Journal of Electrical and Electronic Engineering
2020; 8(6): 127-136
http://www.sciencepublishinggroup.com/j/jeee
doi: 10.11648/j.jeee.20200806.11
ISSN: 2329-1613 (Print); ISSN: 2329-1605 (Online)

Shallow SqueezeNext: Real Time Deployment on Bluebox2.0 with 272KB Model Size

Jayan Kant Duggal[*], **Mohamed El-Sharkawy**

Internet of Things Collaboratory, Purdue School of Engineering and Technology, Indiana University Purdue University Indianapolis (IUPUI), Indianapolis, USA

Email address:
jk009d@gmail.com (J. K. Duggal), melshark@iupui.edu (M. El-Sharkawy)

[*]Corresponding author

To cite this article:
Jayan Kant Duggal, Mohamed El-Sharkawy. Shallow SqueezeNext: Real Time Deployment on Bluebox2.0 with 272KB Model Size. *Journal of Electrical and Electronic Engineering*. Vol. 8, No. 6, 2020, pp. 127-136. doi: 10.11648/j.jeee.20200806.11

Received: November 30, 2020; **Accepted**: December 17, 2020; **Published**: December 31, 2020

Abstract: The significant challenges for deploying CNNs/DNNs on ADAS are limited computation and memory resources with very limited efficiency. Design space exploration of CNNs or DNNS, training and testing DNN from scratch, hyper parameter tuning, implementation with different optimizers contributed towards the efficiency and performance improvement of the Shallow SqueezeNext architecture. It is also computationally efficient, inexpensive and requires minimum memory resources. It achieves better model size and speed in comparison to other counterparts such as AlexNet, VGGnet, SqueezeNet, and SqueezeNext, trained and tested from scratch on datasets such as CIFAR-10 and CIFAR-100. It can achieve the least model size of 272KB with a model accuracy of 82%, a model speed of 9 seconds per epoch, and tested on the CIFAR-10 dataset. It achieved the best accuracy of 91.41%, best model size of 0.272 MB, and best model speed of 4 seconds per epoch. Memory resources are of high importance when it comes down to real time system or platforms because usually the memory is quite limited. To verify that the Shallow SqueezeNext can be successfully deployed on a real time platform, bluebox2.0 by NXP was used. Bluebox2.0 deployment of Shallow SqueezeNext architecture achieved a model accuracy of 90.50%, 8.72MB model size and 22 seconds per epoch model speed. There is another version of the Shallow SqueezeNext which performed better that attained a model size of 0.5MB with model accuracy of 87.30% and 11 seconds per epoch model speed trained and tested from scratch on CIFAR-10 dataset.

Keywords: Deep Neural Network (DNN), Design Space Exploration (DSE), Pytorch Implementation, Real-time Deployment, RTMaps, SqueezeNext, Shallow SqueezeNext

1. Introduction

DNN model performance is very critical to ADAS and UAV applications safety. DNNs overcame the limitations of its traditional counterparts which are more memory and computationally expensive algorithms. Here, DNN model performance refers to model accuracy, model memory size, and model speed. Due to more intensive DSE of CNNs in small and macro-CNN architectures, especially for ADAS or real-time embedded systems [10], new architectures were introduced such as SqueezeNet [1] and SqueezeNext [2] baseline architectures, efficient and better than the traditional architectures [8, 10, 15]. DNNs are usually trained and tested on some widely available datasets such as MNIST, CIFAR-10, COCO, ImageNet, etc. DNNs usually comprise of four elemental layers such as activation, convolution, pooling, and fully connected layers. The convolution operation is performed with striding and padding values, the default value of striding is 1 and the padding used is zero-padding to maintain the spatial dimension of the DNN. Different optimizers [7] and learning rate scheduling methods are implemented. To improve DNN architectures, we perform Design Space Exploration (DSE) of DNNs, architecture modification, hyperparameter tuning, and tweaking [4, 5, 11, 15-18]. In this paper, the proposed architecture is implemented on datasets such as CIFAR-100 and CIFAR-10 [9], initially, trained and tested on a GPU and later, was deployed on Bluebox2.0 [11, 16], real time embedded platform by NXP. This research complies Design Space Exploration of DNNs for Shallow SqueezeNext architecture with the help of insights from the following research papers [16-18]. In the end, we deploy the Shallow SqueezeNext with the help of RTMaps [19] on the real time platform, bluebox2.0 by NXP.

2. Literature Review

2.1. SqueezeNet

SqueezeNet architecture [1] comprises of convolutions layers, max pooling layers, fire modules, ReLU and ReLU in place activations, and softmax activation layers. Fire module is the mainstay of SqueezeNet architecture. This module consists of these two following layers, one squeeze layer; s2 (1x1)) and two expand layers; e1 (1x1) and e3 (3x3). They are further, responsible for model size or model parameter reduction and better model speed performance. The three key design strategies implemented to develop this architecture:

1) Replacing the 3x3 convolution layers with 1x1 convolution layers.
2) Reducing number of the input channel to 3x3 convolution layers.
3) Down-sampling or perform max pooling late down in the CNN network.

In comparison, to VGG architecture, it performs better in terms of model size and speed. It reduced the VGG architecture model size from 385MB down to the model size of 0.5MB with an accuracy tradeoff. Additionally, there is a colossal decrease in the parameter count, further, leading to a better model speed for processing per epoch of the SqueezeNet model.

2.2. SqueezeNext

SqueezeNext baseline architecture [2] consists of the following key factors:

1) Better channel reduction by incorporating a two-stage squeeze module subsequently reducing parameters significantly with the help of 3x3 convolutions.
2) It uses separable 3x3 convolutions for model size reduction, and removal of 1x1 convolution after the squeeze module.
3) It incorporates an element-wise addition skip connection identical to ResNet.

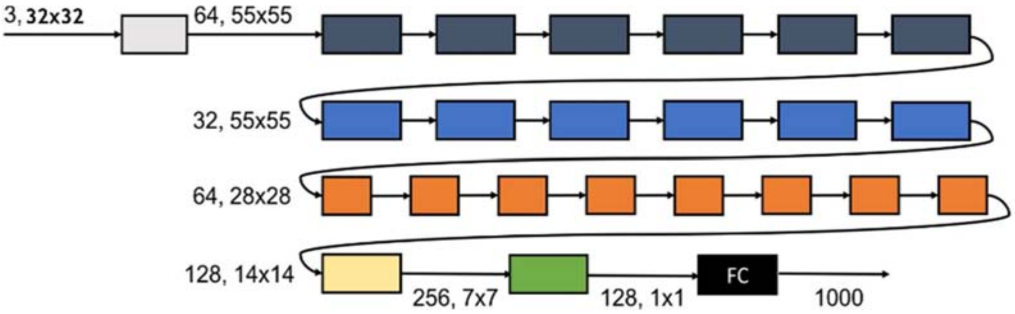


Figure 1. *Four stage [6, 6, 8, 1] baseline SqueezeNext configuration for CIFAR-10.*

Figure 1. illustrates a modified version of the baseline SqueezeNext architecture implemented in Pytorch framework which is trained and tested from scratch on datasets such as CIFAR-100 and CIFAR-10. The baseline SqueezeNext is formed by four stage implementation of bottleneck modules, skip connections, ReLU and ReLU (in-place) layers, batch normalization, spatial resolution layer, max pooling layers, and a fully connected layer. Within baseline SqueezeNext, bottleneck modules are majorly responsible for the rigorous parameters' reduction [14-16]. It comprises of the white block (Figure 1. grey block), the first convolution for the input channel taking in a 3-channel feature map. The consecutive output of the first convolution becomes the input for the subsequent four- stage configuration implementation of the architecture, baseline SqueezeNext. The sequence of different colored blocks (dark blue, blue, orange, and yellow blocks) in Figure 1. succeeding the first convolution (white block) illustrates the four-stage configuration implementation belonging to Shallow SqueezeNext which depicts low level, medium level, and high-level features, respectively. They also depict a change in the resolution of the input feature map in the baseline SqueezeNext architecture. The fact here, is that a smaller number of initial blocks, low-level features holds the redundant information in contrast to mid or high-level features later down the CNN which carries most of the useful feature map information data.

2.3. Modified SqueezeNext

Modified SqueezeNext architecture is developed for the purpose of this research for unbiased comparison between the proposed Shallow SqueezeNext architecture and Modified SqueezeNext implementation based on Pytorch framework. It also assisted in providing great insights for the possible domains of improvement within the baseline architecture and to further, explore the baseline SqueezeNext. Modified SqueezeNext architecture was built out of the basic block illustrated in the Figure 2 (right), which are arranged in a structural form of two block structures, represented in Figure 3. (left) and (right). There is another difference here from the original SqueezeNext architecture, both DNN models developed with the help of the mentioned below basic blocks (Figure 2.) were implemented on Pytorch framework instead of Caffe.

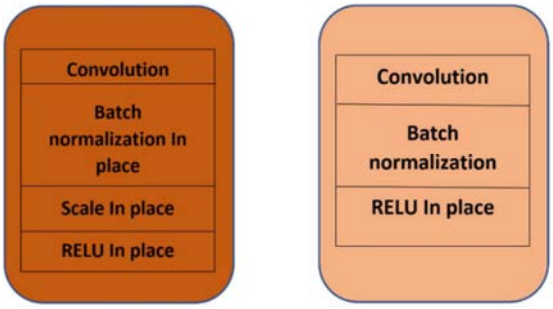


Figure 2. *(Left) Basic block for baseline SqueezeNext, (Right) Modified SqueezeNext basic block.*

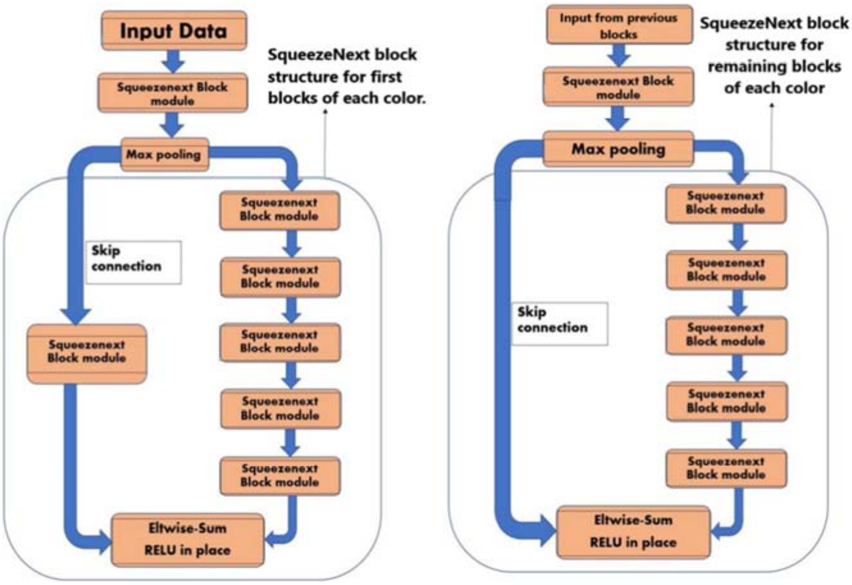


Figure 3. *(Left) Modified SqueezeNext first block structure, (Right) Modified SqueezeNext second block structure.*

Figure 2. (right) illustrates the fundamental basic building block for the Modified SqueezeNext trained and tested on Cifar-10 from scratch implemented in Pytorch framework. It consists of convolution (1x1), regular batch normalization and ReLU (in-place) layers excluding the scaling layer.

In Figure 3, initially, both block structures (left & right) begin with an output of input data being fed into the Figure 2. right basic block (Modified SqueezeNext basic block) which further fed the input to the max pooling layer. The block structure on the right, depicts the first individual initial blocks implementation of the four-stage configuration. It represents, the first dark blue, blue, orange and the last yellow block of the four-stage configuration.

The block structure on the left, forms each of the remaining blocks of the four-stage configuration of the Modified SqueezeNext. For the fair and unbiased comparison with the proposed architecture, all the architectures are trained and tested in Pytorch only with datasets such as CIFAR-100 and CIFAR-10, respectively.

2.4. Architecture Tuning

A recently introduced optimizer and some other activation functions [4] had been used for experiments on the proposed Shallow SqueezeNext Architecture, further, fine tuning and tweaking the proposed architecture.

2.4.1. Adabound

Adabound [12], a newly introduced optimizer which employs bounds on their learning rates dynamically and achieving a transition. It shows good results with the benefits of adaptive methods. The lower and upper bound of it will adjust after running the CNN/DNN for several epochs (in proposed architecture case it was between 60 to 90 epochs) so that it transforms from Adam to SGD. The default hyperparameters for it are learning rate of 0.001, beta1 = 0.9 and beta2 = 0.999. It was seen that the optimizers such as adagrad, adam, and rmsgrad seem to perform better in training, initially. When the learning rates are decayed, SGD begins to outperform. But, in the case of adabound, it converges fast and achieves a bit higher accuracy than SGD.

2.4.2. Rectified Linear Units (ReLU) in Place

RELU-in place is not a linear activation function layer, but it provides similar advantages as of ReLU, additionally with a better performance. It modifies the input directly without allocating any additional output. It is observed to save some amount of memory in comparison to RELU. It cannot be used all the time as it needs a valid operation or valid use case.

2.4.3. Exponential Linear Units (ELU) in Place

ELU is an activation function, converging to zero cost faster and then, producing better and more accurate results. The curve for this activation function will smooth over time, slowly. It also has another special operation case, that is, ELU (in-place). All in-place are observed to save memory, further not allocating any additional outputs which is huge benefactor for a CNN/DNN model.

2.4.4. BlueBox2.0 by NXP

Bluebox2.0 [19] is the second version of real time deployment platform for autonomous driving applications. It provides automotive reliability, functional safety, and freedom to implement the algorithms on frameworks such as Pytorch, TensorFlow and Keras. The recent edition of bluebox2.0 incorporates three essential components are S32V234 (vision processor), LS2084A (embedded compute processor), and S32R27 (radar). It is operated with the help of Linux BSP image on a 16GB microSD card.

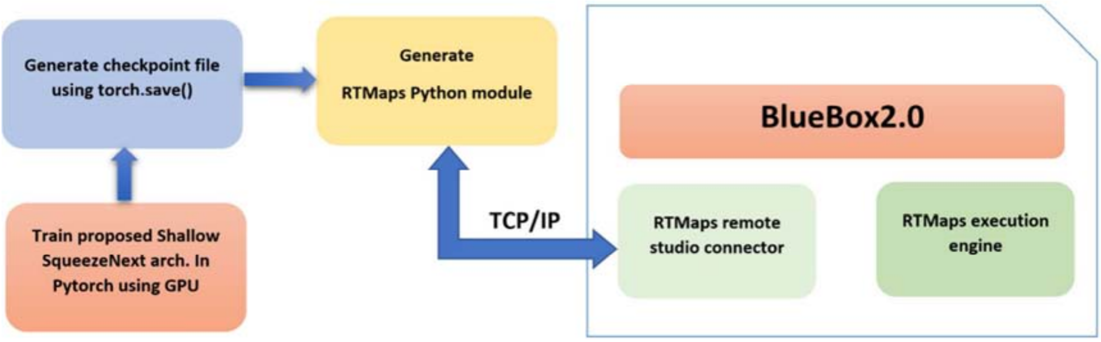

Generate checkpoint file using torch.save()
Generate RTMaps Python module
BlueBox2.0
Train proposed Shallow SqueezeNext arch. In Pytorch using GPU
TCP/IP
RTMaps remote studio connector
RTMaps execution engine

Figure 4. *Shallow SqueezeNext basic block; the fundamental building block for the architecture.*

For deployment of the CNNs/DNNs or the proposed Shallow SqueezeNext architecture it makes use of RTMaps framework [19], another tool used with bluebox2.0 for the architecture deployment.

RTMaps: Real-time Multisensor applications is easy to use, efficient and robust real-time embedded systems. It is designed for developing multimodal based applications, testing, benchmarking, validation, and execution. It consists of four key modules that are RTMaps Runtime Engine, RTMaps Component Library, RTMaps Studio, RTMaps Embedded. The connection between the computer running RTMaps and the remote studio RTMaps on bluebox2.0 can be accessed via a static TCP/IP connection.

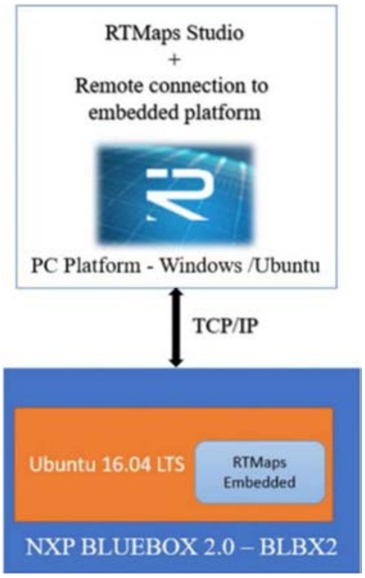


Figure 5. *Shallow SqueezeNext connection between a PC and bluebox2.0 real-time platform with the help of RTMaps via TCP/IP.*

Architecture Deployment: To deploy a Pytorch code with the help of RTMaps for bluebox2.0, it must consist of three key functions to make it work in RTMaps. Three function definitions are birth (), core (), and death () [16, 19]. Pytorch deployment with the help of RTMaps on bluebox2.0 for the Shallow SqueezeNext architecture is shown in Figure 4. The connection between the RTMaps studio with remote connection to embedded platform on a PC and real-time platform with Ubuntu BSP image, bluebox2.0 by NXP can accessed via TCP/IP, illustrated in Figure 5.

3. Shallow SqueezeNext

Shallow SqueezeNext architecture is a shallow (refers to not too deep or small DNN models) and compact DNN architecture. The motivational architectures behind this proposed architecture were SqueezeNext [2], SqueezeNet [1], and MobileNet [3] architectures. During the research, another architecture was developed for better accuracy with model size tradeoff, that is basically a deeper or more comprehensive version of it, High Performance SqueezeNext [17]. Shallow SqueezeNext architecture is made up of bottleneck modules [2] further, consisting of the basic blocks mentioned below in Figure 6. These basic blocks are arranged in a four-stage configuration implementation (Figure 7.) followed by a spatial resolution layer, dropout layer with probability; p equal to 0.3, average pooling, and a fully connected layer.

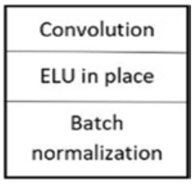


Figure 6. *Fundamental basic block for Shallow SqueezeNext architecture.*

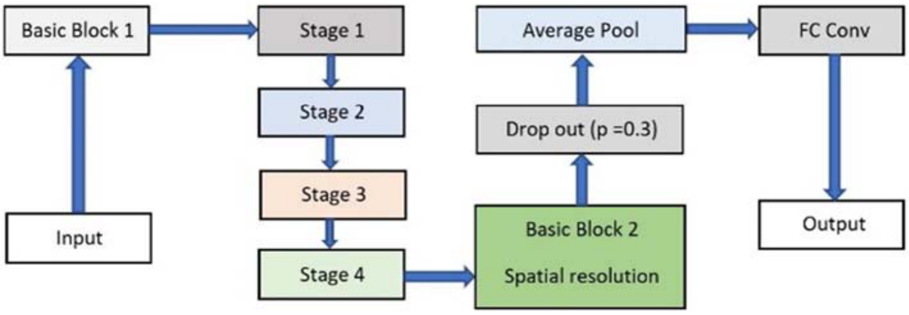


Figure 7. *Architecture illustration for the proposed Shallow SqueezeNext.*

It is based on the following important strategies:

1) Managing depth and width scaling with resolution and width multipliers.
2) Use of only in-place operations in all layers except in the layers where we have a gradient change operation. Carefully, placing it between ELU in-place and batch normalization layer (Figure 6).
3) Incorporating an element-wise addition skip connection to avoid vanishing gradient problem.
4) Addition of a drop out layer at the end of four stage configuration after the average pooling layer.
5) Reduction of max-pooling layers and replacing them with average pooling layers. As observed in Figure 7, average pooling layer after drop-out layer.

The architecture implements the strategy of training and testing different optimizers. Figure 6. represents the basic block which is the fundamental building for the architecture with following layers convolution (1x1), ELU (in-place) [13], and batch normalization. Shallow SqueezeNext basic blocks together form bottleneck modules, illustrated in Figure 8. (left), these bottleneck modules are arranged in a four-stage configuration as shown in Figure 8 (right).

The basic blocks in Figure 6. and bottleneck modules four-stage configuration (Figure 8.) together combines to build and form the proposed DNN architecture, Shallow SqueezeNext architecture (Figure 7). Figure 8. (left) illustrates bottleneck module made from basic block (refer Figure 6.) combined with different types of convolution layers specifically, such as, 1x1, 3x1, and 1x3 convolutions, respectively.

Figure 8. (right) illustrates the detailed proposed Shallow SqueezeNext with (1 2 8 1) four-stage configuration depicting bottleneck modules representing one grey colored bottleneck module in first stage, two blue colored bottleneck modules in second stage, four orange colored bottleneck modules in third stage and finally, one green bottleneck module in the fourth stage.

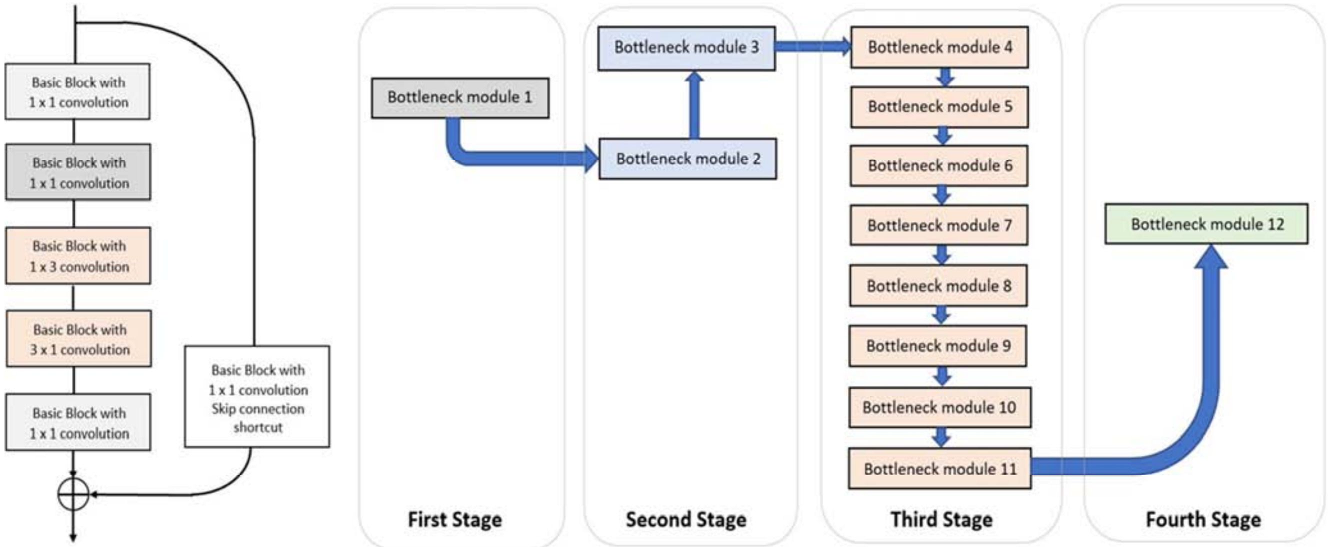


Figure 8. *(Left) Bottleneck module depicting different convolutions and skip connection, (Right) Four stage [1, 2, 8, 1] configuration implementation of Shallow SqueezeNext depicting the left bottleneck modules.*

3.1. Resolution Multiplier

Resolution multiplier [3] is the first hyper-parameter used to reduce the computational resource usage belonging to a CNN/DNN. It is another important parameter which have a significant effect on the parameter reduction and apparently, effect the scaling size of the model. This is responsible for reduced size and parameter for the Shallow SqueezeNext architecture.

3.2. Width Multiplier

Width multiplier [3] is the second hyper-parameter used to develop small, compact, and less expensive DNN models in terms of computation and memory resource usage. It develops a uniformly thin deep neural network at each layer, further, helping to reduce the computational expenses and number of parameters by a power of two of the width multiplier term.

4. Results

4.1. Shallow SqueezeNext Results

Shallow SqueezeNext architecture was implemented with the approaches mentioned in the literature review section, leading to various number of models of the proposed architecture. The model size ranges from 4.2MB to a small size of 115KB or 0.115MB as shown in Table 3 with mostly model accuracy above 80% and model speed of approximately under 15 seconds per epoch for the experimental models. In the following tables, only few of the several better model's results out of total 600 models or experiments are being discussed below. The nomenclature for the proposed Shallow SqueezeNext models and results from the tables within this section represents Shallow SqueezeNext architecture version name followed by resolution multiplier, and width multiplier. We can infer from Table 1. that a better reduced Shallow SqueezeNext model size is achieved that is 272KB or 0.272 MB, (Shallow SqueezeNext-06-0.4x model) from the 9.525MB, baseline SqueezeNext model size. Shallow SqueezeNext-06-0.4x model is 35x smaller than SqueezeNext-23-2x, 10x smaller than SqueezeNext-23-1x and approximately, 11x smaller than SqueezeNet v1.0 and SqueezeNet v1.1.

Table 1. *Comparison with baseline SqueezeNet architecture and baseline SqueezeNext architecture.*

Model	Accuracy%	Model Size (MB)	Model speed (sec)
Baseline SqueezeNet-v1.0	79.59	3.013	04
SqueezeNet-v1.1	77.55	2.961	04
Baseline SqueezeNext-23-1x	87.15	2.586	19
SqueezeNext-23-2x	90.48	9.525	22
Shallow SqueezeNext-14-1.5x	91.41	8.720	22
Shallow SqueezeNext-21-0.2x	90.29	1.814	27
Shallow SqueezeNext-12-1.0x	88.46	0.504	19
Shallow SqueezeNext-06-0.4x	81.97	0.272	04
Shallow SqueezeNext-06-0.2x	81.86	0.273	04

+All results are 3 average runs implemented along with SGD optimizer with Nesterov plus momentum and LR equal to 0.01

Implementation of in-place activation functions, elimination of the extra max-pooling layers and with the introduction of the suitable resolution and width multipliers made the proposed architecture more compact, efficient, and flexible. With the change of resolution and width multiplier, the proposed Shallow SqueezeNext architecture can be deployed with better accuracy but with a trade-off of memory size and memory speed. Shallow SqueezeNext hyperparameters for each variation of model was saved with a Pytorch function, save (). The checkpoint is then, loaded with the help of Pytoch function, load () which is subsequently utilized for the training the architecture. This step of saving and loading the checkpoint is critical for the success of the Shallow SqueezeNext because not all hyper-parameters are saved and loaded but just the important ones. The generated model checkpoint file size is used to determine the model size and final average accuracy. This checkpoint file is again utilized for the testing Shallow SqueezeNext architecture deployment on Bluebox2.0 by NXP [11, 16].

Table 2. *Different resolution multipliers implemented on Shallow SqueezeNext [16, 18].*

Model	Acc.%	Mod. Size (MB)	Mod. Speed (seconds)	Resol (R)
Shallow SqueezeNext-06-0.2x	82.47	0.296	13	1111
Shallow SqueezeNext-06-2x	89.35	4.210	21	1111
Shallow SqueezeNext-08-2x	77.48	2.961	04	1221
Shallow SqueezeNext-10-1x	87.63	2.560	23	1331
Shallow SqueezeNext-12-2x	87.96	2.563	19	1441
Shallow SqueezeNext-14-1x	82.44	0.370	07	1551
Shallow SqueezeNext-14-1.5x	91.41	8.720	22	1281
Shallow SqueezeNext-16-1x	82.86	1.240	08	1661
Shallow SqueezeNext-21-0.2x	90.29	1.814	27	2 2 14 1

*Acc. – Accuracy; Mod. Size – Model Size; Mod. Speed – Model Speed; Resol- Resolution multiplier for each of four-stage configuration.

Table 3. *Different width multipliers implemented with Shallow SqueezeNext [16, 18].*

Model	Acc.%	Size (MB)	Speed (sec)	Width (x) W
Shallow SqueezeNext-06-0.125x	66.4	0.115	07	0.125
Shallow SqueezeNext-06-0.15x	72.2	0.141	08	0.150
Shallow SqueezeNext-06-0.2x	82.5	0.296	13	0.200
Shallow SqueezeNext-06-0.3x	77.9	0.196	08	0.300
Shallow SqueezeNext-12-0.4x	87.3	0.485	13	0.400
Shallow SqueezeNext-14-0.5x	89.0	0.772	17	0.500
Shallow SqueezeNext-06-0.6x	84.6	0.480	10	0.600
Shallow SqueezeNext-07-0.7x	88.1	0.704	12	0.700
Shallow SqueezeNext-06-0.8x	87.7	0.774	12	0.800
Shallow SqueezeNext-06-0.9x	86.3	0.950	12	0.900
Shallow SqueezeNext-12-1.0x	88.5	0.504	19	1.000
Shallow SqueezeNext-06-1.5x	82.5	2.442	17	1.500
Shallow SqueezeNext-06-2.0x	89.4	4.201	21	2.000

+Acc. – Accuracy; Size – Model Size; Speed – Model Speed; Width- Width times multiplication for each of the four-stage configuration.

Table 4. *Different dropout layer probabilities with Shallow SqueezeNext [16, 18].*

Model	Acc.%	Mod. Size (MB)	Mod. Speed (sec)	Dropout (p)
Shallow SqueezeNext-06-0.2x-v1	80.82	0.273	04	0.1
Shallow SqueezeNext-06-0.2x-v1	81.44	0.273	04	0.2
Shallow SqueezeNext-06-0.2x-v1	81.87	0.273	04	0.3
Shallow SqueezeNext-06-0.2x-v1	81.86	0.273	04	0.4
Shallow SqueezeNext-06-0.2x-v1	81.70	0.273	04	0.5

+Acc. – Accuracy; Mod. Size – Model Size; Mod. Speed – Model Speed; Drop out (p)- Resolution multiplier for each of four-stage configuration.

The benefit of this proposed architecture is that it can be readily implemented on real-time systems, BlueBox2.0 by NXP [16, 19] with limited memory with the help of dropout layer [6]. Table 4. illustrates the results attained with the different values of dropout layer probabilities for Shallow SqueezeNext justifying dropout with probability value, p = 0.3 or 0.4 is a better default value for the proposed architecture. Table 5. represents the additional results for the Shallow SqueezeNext [9].

Table 5. *Additional results for Shallow SqueezeNext with CIFAR-10 [16, 18].*

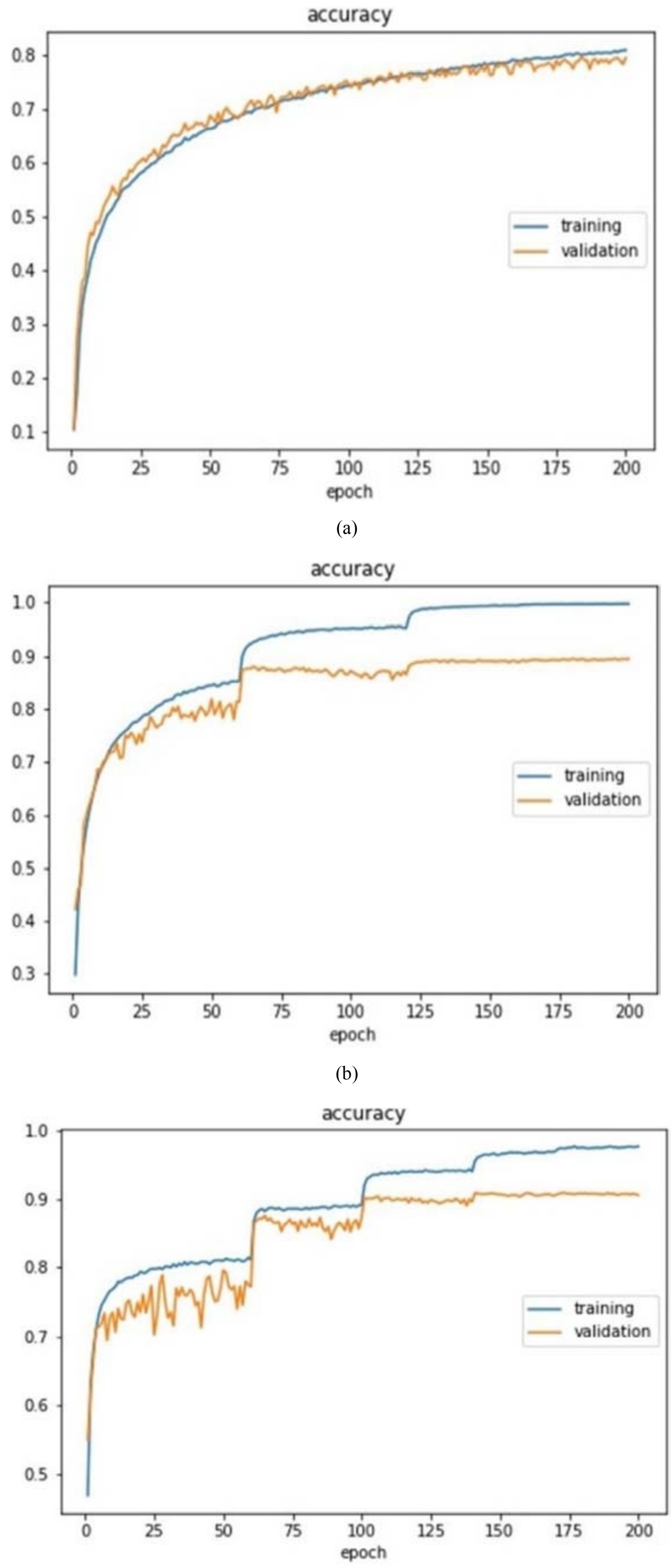
Model	Acc.%	Mod. Size (MB)	Mod. Speed (sec)	R, W
Shallow SqueezeNext-14-1.5x-v1	91.41	8.720	22	1281, 1.5x
Shallow SqueezeNext-21-0.2x-v1	90.27	1.8.14	27	2 2 14 1, 0.2
Shallow SqueezeNext-06-0.575x-v1	81.80	0.449	06	1111,0.575
Shallow SqueezeNext-06-0.4x-v1	81.97	0.272	09	1111,0.4
Shallow SqueezeNext-09-0.5x-v1	87.73	0.531	11	1141,0.5

+Acc. – Accuracy; Mod. Size – Model Size; Mod. Speed – Model Speed; R, W- Resolution multiplier, Width multiplier for each of four-stage configuration**.**

Table 6. *Shallow SqueezeNext Results trained and tested with CIFAR-100 from scratch [16, 18].*

Model	Acc.%	Size (MB)	Speed (sec)	Optimizer (x)
Shallow SqueezeNext-14-1.0x	66.12	6.90	20	Adabound
Shallow SqueezeNext-09-0.5x	58.27	1.40	11	Adam
Shallow SqueezeNext-09-0.5x	33.73	1.40	13	Adamax
Shallow SqueezeNext-09-0.5x	77.9	1.00	09	Adagrad
Shallow SqueezeNext-09-0.5x	87.3	1.40	12	Adabound
Shallow SqueezeNext-09-0.5x	89.0	1.40	12	Adadelta
Shallow SqueezeNext-09-0.5x	84.6	1.00	09	ASGD
Shallow SqueezeNext-09-0.5x	88.1	1.40	11	RMSprop
Shallow SqueezeNext-09-0.5x	87.7	1.40	25	Rprop
Shallow SqueezeNext-09-0.5x	89.4	1.10	08	SGD with momentum and nestrov

+Acc. – Accuracy; Size – Model Size; Speed – Model Speed; Optimizer- Optimizer implemented with the proposed architecture.



(a)

(b)

(c)

Figure 9. *(a) SqueezeNet accuracy baseline architecture, (b) SqueezeNext accuracy baseline architecture, (c) Shallow SqueezeNext accuracy proposed.*

In Table 6, all results have a unique behavior illustrating the effect of different optimizers [7] and ELU [13] implementation on the proposed architecture.

Also, from the above-mentioned tables the inference can be that deep residual layer [4, 8, 13, 15] effects the tradeoff between model accuracy, model speed, and size of the proposed Shallow SqueezeNext architecture. Figure 9. (a-c) illustrates the baseline SqueezeNet, baseline SqueezeNext and the proposed architecture Shallow SqueezeNext accuracies trained on the CIFAR-10 dataset. The graph comparison between the Figures 9 (a), (b) & (c) illustrates, less overfitting in Figure 9 (c) depicted by the empty space or gap between training and validation curve in comparison to (a) & (c). These curves approach to 1.0 quickly. This validates the proposed architecture performs better in terms of model parameters (model accuracy, model speed and model size) than the SqueezeNext and SqueezeNet baseline model which is trained and tested from scratch on CIFAR-10 and CIFAR-100 datasets.

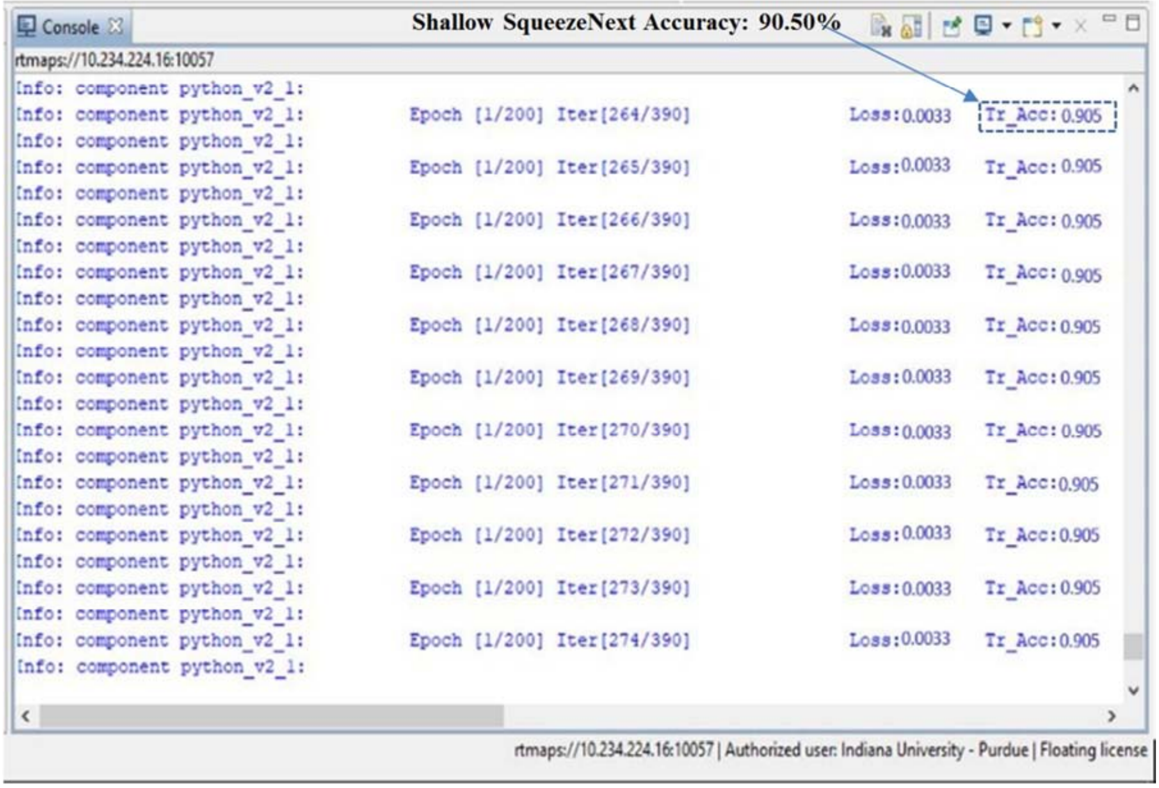
4.2. Bluebox2.0 Implementation Results

The Shallow SqueezeNext architecture is finally deployed on bluebox2.0 by NXP to verify and validate the efficiency and integrity of the Shallow SqueezeNext architecture [16]. The Pytorch generated checkpoint files were trained on datasets such as CIFAR-100 and CIFAR-10 with the help of RTX 2080ti GPU and then, deployed and tested on bluebox2.0 by NXP. The deployment of the Shallow SqueezeNext is shown in Figure 4. The result comparison of the Shallow SqueezeNext is shown below in Table 7.

Figure 10. illustrates the Shallow SqueezeNext deployment results attained by training the architecture on RTX 2080ti GPU with CIFAR-10 dataset from scratch and test the architecture by deploying it with the help of RTMaps on a real-time development platform, bluebox2.0 by NXP.

Table 7. *Bluebox2.0 deployment results for Shallow SqueezeNext [16, 18].*

Model	Acc.%	Mod. Size (MB)	Mod. Speed (sec)
Squeezed CNN (SqueezeNet implementation)	79.30	12.9	11
Shallow SqueezeNext-14-1.5x	**90.50**	8.72	22
Shallow SqueezeNext-06-0.575x	81.50	0.449	**06**
Shallow SqueezeNext-09-0.5	87.30	**0.531**	11



```
rtmaps://10.234.224.16:10057
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[264/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[265/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[266/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[267/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[268/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[269/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[270/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[271/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[272/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[273/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
Info: component python_v2_1:      Epoch [1/200] Iter[274/390]      Loss:0.0033   Tr_Acc:0.905
Info: component python_v2_1:
rtmaps://10.234.224.16:10057 | Authorized user: Indiana University - Purdue | Floating license
```

Figure 10. *Shallow SqueezeNext deployment on real time platform Bluebox2.0.*

5. Conclusion

In this paper, based on the insights from the existing CNNs/DNNs and methods such as fine hyperparameter tuning (refers to implementation of different optimizer with step size decay learning rate scheduling, using momentum and nestrov with SGD optimizer, tuning the parameters for normalization and data preprocessing), training the proposed architecture from scratch with no transfer learning, using comparatively small datasets, and architecture modifications the proposed Shallow SqueezeNext architecture is introduced. It further, explores the DSE of CNNs, optimizing it with the help of all the different activation functions, in-place functions, introduction of ELUs and optimizers. Shallow SqueezeNext has 120x fewer parameters than AlexNet architecture and achieved a reduced, 0.5MB model size. It has further, 600x smaller model size than AlexNet architecture without compression. The results show the tradeoff between the proposed model speed, size, and accuracy. About different optimizers, SGD and Adabound optimizers outperformed their counterparts. With a minimum model size of 0.272MB and model accuracy 81.97% it is expected to be deployed efficiently on ADAS applications. The deployment results for Shallow SqueezeNext trained and tested on Cifar-10 attained a model accuracy of 90.50%, 8.72 MB model size and 22 seconds per epoch model speed. Additionally, another model variation of Shallow SqueezeNext achieved a reduced model size of 0.531 MB with 87.30% model accuracy. In the research, the focus is laid on DSE of DNNs, hyper-parameter optimizer, training and testing the Shallow SqueezeNext architecture from scratch without any transfer learning in contrast to the conventional approach.

References

[1] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J. and Keutzer, K., 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. arXiv preprint arXiv: 1602.07360.

[2] Gholami, A., Kwon, K., Wu, B., Tai, Z., Yue, X., Jin, P., Zhao, S. and Keutzer, K., 2018. Squeezenext: Hardware-aware neural network design. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (pp. 1638-1647).

[3] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H., 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv: 1704.04861.

[4] Ludermir, T. B., Yamazaki, A. and Zanchettin, C., 2006. An optimization methodology for neural network weights and architectures. IEEE Transactions on Neural Networks, 17 (6), pp. 1452-1459.

[5] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B. and Shelhamer, E., 2014. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv: 1410.0759.

[6] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15 (1), pp. 1929-1958.

[7] Ruder, S. An overview of gradient descent optimization algorithms. arXiv preprint arXiv: 1609.04747, 2016.

[8] 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 the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[9] Krizhevsky, A., Nair, V. and Hinton, G., 2010. Cifar-10 (canadian institute for advanced research). URL http://www. cs. toronto. edu/kriz/cifar. html, 5.

[10] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).

[11] Jayan Kant Duggal and Mohamed El-Sharkawy, “Shallow SqueezeNext Architecture Implementation on Bluebox2.0,” Transactions on Computational Science and Computational Intelligence, Springer, 2020.

[12] Luo, L., Xiong, Y., Liu, Y. and Sun, X., 2019. Adaptive gradient methods with dynamic bound of learning rate. arXiv preprint arXiv: 1902.09843.

[13] Clevert, D. A., Unterthiner, T. and Hochreiter, S., 2015. Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv: 1511.07289.

[14] Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv: 1409.1556.

[15] Shah, A., Kadam, E., Shah, H., Shinde, S. and Shingade, S., 2016, September. Deep residual networks with exponential linear unit. In Proceedings of the Third International Symposium on Computer Vision and the Internet (pp. 59-65).

[16] Duggal, Jayan Kant. Design Space Exploration of DNNs for Autonomous Systems. Diss. Purdue University Graduate School, 2019.

[17] Duggal, Jayan Kant, and Mohamed El-Sharkawy. "High Performance SqueezeNext for CIFAR-10." 2019 IEEE National Aerospace and Electronics Conference (NAECON). IEEE, 2019.

[18] Duggal, Jayan Kant, and Mohamed El-Sharkawy. "Shallow SqueezeNext: An Efficient \& Shallow DNN." 2019 IEEE International Conference of Vehicular Electronics and Safety (ICVES). IEEE, 2019.

[19] Venkitachalam, S., Manghat, S. K., Gaikwad, A. S., Ravi, N., Bhamidi, S. B. S. and El-Sharkawy, M., 2018. Realtime Applications with RTMaps and Bluebox 2.0. In Proceedings on the International Conference on Artificial Intelligence (ICAI) (pp. 137-140). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).