638	0.625	0.556	0.556	0.497	0.477

Table 8: Model comparisons ($N_{train} = 20000$, $N_{test} = 7000$, 20 epochs, 15 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.879**	0.858	0.853	0.819	0.821	0.768	0.804	0.765
$F1_{test}$	0.727	0.722	**0.756**	0.731	0.740	0.710	0.741	0.712
Fit_{train}	**0.795**	0.781	0.717	0.693	0.634	0.598	0.563	0.536
Fit_{test}	**0.689**	0.685	0.649	0.631	0.581	0.557	0.519	0.498

Table 9: Model comparisons ($N_{train} = 20000$, $N_{test} = 7000$, 25 epochs, 15 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.893**	0.879	0.866	0.796	0.855	0.779	0.825	0.712
$F1_{test}$	0.727	0.718	**0.765**	0.720	0.761	0.721	0.744	0.665
Fit_{train}	**0.805**	0.795	0.726	0.677	0.659	0.605	0.578	0.498
Fit_{test}	**0.689**	0.683	0.656	0.624	0.593	0.565	0.521	0.466

Table 10: Model comparisons ($N_{train} = 20000$, $N_{test} = 7000$, 20 epochs, 20 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.883**	0.802	0.849	0.767	0.835	0.803	0.813	0.733
$F1_{test}$	0.730	0.691	**0.750**	0.709	0.749	0.732	0.744	0.677
Fit_{train}	**0.798**	0.741	0.714	0.657	0.645	0.622	0.569	0.513
Fit_{test}	**0.691**	0.664	0.645	0.616	0.584	0.572	0.521	0.474

Table 11: Model comparisons ($N_{train} = 20000$, $N_{test} = 7000$, 25 epochs, 20 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.893**	0.862	0.867	0.832	0.852	0.785	0.815	0.749
$F1_{test}$	0.736	0.714	0.759	0.736	**0.760**	0.708	0.741	0.695
Fit_{train}	**0.805**	0.783	0.727	0.702	0.656	0.610	0.571	0.524
Fit_{test}	**0.695**	0.680	0.651	0.635	0.582	0.556	0.519	0.487

Simulation results using the second data partition method Simulation results using the second data partition method are shown in Tables 14-18.

Table 12: Model comparisons ($N_{train} = 15000$, $N_{test} = 6000$, 20 epochs, 25 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.899**	0.886	0.870	0.825	0.837	0.760	0.780	0.748
$F1_{test}$	0.711	0.714	**0.741**	0.725	0.733	0.695	0.693	0.686
Fit_{train}	**0.809**	0.800	0.729	0.698	0.646	0.592	0.546	0.524
Fit_{test}	0.678	**0.680**	0.639	0.628	0.573	0.547	0.485	0.480

Table 13: Model comparisons ($N_{train} = 20000$, $N_{test} = 7000$, 20 epochs, 25 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.883**	0.880	0.854	0.836	0.835	0.772	0.803	0.779
$F1_{test}$	0.727	0.732	**0.762**	0.743	0.745	0.715	0.740	0.714
Fit_{train}	**0.798**	0.796	0.718	0.705	0.645	0.600	0.562	0.545
Fit_{test}	0.689	**0.692**	0.653	0.640	0.582	0.561	0.518	0.500

Table 14: Model comparisons ($K_{train} = 20000$, $K_{test} = 7000$, 20 epochs, 5 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.875**	0.849	0.840	0.819	0.807	0.807	0.772	0.759
$F1_{test}$	0.725	0.703	**0.747**	0.736	0.736	0.736	0.721	0.708
Fit_{train}	**0.793**	0.774	0.708	0.693	0.625	0.625	0.540	0.531
Fit_{test}	**0.688**	0.672	0.643	0.635	0.575	0.575	0.505	0.496

Table 15: Model comparisons ($K_{train} = 20000$, $K_{test} = 7000$, 20 epochs, 10 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.863**	0.855	0.851	0.824	0.817	0.801	0.776	0.730
$F1_{test}$	0.716	0.715	**0.765**	0.734	0.742	0.732	0.720	0.687
Fit_{train}	**0.784**	0.779	0.716	0.697	0.632	0.621	0.543	0.511
Fit_{test}	**0.681**	**0.681**	0.656	0.634	0.579	0.572	0.504	0.481

Table 16: Model comparisons ($K_{train} = 20000$, $K_{test} = 7000$, 20 epochs, 15 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.878**	0.867	0.852	0.850	0.830	0.748	0.795	0.763
$F1_{test}$	0.732	0.723	**0.761**	0.756	0.756	0.698	0.734	0.711
Fit_{train}	**0.795**	0.787	0.716	0.715	0.641	0.583	0.557	0.534
Fit_{test}	**0.692**	0.686	0.653	0.649	0.589	0.549	0.513	0.498

Table 17: Model comparisons ($K_{train} = 20000$, $K_{test} = 7000$, 20 epochs, 20 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.878**	0.850	0.852	0.835	0.832	0.811	0.807	0.785
$F1_{test}$	0.731	0.717	**0.764**	0.748	0.755	0.739	0.743	0.731
Fit_{train}	**0.795**	0.775	0.716	0.705	0.642	0.628	0.565	0.550
Fit_{test}	**0.692**	0.682	0.655	0.644	0.589	0.577	0.520	0.512

Table 18: Model comparisons ($K_{train} = 20000$, $K_{test} = 7000$, 20 epochs, 25 generations)

Model:	CM^4_{GA}	CM^4	CM^6_{GA}	CM^6	CM^8_{GA}	CM^8	M^{10}_{GA}	M^{10}
$F1_{train}$	**0.887**	0.868	0.857	0.804	0.830	0.803	0.804	0.769
$F1_{test}$	0.730	0.720	**0.768**	0.734	0.757	0.727	0.743	0.715
Fit_{train}	**0.801**	0.788	0.720	0.683	0.641	0.622	0.563	0.538
Fit_{test}	**0.691**	0.684	0.658	0.634	0.590	0.569	0.520	0.501

Performance analysis

Comparison between GA-based MCNN models and non-GA-based MCNN models Simulation results as shown in Tables 4-13 for the first data partition and Tables 14-18 for

the second data partition indicated that four MCNN models (i.e., CM^4_{GA}, CM^6_{GA}, CM^8_{GA} and M^{10}_{GA}) generated by Algorithm 2 with the new GA outperformed four MCNN models (i.e., CM^4, CM^6, CM^8 and M^{10}) without the new GA in terms of testing F1-scores for 58 cases among 60 cases, respectively; CM^8_{GA} tied with CM^8 as shown in Tables 7 and 14. Thus, Algorithm 2 with the new GA is useful.

Comparison between CMCNN models and non-compressed MCNN models Simulation results as shown in Tables 4-18 indicated that three CMCNN models with the new GA had lower testing F1-scores and lower memory usage than a non-compressed MCNN model (M^{10}_{GA}) with the new GA for 43 cases among 45 cases; CM^4_{GA} had higher testing F1-scores than M^{10}_{GA} as shown in Tables 17 and 18. In addition, three CMCNN models without the new GA had lower testing F1-scores and lower memory usage than a non-compressed MCNN model (M^{10}) without the new GA for 42 cases among 45 cases; CM^4 had higher testing F1-scores than M^{10} as shown in Tables 14 and 17, and CM^8 had higher testing F1-scores than M^{10} as shown in Table 16. Thus, CMCNN models can outperform a non-compressed MCNN model.

Overall comparisons The best CNN models come from the two smallest CMCNNs in terms of the number of convolutional layers. Also, all bolded testing F1-scores are for CMCNN models using the new GA, showing that the new GA is useful. Although the best CMCNN selected by Algorithm 2 does not have the best testing F1-scores, it has the shortest execution time (smallest power usage) and the smallest model size (454KB, smallest memory usage).

Conclusions

Simulation results show that CMCNNs can achieve better performance, shorter training time (i.e., less power consumption), and less memory usage (model size) than MCNNs. For example, CM^4_{GA} with 454KB model size is more accurate, memory-efficient, power-efficient, and faster than M^{10} with 815KB model size. Results show that the new GA-based model selection algorithm can perform better than a random model selection algorithm. CM^k_{GA} outperformed CM^k for $k = 4, 6, 8$ in most simulation cases, so the new GA is effective in automatically finding the best CMCNNs. However, the new GA led to overfitting. Effective, fast, power-efficient, and memory-efficient CMCNNs using a small number of convolutional layers with different activation functions can be used in various applications, especially in computer vision. With faster training, computer systems and mobile devices running CMCNNs can save power, which would increase power efficiency and battery life. CMCNNs are more memory-efficient than MCNNs; the model sizes of CMCNNs are much smaller than those of MCNNs.

Future works

The fitness function for GA does not have to be linear, and it can be changed and optimized to meet different users' requirements, especially the weights. Other compressed deep neural networks, such as ResNets and DenseNets with a reduced number of convolutional layers using different activation functions, will be developed to increase classification accuracy, reduce training times, reduce computational power, and reduce memory usage (model size), especially for systems with limited power and memory.

Other NAS methods that are not evolution-based search algorithms will be tested. Other general optimization techniques, such as particle swarm optimization (Sinha et al. 2018; Tan et al. 2019) and microcanonical annealing (Ayumi et al. 2018), will be used to develop more effective CMCNN model selection algorithms. Since the CMCNN model selection among a large number of potential CMCNNs takes a very long time for many generations, parallel optimization techniques will be developed.

A new compact neural network using novel differential equation unit activation functions could achieve comparable performance compared with bigger neural networks (Torkamani et al. 2019). CMCNNs using the new differential equation unit activation functions and other commonly used activation functions will be used for new CMCNN model selection simulations. A transfer learning method will be used to speed up training CMCNNs by reusing previously optimized parameters for AIoT applications.

Acknowledgments

The author would like to thank the reviewers very much for their valuable comments that help improve the quality of this paper.

References

Ayumi, V.; Rere, L. M. R.; Fanany, M. I.; and Arymurthy, A. M. 2016. Optimization of convolutional neural network using microcanonical annealing algorithm, In *Proceedings of 2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS 2016),* 506–511.

Cheng, A.-C.; Lin, C.H.; Juan, D.-C.; Wei, W.; and Sun, M. 2019. Toward Instance-aware Neural Architecture Search, In *Proceedings of the 6th ICML Workshop on Automated Machine Learning,* $https : //www.automl.org/wp - content/uploads/2019/06/automlws2019_Paper07.pdf$.

Esteva, A.; Kuprel, B.; Novoa, R.A.; Ko, J.; Swetter, S.M.; Blau, H.M.; and Thrun, S. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542(7639): 115–118.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* 770–778.

Krizhevsky, A. 2009. Learning multiple layers of features from tiny images, https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf.

Krizhevsky, A.; Sutskever, I.; and Hinton, G.E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25,* 1097–1105. Cambridge, MA: MIT Press.

LeCun, Y.; Bengio, Y.; and Hinton, G.E. 2015. Deep learning. *Nature* 521: 436–444.

Li, K. 2008. Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed. *IEEE Transactions on Parallel and Distributed Systems* 19(11): 1484–1497.

Lu, Z.; Whalen, I.; Boddeti, V.; Dhebar, Y.D.; Deb, K.; Goodman, E.D.; and Banzhaf, W. 2018. NSGA-NET: A Multi-Objective Genetic Algorithm for Neural Architecture Search. arXiv: 1810.03522.

Macko, V.; Weill, C.; Mazzawi, H.; and Gonzalvo, J. 2019. Improving Neural Architecture Search Image Classifiers via Ensemble Learning. In *Proceedings of the 6th ICML Workshop on Automated Machine Learning, https : //www.automl.org/wp − content/uploads/*2019/06/$automlws2019_Paper39.pdf$.

Marcus, D.S.; Wang, T.H.; Parker, J.; Csernansky, J.G.; Morris, J.C.; and Buckner, R.L. 2007. Open access series of imaging studies (oasis): cross-sectional MRI data in young, middle aged, nondemented, and demented older adults. *Journal of Cognitive Neuroscience* 19(9), 1498–1507.

Maziarz, K.; Khorlin, A.; Laroussilhe, Q.d.; Jastrzebski, S.; Tan, M.; and Gesmundo, A. 2019. Evolutionary-Neural Hybrid Agents for Architecture Search. In *Proceedings of the 6th ICML Workshop on Automated Machine Learning, https : //www.automl.org/wp − content/uploads/*2019/06/$automlws2019_Paper38.pdf$.

Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; and Kautz, J. 2016. Pruning convolutional neural networks for resource efficient inference. arXiv: 1611.06440.

Niu, W.; Wang, Y.; and Ren, B. 2019. CADNN: Ultra Fast Execution of DNNs on Mobile Devices with Advanced Model Compression and Architecture-Aware Optimization. In *Proceedings of ICML 2019 Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations,* arXiv: 1905.00571.

OASIS Brains Datasets, http://www.oasis-brains.org/#data.

Sinha, T.; Haidar, A.; and Verma, B. 2018. Particle Swarm Optimization Based Approach for Finding Optimal Values of Convolutional Neural Network Parameters. In *Proceedings of 2018 IEEE Congress on Evolutionary Computation,* 1–6.

Stamoulis, D.; Ding, R.; Wang, D.; Lymberopoulos, D.; Priyantha, B.; Liu, J.; and Marculescu, D. 2019. Single-Path NAS: Device-Aware Efficient ConvNet Design. In *Proceedings of ICML 2019 Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations,* arXiv: 1905.04159.

Sun, Y.; Xue, B.; Zhang, M.; and Yen, G. 2018. Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification. arXiv: 1808.03818.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed S.; Anguelov D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2015. Going Deeper with Convolutions. In *Proceedings of 2015 IEEE Conference on Computer Vision and Pattern Recognition,* 1–9.

Szegedy, C.; Ioffe, S.; Vanhoucke, V.; and Alemi, A. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17),* 4278–4284.

Tan, T.Y.; Zhang, L.; Lim, C.P.; Fielding, B.; Yu, Y.; and Anderson, E. 2019. Evolving Ensemble Models for Image Segmentation Using Enhanced Particle Swarm Optimization. *IEEE Access* 7: 34004–34019.

Torkamani, M.; Wallis, P.; Shankar, S.; and Rooshenas, A. 2019. Learning Compact Neural Networks Using Ordinary Differential Equations as Activation Functions. In *Proceedings of ICML 2019 Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations,* arXiv: 1905.07685.

You, Z.; and Pu, Y. 2015. The Genetic Convolutional Neural Network Model Based on Random Sample. *International Journal of u- and e- Service, Science and Technology,* 317–326.

Wiedemann, S.; Kirchhoffer, H.; Matlage, S.; Haase, P.; Gonzalez, A.M.; Marinc, T.; Schwarz, H.; Marpe, D.; Wiegand, T.; Osman, A.; and Samek, W. 2019. DeepCABAC: Context-adaptive binary arithmetic coding for deep neural network compression. In *Proceedings of ICML 2019 Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations,* arXiv: 1905.08318.

Zhang, X.; Cong, H.; Li, Y.; Chen, Y.; Xiong, J.; Hwu, W.-M.; and Chen, D. 2019. A Bi-Directional Co-Design Approach to Enable Deep Learning on IoT Devices. In *Proceedings of ICML 2019 Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations,* arXiv: 1905.08369.

Zhang, X.; Cong, H.; Li, Y.; Chen, Y.; Xiong, J.; Hwu, W.-M.; and Chen, D. 2019. A Bi-Directional Co-Design Approach to Enable Deep Learning on IoT Devices. In *Proceedings of ICML 2019 Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations,* arXiv: 1905.08369.

Zhang L. M. 2018. Multi-function Convolutional Neural Networks for Improving Image Classification Performance. arXiv: 1805.11788 [cs.CV]. May, 2018.

Zur, Y.; Baskin, C.; Zheltonozhskii, E.; Chmiel, B.; Evron, I.; Bronstein, A.M.; and Mendelson, A. 2019. Towards Learning of Filter-Level Heterogeneous Compression of Convolutional Neural Networks. In *Proceedings of the 6th ICML Workshop on Automated Machine Learning, https : //www.automl.org/wp − content/uploads/*2019/06/$automlws2019_Paper01.pdf$.