Shallow SqueezeNext: An Efficient & Shallow DNN

Jayan Kant Duggal and Mohamed El-Sharkawy
Electrical and Computer Engineering
IoT Collaboratory, Purdue School of Engineering and Technology, IUPUI
jaydugga@iu.edu, melshark@iupui.edu

***Abstract*—CNN has gained great success in many applications but the major design hurdles for deploying CNN on driver assistance systems or ADAS are limited computation, memory resource, and power budget. Recently, there has been greater exploration into small DNN architectures, such as SqueezeNet and SqueezeNext architectures. In this paper, the proposed Shallow SqueezeNext architecture for driver assistance systems achieves better model size with a good model accuracy and speed in comparison to baseline SqueezeNet and SqueezeNext architectures. The proposed architecture is compact, efficient and flexible in terms of model size and accuracy with minimum tradeoffs and less penalty. The proposed Shallow SqueezeNext uses SqueezeNext architecture as its motivation and foundation. The proposed architecture is developed with intention for implementation or deployment on a real-time autonomous system platform and to keep the model size less than 5 MB. Due to its extremely small model size, 0.370 MB with a competitive model accuracy of 82.44 %, decent both training and testing model speed of 7 seconds, it can be successfully deployed on ADAS, driver assistance systems or a real time autonomous system platform such as BlueBox2.0 by NXP. The proposed Shallow SqueezeNext architecture is trained and tested from scratch on CIFAR-10 dataset for developing a dataset specific trained model.**

***Index Terms*—Autonomous Driver Assistance Systems (ADAS), Shallow SqueezeNext architecture, Convolution Neural Networks (CNN), Deep Neural Networks (DNN), SqueezeNext, SqueezeNet, Design space exploration (DSE), CIFAR-10, Pytorch.**

I. Introduction

In many real world applications such as ADAS, robotics, self-driving cars, and augmented reality, the recognition tasks are needed to be carried out in a timely fashion on a computationally limited platform. The general trend has been to make the DNN architectures deeper in order to achieve higher accuracy [13, 15, 16, 29]. However, these advances to improve accuracy are not necessarily making networks more efficient with respect to model size and speed. They have been used successfully in object recognition, object detection [17, 22], object segmentation [16], face recognition, pose estimation, style transferring, natural language processing, and many more applications. DNN-based methods demand much more computations and memory resources compared with traditional methods. Recently, CNN achieved an astonishing benchmark accuracy of 99% with GPipe: efficient training of giant neural networks using pipeline parallelism. DNNs have been shown in recent years to outperform other machine learning methods in a wide range of applications such as ADAS, intelligent cameras surveillance and monitoring, autonomous car, drones, and robots. Many papers on small networks focus only on model size but do not consider speed. As such, the majority of research in DNN has largely focused on designing deeper and more complex deep neural network architectures for improved accuracy. The increased demand for machine learning applications in embedded devices caused an increase in the amount of research exploration on the design of smaller, more efficient DNN architectures. They can both infer and train faster, as well as transfer faster onto embedded devices. The approach to design smaller and shallow DNN architectures is to take a principled approach and employ architectural design strategies to achieve more efficient DNN macro architectures. An exemplary case of what can be achieved using such an approach is SqueezeNet [1] and also, smaller DNN architectures as SquishedNets and SqueezeNext. This paper proposes an efficient network architecture in order to a build very small, efficient DNN model that is the proposed Shallow SqueezeNext architecture. Specifically, this paper makes the following key contributions, small model size and better model accuracy results are shown for the proposed Shallow SqueezeNext architecture. The rest of the paper is organized as follows. In section 2, a background on SqueezeNet and SqueezeNext architectures are reviewed. Consequently, section 3 describes the proposed Shallow SqueezeNext architecture. Section 4, explains the hardware and software used for training and testing the proposed architecture from scratch on CIFAR-10 dataset. After that, in section 5, focus is laid on the results, tables and s shown. Finally, section 6 mentions the conclusions and discussions of the paper.

II. Background

A. SqueezeNet

SqueezeNet [1] is the state-of-the-art CNN model which only uses 3x3 and 1x1 convolutional kernels. Using 1x1 filters reduces depth, hence, it reduces the computation of the 3x3 filters. It achieves the same accuracy as AlexNet does for ImageNet with 50x fewer parameters which make it suitable for the embedded systems. The distinct feature of SqueezeNet is a lack of fully connected layers. SqueezeNet uses an average pooling layer to calculate classification scores using small convolution kernels instead of using a fully connected layer which have immensely reduced computation and memory demand. This feature makes SqueezeNet best suited for the embedded platform with three key design strategies employed: (1) decrease the number of

This is the author's manuscript of the article published in final edited form as:

Duggal, J. K., & El-Sharkawy, M. (2019). Shallow SqueezeNext: An Efficient Shallow DNN. 2019 IEEE International Conference on Vehicular Electronics and Safety (ICVES), 1–6. https://doi.org/10.1109/ICVES.2019.8906416

3x3 filters, (2) decrease the number of input channels to 3x3 filters, and (3) downsample late in the network. This macro architecture is composed of fire modules that possess an incredibly small model size. Then, SqueezeNet v1.1 is introduced, where the number of filters as well as the filter sizes are further reduced, resulting in 2.4x less computation than the original SqueezeNet without sacrificing model accuracy. Inspired by the incredibly small macro architecture of SqueezeNet, insights are gained from this and some modifications are made in the proposed architecture. The SqueezeNet architecture is shown on the left side of Fig. 1.

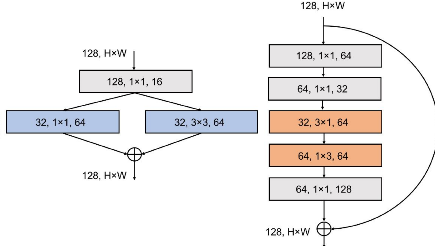

Fig. 1: Illustration of SqueezeNet's fire module (left) and SqueezeNext's bottleneck module (right).

B. SqueezeNext Architecture

SqueezeNext[2] uses SqueezeNet architecture as a baseline architecture. It consists of the following key strategies (1) A more aggressive channel reduction by incorporating a two-stage squeeze module, significantly reducing the total number of parameters used with the 3x3 convolutions. (2) Separable 3x3 convolutions to further reduce the model size, and remove the additional 1x1 branch after the squeeze module. (3) An element-wise addition skip connection similar to that of ResNet architecture. SqueezeNext baseline architecture comprises of bottleneck modules with four stage implementation, batch normalization layers, Relu and Relu(in-place) nonlinear activations, max, and average pool layers, Xavier uniform initialization, spatial resolution layer, and lastly, a fully connected layer is used with [1,2,8,1] four stage block configuration. The bottleneck module, shown in Fig. 1 (right hand side), is the backbone of the SqueezeNext architecture as it significantly reduces the number of parameters without reducing the model accuracy. The SqueezeNext baseline architecture achieves better model accuracy and size in comparison to SqueezeNet baseline architecture because of the use of bottleneck modules and the width multiplier.

The architecture comprises of the following layers in a sequence: first convolution layer, second max pooling layer, shown in Fig. 2, then a four-stage configuration with a kernel size 3, spatial resolution layer and average pooling layer, finally, followed by a fully connected layer. The SqueezeNext baseline architecture is trained from scratch on the CIFAR-10 dataset with input size 32x32 and 3 input channels with 10 target classes for the fair comparison with

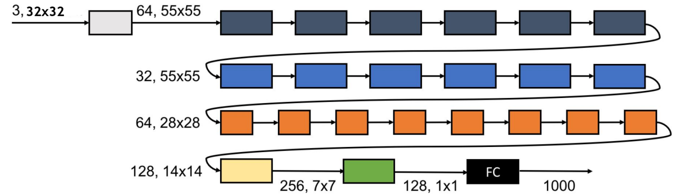

Fig. 2: SqueezeNext baseline architecture for CIFAR-10.

the proposed architecture with Pytorch Implementation of it. For implementation purpose, we used batch normalization layer and removed scale in place layers in the basic block for SqueezeNext baseline without use of transfer learning. It was trained and tested from scratch on CIFAR-10 dataset. As the original squeezenext baseline was trained and tested for ImageNet only and CNN model was a Caffe based model.

III. Shallow SqueezeNext Architecture

The proposed Shallow SqueezeNext architecture is a CNN architecture. It is inspired from SqueezeNext [2], SqueezeNet [1] and Mobilenet [3] architectures. It is based on the SqueezeNext architecture and a shallower architecture. It comprises of bottleneck modules [2] which are further made up of basic blocks arranged in a four stage configuration followed by a spatial resolution layer, average pooling layer and a fully connected layer. The architecture implements SGD optimizer with momentum, decay and nestrov terms are used for the optimizer. It also makes use of a step decay with exponential based learning rate schedule with four LR update that first LR change after 60 epochs, second after 120 epochs, third after 150 epochs and fourth after 180 epochs. Further, the bottleneck module, shown in Fig. 5, comprises of a 1x1 convolution, second 1x1 convolution, 3x1 convolution, 1x3 convolution and then a 1x1 convolution. These convolutions are basic block (Fig 4) consists of a convolution layer, ELU in place, and batch normalization layer. These basic blocks form convolutions within bottleneck modules which further, are put together and arranged in the four stage implementation configuration along with a spatial layer, dropout layer, average pooling and a FC layer are shown in Fig. 4. The spatial layer (green block) can be removed in the proposed shallower versions or proposed architecture's small sized models to reduce the parameter count with the CNN. The trained checkpoint file is saved using the model stat dictionary method of Pytorch avoiding the optimizer state dictionary or other parameters to again reduce the model size and improve the model speed.

It is concluded with the descriptions of the two model shrinking hyper parameters such as the width multiplier and resolution multiplier in the following subsections. The right side of Fig. 3 illustrates the proposed architecture with [1,2,8,1] four stage configuration. Fig. 4 illustrates the Shallow SqueezeNext bottleneck module comprising of Shallow SqueezeNext basic blocks. Table II presents the proposed architecture table with [1,2,8,1] four stage

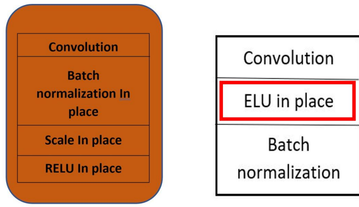

Fig. 3: SqueezeNext baseline block and Proposed Shallow SqueezeNext Basic Block.

configuration, spatial resolution, dropout layer and fully connected convolution. The bottle neck module used in the proposed architecture uses a different basic block (Fig. 2) in comparison to squeezenext baseline architecture.

A. Dropout layer

Dropout is a technique used to improve over fit on neural networks. It is a regularization method that approximates training a large number of neural networks with different parallel architectures. Large neural nets trained on relatively small datasets can over fit the training data. This has the effect of the model learning the statistical noise in the training data, which results in poor performance and increase generalization errors due to over fitting. The approach to reduce over fitting is to fit all possible different neural networks on the same dataset and to average the predictions from each model. In the proposed architecture, the dropout layer is used before the spatial resolution layer followed by the average pooling layer. It is observed during the experiments conducted that the dropout layer performs better than an additional batch normalization layer.

B. Resolution Multiplier

This hyper-parameter, resolution multiplier, is used to reduce the computational cost of a neural network. It is applied s, width multiplier, and resolution multiplier reduce the cost and parameters.

C. Width Multiplier

Width multiplier is used in order to construct these smaller and less computationally expensive models. The role of the width multiplier is to thin a network uniformly at each layer. The typical settings of width multiplier are 1, 0.75, 0.5 and 0.25. Width multiplier has the effect of reducing computational cost and the number of parameters quadratically by roughly twice the power of the width multiplier term. Width multiplier can be applied to any model structure to define a new smaller model with a reasonable accuracy and size trade off. It is used to define a new reduced structure that needs to be trained from scratch.

IV. Hardware and Software used

- Intel i9 9th generation processor with 32 GB RAM.
- Required memory for dataset and results: 4GB.
- Aorus Geforce RTX 2080Ti GPU.

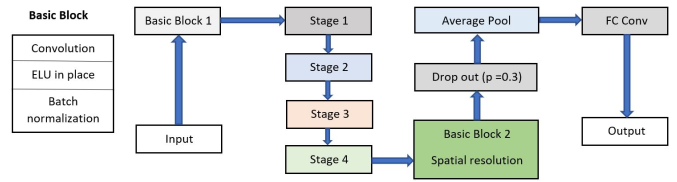

Fig. 4: Illustration of Basic Block (left) and Shallow SqueezeNext architectures.

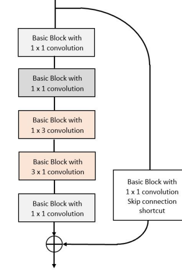

Fig. 5: Illustration of Shallow SqueezeNext's bottleneck module

- Nvidia Geforce GTX 1080Ti GPU.
- Python version 3.6.7.
- Spyder version 3.6.
- Pytorch version 1.0.
- Livelossplot (Loss and accuracy visualization).
- Netscope (SqueezeNext baseline visualization)

V. Results

A. Shallow SqueezeNext Results

It can be observed that leveraging architectural modifications led to the generation of even more efficient network architectures, as evident by the Shallow SqueezeNext having model sizes range from 4.2MB to just 0.115MB shown in Table III. A better reduced model size is achieved from baseline SqueezeNext's model size, 9.525MB to the reduced model size of the proposed Shallow SqueezeNext architecture, 0.115MB. Few major factors

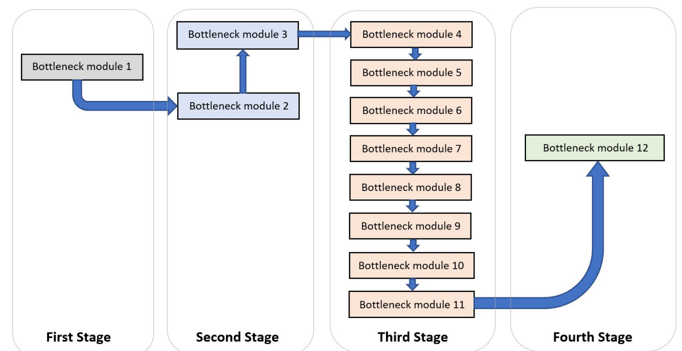

Fig. 6: Illustration of Four stage [1,2,8,1] configuration of the Shallow SqueezeNext architectures.

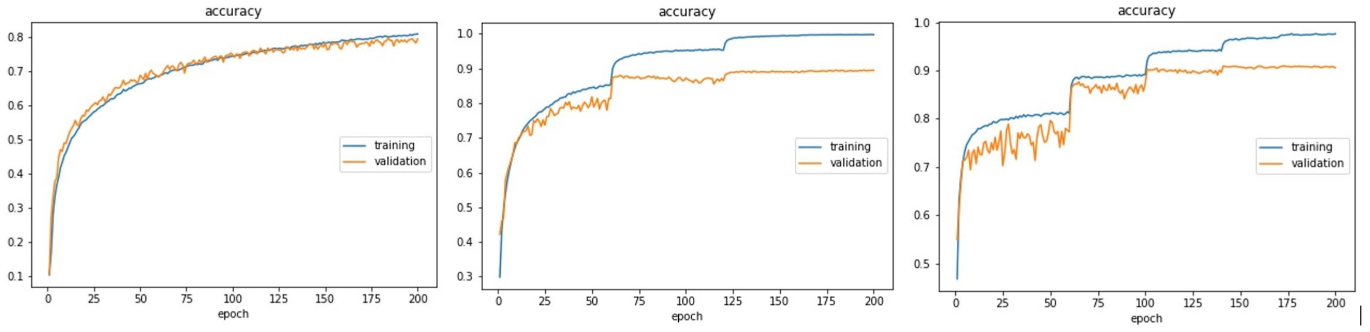

Fig. 7: 1: SqueezeNet accuracy, 2: SqueezeNext accuracy, 3: Shallow SqueezeNext accuracy

responsible for this model size reduction are different resolution and width multipliers. The ability to not only achieve very small model sizes but also fast runtime speeds has great benefits when used in resource-starved environments with limited computational, memory, and energy requirements. The width and resolution multipliers are useful modifications for producing very small deep neural network architectures that are well-suited for embedded device scenarios without the need for compression or quantization. Therefore, Shallow SqueezeNext-06-0.125 is 22X smaller than SqueezeNext-23-1x-v1 (or 26X smaller than SqueezeNext v1.0). Implementation of in place operations such as ELU in place, Relu in place and elimination of the extra max pooling layers along with a suitable resolution and width multiplier makes this architecture more compact, efficient and flexible. With a change in resolution and width multiplier this architecture can be deployed with better accuracy with a trade off with the large memory size . Each model is trained from scratch on CIFAR-10 without the use of transfer learning method. The network parameters for each model is saved and loaded from a checkpoint file for training and testing using pytorch save and load method for saving and loading a neural network checkpoint model file. Most important factor of this architecture is that it can be readily implemented on real time systems with memory constraints. The dropout layer further, improved the accuracy of the architecture. The format for Shallow SqueezeNext in all the tables in section VI illustrates the Shallow SqueezeNext architecture with resolution multiplier followed by a width multiplier and the version number. From the results of Table I, it is observed that bottleneck module makes a large difference along with appropriate use of multipliers makes the Shallow SqueezeNext with dropout layer more efficient. Table IV illustrates results attained for different values of dropout layer probabilities implemented with the proposed Shallow SqueezeNext architecture. It shows the efficient results attained for the experiments and justifies the reason to choose the dropout probability p with value equal to 0.3. In the last, 5 illustrates the accuracies for the various models trained on CIFAR-10 dataset.

TABLE I: Results comparison with SqueezeNet & SqueezeNext trained from scratch on CIFAR-10

Model	Accuracy%	Model Size(MB)	Model speed (sec)
SqueezeNet-v1.0	79.59	3.013	4
SqueezeNet-v1.1	77.55	2.961	4
SqueezeNext-23-1x-v1	87.15	2.586	19
SqueezeNext-23-1x-v5	87.96	2.586	19
SqueezeNext-23-2x-v1	90.48	9.525	22
SqueezeNext-23-2x-v5	90.48	9.525	28
Shallow SqueezeNext-v1	82.44	0.370	7
Shallow SqueezeNext-v1	82.86	1.24	8

*All results are 3 average runs with SGD, LR is 0.1

TABLE II: Shallow SqueezeNext Results with different resolution multipliers

Model	Resolution	Accuracy%	Model Size(MB)	Model speed (sec)
Shallow SqueezeNext-06-2x-v1	1111	89.35	4.21	21
Shallow SqueezeNext-06-2x-v1	1111	89.35	4.21	21
Shallow SqueezeNext-08-1x-v1	1221	77.48	2.96	4
Shallow SqueezeNext-10-1x-v1	1331	87.63	2.5863	23
Shallow SqueezeNext-12-1x-v1	1441	87.63	2.5863	23
Shallow SqueezeNext-14-1x-v1	1551	82.44	0.370	7
Shallow SqueezeNext-16-1x-v1	1661	82.86	1.24	8

*All results are 3 average runs with SGD, LR is 0.1

TABLE III: Shallow SqueezeNext Results with different width multipliers

Model	Width	Accuracy%	Model Size(MB)	Model speed (sec)
Shallow SqueezeNext-06-0.125x-v1	0.125x	66.40	0.115	7
Shallow SqueezeNext-06-0.2x-v1	0.2x	72.16	0.141	8
Shallow SqueezeNext-06-0.2x-v1	0.2x	82.47	0.296	13
Shallow SqueezeNext-06-0.3x-v1	0.3x	77.87	0.196	8
Shallow SqueezeNext-12-0.4x-v1	0.4x	87.27	0.485	13
Shallow SqueezeNext-14-0.5x-v1	0.5x	88.96	0.772	15
Shallow SqueezeNext-06-0.6x-v1	0.6x	84.63	0.48	10
Shallow SqueezeNext-07-0.7x-v1	0.7x	88.10	0.704	12
Shallow SqueezeNext-06-0.8x-v1	0.8x	87.71	0.774	12
Shallow SqueezeNext-06-0.9x-v1	0.9x	86.25	0.950	12
Shallow SqueezeNext-12-1.0x-v1	1.0x	88.28	0.557	19
Shallow SqueezeNext-06-1.5x-v1	1.5x	82.44	2.44	17
Shallow SqueezeNext-06-2.0x-v1	2.0x	89.35	4.20	21

TABLE IV: Shallow SqueezeNext Results with different dropout layer probabilities

Model	dropout p	Accuracy%	Model Size(MB)	Model speed (sec)
Shallow SqueezeNext-06-0.1x-v1	0.1	80.82	0.273	4
Shallow SqueezeNext-06-0.2x-v1	0.2	81.44	0.273	4
Shallow SqueezeNext-06-0.3x-v1	0.3	81.87	0.273	4
Shallow SqueezeNext-06-0.4x-v1	0.4	81.86	0.273	4
Shallow SqueezeNext-06-0.5x-v1	0.5	81.70	0.273	4

TABLE V: Shallow SqueezeNext Results

Model	Width, Resolution	Accuracy%	Model Size(MB)	Model speed (sec)
Shallow SqueezeNext-14-1 5x-v1	1 5x, 1281	91.41	8.72	22
Shallow SqueezeNext-21-0.2x-v1	0.2x, 22141	90.27	1.814	27
Shallow SqueezeNext-06-0.575x-v1	0 575x, 1111	81.80	0.449	6
Shallow SqueezeNext-06-0.4x-v1	0.4x, 1111	81.97	0.2717	9
Shallow SqueezeNext-09-0 5x-v1	0 5x, 1141	87.73	0.531	11

VI. Conclusion and Discussions

The results clearly shows the tradeoff between model's speed, size and accuracy for different resolution and width multipliers. Reducing the depth of the model did not necessary decrease the model accuracy. Replacing ReLU with ELU-in-place has a good impact on learning and accuracy of the model. Choice of hyperparameters, resolution, and width multipliers are the key in reducing the losses, attaining a better model size, and model accuracy. In the results, SGD optimizer with momentum, decay and nestrov terms performed better as compared to other counter parts. The use of average pooling instead of max pooling layer along with the use of drop out layer, increased the model accuracy with a model speed tradeoff without affecting the model size. The proposed architecture with different resolution and width multiplier combinations can be used to create an efficient and flexible CNN model depending on the user-end application. Proposed Shallow SqueezeNext was trained and tested from scratch on CIFAR-10 with a best model size of 196 KB (15x better than SqueezeNet & 13x better than SqueezeNext baseline architectures). It also attains a best model speed (training and testing on GPU) of 4 seconds/ epoch (15 seconds better than Squeezenext and equivalent to SqueezeNet baseline). As the experiments were conducted without data augmentation or transfer learning techniques, using data augmentation and transfer learning will increase the performance of the proposed architecture.

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 ¡ 0.5MB model size.arXiv preprint arXiv:1602.07360.

[2] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai,Xiangyu Yue, Peter Jin, Sicheng Zhao, Kurt Keutzer, (2018). SqueezeNext: Hardware-Aware Neural Network Design. arXiv preprint arXiv: 1803.10615

[3] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).

[4] T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energyefficient convolutional neural networks using energyaware pruning. arXiv preprint, 2017.

[5] Ashraf, Khalid, et al. "Shallow networks for high-accuracy road object-detection." arXiv preprint arXiv:1606.01561 (2016).

[6] Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759 (2014).

[7] Denton, Emily L., et al. "Exploiting linear structure within convolutional networks for efficient evaluation." Advances in neural information processing systems. 2014.

[8] Guo, Yiwen, Anbang Yao, and Yurong Chen. "Dynamic network surgery for efficient dnns." Advances In Neural Information Processing Systems. 2016.

[9] He, Kaiming, and Jian Sun. "Convolutional neural networks at constrained time cost." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[10] Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." European conference on computer vision. Springer, Cham, 2014.

[11] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

[12] Ludermir, Teresa B., Akio Yamazaki, and Cleber Zanchettin. "An optimization methodology for neural network weights and architectures." IEEE Transactions on Neural Networks 17.6 (2006): 1452-1459.

[13] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[14] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[15] Stanley, Kenneth O., and Risto Miikkulainen. "Evolving neural networks through augmenting topologies." Evolutionary computation 10.2 (2002): 99-127.

[16] B. Wu, A. Wan, X. Yue, and K. Keutzer. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar p

[17] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. arXiv preprint arXiv:1612.01051, 2016.

[18] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. CoRR, abs/1504.04788, 2015.

[19] M. Courbariaux, J.-P. David, and Y. Bengio. Training deep neural networks with low precision multiplications. arXiv preprint arXiv:1412.7024, 2014.

[20] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. CoRR, abs/1510.00149, 2, 2015.

[21] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531, 2015

[22] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. arXiv preprint arXiv:1611.10012, 2016.

[23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. arXiv preprint arXiv:1609.07061, 2016.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093, 2014.

[25] J. Jin, A. Dundar, and E. Culurciello. Flattened convolutional neural networks for feedforward acceleration. arXiv preprint arXiv:1412.5474, 2014.

[26] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. arXiv preprint arXiv:1412.6553, 2014.

[27] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems, pages 9199, 2015.

[28] V. Sindhwani, T. Sainath, and S. Kumar. Structured transforms for small-footprint deep learning. In Advances in Neural Information Processing Systems, pages 30883096, 2015.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 19, 2015.

[30] Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton. "Cifar-10 (canadian institute for advanced research)." URL http://www. cs. toronto. edu/kriz/cifar. html (2010).

[31] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. arXiv preprint arXiv:1512.00567, 2015.

[32] Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." arXiv preprint arXiv:1811.06965 (2018).

TABLE VI: Shallow SqueezeNext architecture [1,2,8,1] four stage configuration

Layer Name	**Input Size** Wi X Hi X Ci	**Padding** Pw X Ph	**Stride**	**Filter size** Kw X Kh	**Output size** W0 X H0 X C0	**Repeat**	**Parameters**
Convolution 1	32x32x3	0x0	1	3x3	30x30x64	1	1792
Convolution 2	30x30x64	0x0	1	1x1	30x30x16	1	1040
Convolution 3	30x30x16	0x0	1	1x1	30x30x8	1	136
Convolution 4	30x30x8	0x1	1	1x3	30x30x16	1	400
Convolution 5	30x30x16	1x0	1	3x1	30x30x16	1	784
Convolution 6	30x30x16	0x0	1	1x1	30x30x32	1	544
Convolution 32	30x30x32	0x0	2	1x1	30x30x32	1	1056
Convolution 33	15x15x32	0x0	1	1x1	15x15x16	1	528
Convolution 34	15x15x16	0x1	1	1x3	15x15x32	1	1568
Convolution 35	15x15x32	1x0	1	3x1	15x15x32	1	3104
Convolution 36	15x15x32	0x0	1	1x1	15x15x64	1	2112
Convolution 37	15x15x64	0x0	1	1x1	15x15x32	1	2080
Convolution 38	15x15x32	0x0	1	1x1	15x15x16	1	528
Convolution 39	15x15x16	1x0	1	3x1	15x15x32	1	1568
Convolution 40	15x15x32	0x1	1	1x3	15x15x32	1	3104
Convolution 41	15x15x32	0x0	1	1x1	15x15x64	1	2112
Convolution 62	15x15x64	0x0	2	1x1	15x15x64	1	4160
Convolution 63	8x8x64	0x0	1	1x1	8x8x32	1	2080
Convolution 64	8x8x32	1x0	1	3x1	8x8x64	1	6208
Convolution 65	8x8x64	0x1	1	1x3	8x8x64	1	12352
Convolution 66	8x8x64	0x0	1	1x1	8x8x128	1	8320
Convolution 67	8x8x128	0x0	1	1x1	8x8x64	7	57792
Convolution 68	8x8x64	0x0	1	1x1	8x8x32	7	14560
Convolution 69	8x8x32	1x0	1	3x1	8x8x64	7	43456
Convolution 70	8x8x64	0x1	1	1x3	8x8x64	7	86464
Convolution 71	8x8x64	0x0	1	1x1	8x8x128	7	58240
Convolution 102	8x8x128	0x0	2	1x1	8x8x128	1	16512
Convolution 103	4x4x128	0x0	1	1x1	4x4x64	1	8256
Convolution 104	4x4x64	0x1	1	1x3	4x4x128	1	24704
Convolution 105	4x4x128	1x0	1	3x1	4x4x128	1	49280
Convolution 106	4x4x256	0x0	1	1x1	4x4x256	1	65792
Convolution 107 Spatial Resolution	4x4x256	0x0	1	1x1	4x4x128	1	32896
Dropout (p=0.5)	4x4x256	-	-	-	4x4x256	1	-
Average Pool	4x4x256	-	-	-	4x4x256	1	-
Fully Connected Convolution	1x1x128	0x0	1	1x1	1x1x10	1	1290

* Wi, Hi, Ci refer to input width, height and number of channels, Pw, Ph refer to padding width and height, Kw, Kh refer to filter or kernel width and height, W0, H0, C0 refer to output width, output height and output number of channels