Multi-Fidelity Deep Neural Networks for Adaptive Inference in the Internet of Multimedia Things.

Sam Leroux*, Steven Bohez, Elias De Coninck, Pieter Van Molle, Bert Vankeirsbilck, Tim Verbelen, Pieter Simoens, Bart Dhoedt

Authors are with Ghent University - imec, IDLab, Department of Information Technology

iGent Tower - Department of Information Technology
Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium

Abstract

Internet of Things (IoT) infrastructures are more and more relying on multimedia sensors to provide information about the environment. Deep neural networks (DNNs) could extract knowledge from this audiovisual data but they typically require large amounts of resources (processing power, memory and energy). If all limitations of the execution environment are known beforehand, we can design neural networks under these constraints. An IoT setting however is a very heterogeneous environment where the constraints can change rapidly. We propose a technique allowing us to deploy a variety of different networks at runtime, each with a specific complexity-accuracy trade-off but without having to store each network independently. We train a sequence of networks of increasing size and constrain each network to contain the parameters of all smaller networks in the sequence. We only need to store the largest network to be able to deploy each of the smaller networks. We experimentally validate our approach on different benchmark datasets for image recognition and conclude that we can build networks that support multiple trade-offs between accuracy and computational cost.

Keywords: IoT, Deep neural networks, resource efficient inference

*Corresponding author
Email address: sam.leroux@ugent.be (Sam Leroux)

1. Introduction

The Internet of Things (IoT) entails the promise of a world with billions of interconnected devices, each with sensors that continuously monitor our homes, offices and streets. These devices will generate massive amounts of information allowing us to analyze and optimize our environments in unprecedented ways.

There are many challenges involved when building large scale sensor networks. The devices themselves need to be small and affordable yet they also need to be robust and reliable. They need to capture and transmit fine-grained information without consuming large amounts of energy and they need to be easy to deploy and to maintain without intensive manual interaction.

Typical sensors that are found in these IoT deployments are temperature, humidity, pressure and motion sensors. With the right analytic tools these sensors can already provide a wealth of information about their surroundings but we will also need additional rich sensors such as cameras and microphones to fully understand our environment. A smart traffic camera with license plate detection capabilities can recognize a stolen car or it can detect a traffic jam. Smart security cameras with facial recognition software can trigger alerts when unauthorized persons are detected. Voice recognition is already used in smart personal assistants and microphones are increasingly used as sensors for example to detect gunshots in urban environments[1].

These multimedia sensors generate large amounts of high dimensional data [1] and extracting useful high level insights typically requires computationally intensive techniques. It is usually not an option to offload the computations to a cloud back-end since the latency and communication cost of transmitting the

[1] https://www.washingtonpost.com/news/true-crime/wp/2017/05/10/how-shotspotter-locates-gunfire-helps-police-catch-shooters-and-denormalize-gun-violence

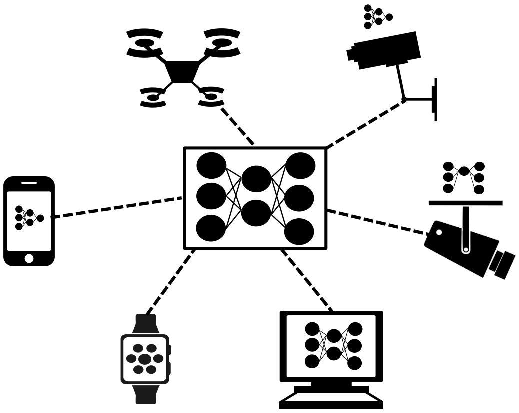

Figure 1: An IoT environment is characterized by a large amount of heterogeneous devices. We store a single set of weights and are able to deploy a different optimized neural network to each device without having to train or store all these networks independently.

data to the cloud would be prohibitively large for some applications. In addition the raw audio and video recordings are often privacy sensitive and should not leave the local device [2].

In this work we look at techniques that could perform these types of operations locally on an Internet of Things device. We focus on deep learning techniques since these are arguably the state of the art methods to extract information from high dimensional observations. Deep learning is typically not a good match for the resource constrained IoT devices since deep learning models require large amounts of memory and computational power. A lot of progress however has been made in reducing the computational cost of deep learning models (we refer to section 2 for an overview).

An IoT setting is characterized by a large amount of heterogeneous devices. Therefore we argue that it is not sufficient to have one optimized neural network with a fixed computational cost. Instead we should be able to optimize the neural network for each specific device. In addition we would also like to trade-off the computational cost and accuracy of the network at runtime. This would allow us to adjust the required resources based on external factors such as tolerable latency, battery level of the device or required accuracy. The trivial solution would be to train a number of different neural networks each with a different configuration and to deploy the network that best fits the current needs. Training all these different networks will take time but even worse, the overhead involved in storing all these different networks on an embedded device in the network might be prohibitively large. A single neural network for large scale image recognition can quickly require millions of parameters adding up to hundreds of megabytes of storage.

We propose to train a sequence of models, increasing in size but instead of training all networks independently we constrain the larger models to contain all parameters of the smaller networks as subsets. Hence, just one set of weights needs to be stored (i.e. the weights of the largest model in the sequence) while we can use subsets of these weights to deploy smaller networks.

We presented the concept of a runtime configurable neural network before in a conference paper [3]. We now extend this work with a more thorough evaluation on different network architectures and datasets including neural networks trained on the Imagenet dataset. We also provide a guideline on how to choose the different subnetworks depending on the bottlenecks of the environment.

The remainder of this paper is organized as follows. We give an overview of related work in section 2. We introduce our multi-fidelity architecture in section 3 and we experimentally validate our approach on different benchmark datasets in section 4. We finally conclude in section 5 with a short summary of possible future research directions.

2. Related work

Neural networks have been around for a long time but recent advances in technology such as efficient GPU implementations and large labelled datasets have renewed interest. Deep neural networks are the current state of the art for image and speech recognition [4]. We refer to [5] for an in depth overview of the history of neural networks and deep learning.

One problem with deep neural networks is the amount of resources they require during training and inference. Training a neural network is the computationally most expensive part. In most cases however training can be done offline on high performance GPU systems where energy consumption is less of an issue. We instead focus on deploying a trained network on an IoT device. These devices are constrained in terms of processing power, memory or energy consumption. In these cases we need to reduce the size or complexity of the network without sacrificing too much accuracy. Various approaches have been proposed to achieve these goals.

2.1. Reducing the memory footprint

Deep neural networks for image classification easily require millions of parameters (hundreds of megabytes). This makes it hard to deploy a neural network to a device with limited memory or to incorporate a neural network into a mobile app. There is a considerable amount of prior work that has focussed on reducing the memory footprint of deep neural networks. Han et al. presented a three stage pipeline that is able to reduce the storage requirement of neural networks by 35x to 49x without affecting the test accuracy [6]. They succeed in reducing the size of the Alexnet [7] and VGG16 [8] architectures from 240MB and 552MB to 6.9MB and 11.3MB respectively.

Chen et al. proposed an elegant Hashing based approach [9]. They use a hashing function that groups the weights into a small number of buckets. All connections belonging to the same bucket share a single weight value. This technique was extended in a follow up paper [10] in which they also apply the hashing trick to the convolutional layers. The authors argue that the weights of the convolutional filters are typically smooth and low-frequency. They first convert the weights to the frequency domain and then use the hash function to group the parameters in hash buckets.

Other approaches to reduce the memory footprint of a neural network include low rank decomposition of the weight matrices [11] or Structured Matrices [12], $m \times n$ matrices that can be described using less than mn parameters because they follow a certain structure. This dramatically reduces the memory footprint but also supports faster matrix multiplications which can accelerate inference and training.

2.2. Reducing the computational cost

Reducing the number of weights in a neural network does not necessarily reduce the computational cost. Most of the parameters are used in the fully connected layers while the convolutional layers are responsible for most operations.

One approach is to introduce sparsity between the layers. In [13] the authors use a sparse connection matrix for the convolutional layers where each output channel is only connected to a small subset of the input channels. Operations on sparse data may need less operations in theory but since most implementations are optimized for dense matrix operations this may not result in a reduction in latency or in an increase in throughput.

Other techniques try to transfer the knowledge stored in large models (teacher) to a smaller more efficient (student) network. Hinton et al. proposed to use the soft outputs of the teacher (the probability distribution over the classes) as a soft target for the student [14]. A soft target includes information about the similarities between classes which can make it easier to optimize the student. This was later extended by Romero et al. Their Fitnets [15] uses the intermediate representations to guide the student in addition to the soft targets.

Most implementations of deep neural networks use 32 bit floating point numbers for weights and activations. Various works have shown that this is not necessary and that 8 bit fixed point integers [16] are usually sufficient. This reduces the memory footprint and allows for a very efficient implementation in hardware. Other works further reduce the precision of the weights to 4 bits (for convolutional layers) or even to two bits (for fully connected layers) [6]. It is even possible to use binary weights and activations [17] which allows for a very efficient implementation in hardware since the floating point operations can be replaced with logical operations.

In this contribution we take a different approach and instead focus on reconfigurability. We argue that it is not enough to train one optimized network because an IoT setting is a very heterogeneous environment where different devices require different trade-offs that might change at runtime. It is however possible to combine our approach with other techniques such as reducing the precision of the weights to further reduce the memory footprint or computational cost.

3. Architecture

Our goal is to train a set of different neural networks each with their own accuracy v.s. cost trade-off. All networks in the set share the same structure (same number and types of layers) but they will have a different number of parameters (see Figure 2). We constrain every network to contain the exact weights of all smaller networks. We start by training a very small network and then gradually add additional parameters to each layer while keeping the already trained parameters fixed. As a result, the largest network in the sequence will contain all the weights of the previous networks as subsets of its own weight matrices. To deploy the networks we only need to store one set of weights (the weights of the largest network that we want to use) and we can use a subset of these weights to deploy smaller versions.

Figure 3a shows how this is done for a fully connected layer. A fully connected layer with n neurons computes $x \cdot W + b$ where x is a one dimensional input

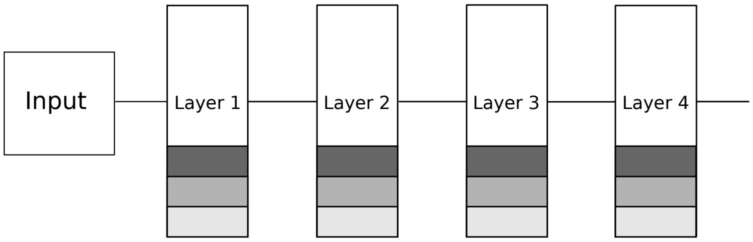

Figure 2: For our multi-fidelity architecture we train a set of weights that can be used to deploy different networks, each with their own accuracy and computational cost trade-off. Each network has the same structure but differs in the number of parameters (the width of the layers).

vector (with size m), W is the $(m \times n)$ weight matrix of the layer and b is a bias vector (with size n). If we increase the number of neurons for this layer from n to $n + k$ we reshape the weight matrix and bias vector for this layer and we initialize the new values with random values (grey blocks in figure 3a). We also need to reshape the weight matrix of the next layer since this layer now receives a larger input.

We use the same approach for convolutional layers. The weight matrix for a layer with n kernels is now a 4D tensor $(n \times c \times w \times h)$ with c the number of channels in the input and w and h the width and height of the convolutional kernels. If we increase the number of kernels in one layer from n to $n + k$ we again reshape the weight matrix and bias vector of this layer (grey blocks in Figure 3b). We also need to reshape the weight matrix of the next layer because the input of this layer will have a larger number of channels.

We use the traditional backpropagation algorithm to train the network. At each step in the backpropagation algorithm we calculate the gradient of the loss function with respect to the weights of the network and update them to minimize the error. We make sure to only update the new parameters by putting all updates to zero except those that correspond to the new parameters. We

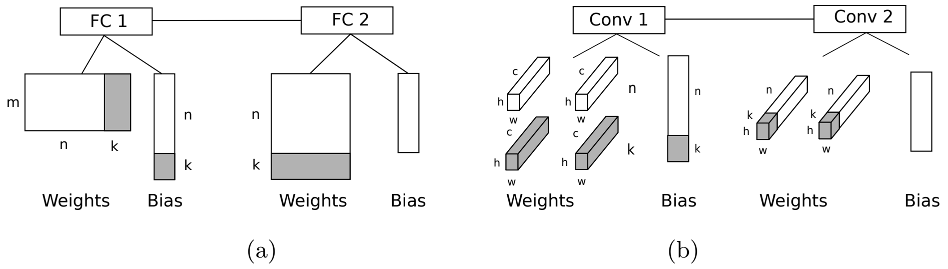

Figure 3: Schematic overview of how the weight tensors change when we scale the number of neurons in a fully connected layer (a) or the number of filters in a convolutional layer (b) from n to $n+k$. For both cases we increase the size of the weight tensor and bias vector. This changes the dimensionality of the output of the layer which is why we also need to change the size of the weight tensor of the next layer.

simply multiply the gradient descent updates with a mask that contains zero values except for those positions that correspond to the new weights. For example in the first fully connected layer of Figure 3a where we increase the number of neurons from n to $n + k$ we calculate the gradient updates with respect to every element of the $m * (n + k)$ weight tensor and $n + k$ bias vector but put all elements to zero except those $m * k$ weights and k bias values that correspond to the grey parts of the tensor.

In the previous paragraph we explained the iterative training routine where we start with the smallest network of the sequence and each time add additional parameters and retrain them without changing the already trained weights. We also experimented with an alternative training approach that trains the different networks all at once. During training we simply select one configuration at random for each batch of training data and only update the weights of this specific subnetwork. This update might reduce the performance of the other networks that share a subset of the weights but we found that during training the networks learn to co-adapt. This approach is easy to implement but results in slightly lower accuracies than the first training technique.

4. Experimental validation

In this section we apply our approach to different benchmark problems. We focus on image classification since this typically requires large networks to process the high dimensional input images.

All our experiments were performed using Pytorch [2]. We trained the networks on NVIDIA GTX1080 GPUs. The reported runtimes were all measured on an Intel Edison[3] device. The Intel Edison is an ultra-small embedded computing platform with an Intel® Atom™ SoC dual-core CPU and integrated WiFi and Bluetooth LE. These features combined with a peak power consumption of under 1W make this platform an interesting computing platform for various IoT applications.

4.1. Small scale experiments on CIFAR10, CIFAR100 and SVHN

In our first experiments we validate our approach on three small scale datasets for image recognition: CIFAR10 [18], CIFAR100 [18] and SVHN [19]. All three datasets are similar in size and they all contain 32 by 32 RGB images. The CIFAR10 dataset contains images from ten classes such as "cat", "dog", "car" and "plane". The CIFAR100 dataset is similar but contains images from 100 classes. The Street View House Numbers (SVHN) dataset contains small cropped digits obtained from house numbers in Google Street View images. The number of train and test samples in each dataset are shown in table 1.
We applied our multi-fidelity approach to a VGG16-like [8] network. The VGG16 architecture was used by the Visual Geometry Group (VGG) at Oxford University in the 2014 ImageNet competition. We used the same network architecture for all three datasets. We trained the network at four different scales following the approach from the previous section. For each scale we divide the number of convolutional filters and neurons of each layer by the same number.

[2] Pytorch website: `http://pytorch.org/`

[3] Intel Edison Wikipedia article: `https://en.wikipedia.org/wiki/Intel_Edison`

	CIFAR10	CIFAR100	SVHN
Number of classes	10	100	10
Number of train images	50 000	50 000	73 257
Number of test images	10 000	10 000	26 032

Table 1: Properties of the small scale datasets.

Scale 1/4 for example means that each layer contains one-fourth of the number of convolutional kernels or neurons of the same layer in the full network. The results are summarized in table 2. For each scale we report the number of parameters, the execution time on the Intel Edison platform and the classification accuracy on the three datasets. The smallest version (scale 1/8) takes only 2% of the execution time of the original network measured on the Intel Edison (60ms vs 3400ms) and requires only 1.7% of the number of parameters. Compared to the largest scale, the accuracy drops by 10% for the CIFAR10 and CIFAR100 datasets and by 5% for the SVHN dataset which is an easier task. By doubling the size to one-fourth of the original size we increase the number of parameters five-fold to just over 1 million (4 MB if they are stored as 32 bit floating point). The execution time increases four-fold to 224 ms. The accuracy increases with 5% for CIFAR10 and CIFAR100 but only by 1% for the SVHN dataset. Doubling the scale to 1/2 and again to the full size increases the number of parameters and execution time four-fold each time.

For each scale we also report the accuracies for a network of the same scale but now trained independently of the other scales. The accuracy of the single scale networks is consistently slightly higher than the multiscale versions because the optimization problem is much harder when we constrain the network to contain all parameters of the smaller networks. This penalty is however a small price to pay for the freedom of having 4 different configurations contained in one set of weights.

Scale	Parameters	Time (ms)	CIFAR10	CIFAR100	SVHN
1/8	280 460	60	82.3%	62.7%	92.7%
1/4	1 061 428	224	87.5%	68.4%	93.5%
1/2	4 125 188	870	90.2%	70.2%	96.0%
1	16 260 004	3400	91.4%	71.7%	97.1%
1/8 (single scale)	280 460	60	82.1%	62.8%	93.0%
1/4 (single scale)	1 061 428	224	87.8%	69.2%	94.6%
1/2 (single scale)	4 125 188	870	90.5%	70.5%	96.8%
1 (single scale)	16 260 004	3400	92.4%	73.1%	97.8%

Table 2: Results for the VGG16 network on the small scale datasets. The first part of the table shows the results of our multi-fidelity approach where the weights are shared between the scales. The second part shows the results of the same networks but now trained independently without the constraint that weights have to be shared between them.

4.2. Imagenet

The previous experiments used small scale datasets that are not really representative of real world applications. In this section we apply our method to the ILSVRC2012 dataset[20]. This dataset contains 1.2 million training images in 1000 classes.

We use the Alexnet architecture [7] for this dataset. This architecture has five convolutional layers and three fully connected layers. The full Alexnet architecture requires 62 million parameters which corresponds to 250 MB of storage when all weights are stored as 32 bit floating point numbers. The original architecture requires 7E+08 floating point operations for a single image.

In the previous sections we used a rather naive way of building the subnetworks by just adding a fixed number of convolutional filters or neurons to each layer. This is far from optimal since the different layers all have very different characteristics. There are three properties that we have to take into account as we decide on an optimal architecture for a certain device:

- The size of the weights. If the storage capacity of the device or the bandwidth of the network connection needed to download the weights is the bottleneck, we can gain most by reducing the number of weights in the network.
- The size of the intermediate representations. In addition to storing the weights we also need memory to temporarily store the intermediate activations of the network.
- The number of operations. This directly determines the latency and energy consumption of the network.

Figure 4 shows these three measurements for each layer in the Alexnet architecture, relative to the entire network. Most of the parameters are needed for the fully connected layers at the end of the network (the first fully connected layer alone is responsible for 63% of the parameters). This is a common observation that can be made for most convolutional neural networks for image classification [21]. When the size of the weight vectors is the bottleneck, for example when the network needs to be downloaded to a device over a slow or costly network connection, we can gain most by reducing the number of parameters in these layers.

The number of floating point operations on the other hand is dominated by the convolutional layers. The fully connected layers are almost negligible when the computation itself is the bottleneck. In these cases we can have the highest impact by reducing the number of filters in the first convolutional layers. The largest intermediate vectors are created by the first convolutional layers and reducing the number of convolutional filters in these layers will have the largest impact on the required memory.

To build our different networks we focus on the Conv1, Conv2 and FC1 layers because these have the largest impact on the needed memory, on the number of operations and on the number of parameters respectively. Note that changing the number of parameters in one layer also has an effect on the number of parameters and number of operations of the next layer.

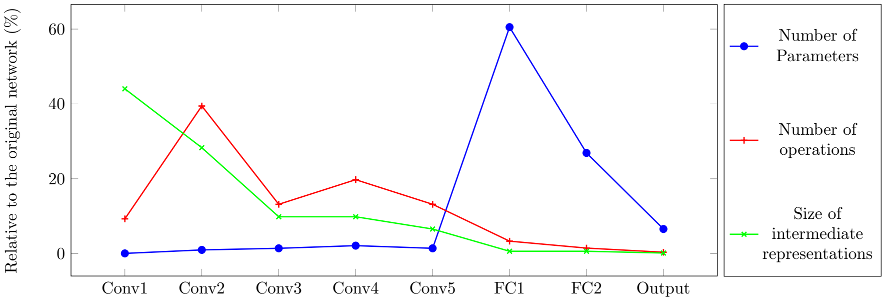

Figure 4: Analysis of the number of operations, number of parameters and the size of the intermediate outputs for each layer in the Alexnet Network.

We trained a multi-fidelity version of the Alexnet network with 16 and 64 filters for the first convolutional layer (C1), 48 and 192 filters for the second convolutional layer (C2) and 512 and 4096 neurons in the first fully connected layer (FC1). This results in eight different combinations each with a computational cost and corresponding accuracy. Table 3 lists the different subnetworks contained in one set of weights. The first row shows the properties of the original Alexnet network. The second row shows the same network but now trained with the constraint that it needs to contain all parameters of all seven smaller networks. The optimization problem for our multi-fidelity network is much harder because of these constraints which explains the 2% drop in accuracy compared to the full model trained from scratch. The following rows show various subnetworks that we have at our disposal thanks to the multi-fidelity architecture. We again report the time needed to forward one image through the network on the Intel Edison platform. We report both the Top 1 and Top 5 accuracy.

Even though we only change three layers, we end up with 8 distinctly different networks, each with there own trade-offs. If we focus on execution time for example, we can reduce the latency by half and the accuracy drops from 55.6% to 47%. We can also reduce the number of parameters to one-fifth of the original number. This reduces the accuracy by 5%. The execution time stays the same since we only removed neurons from the fully connected layers and these layers have a minimal effect on computational cost.

C1	C2	FC1	Parameters	Largest intermediate representation	FLOPS	Time (s)	Top 1	Top 5
64	192	4096	$6.1e7$	$1.9e5$	$7e8$	23	55.6%	78.2%
64	192	4096	$6.1e7$ (100%)	$1.9e5$ (100%)	$7e8$ (100%)	23	53.3%	76.5%
64	192	512	$1.3e7$ (21%)	$1.9e5$ (100%)	$6.7e8$ (95%)	22	50.1%	74.7%
64	48	4096	$6.1e7$ (100%)	$1.9e5$ (100%)	$4.6e8$ (65%)	17	49.9%	73.7%
64	48	512	$1.3e7$ (21%)	$1.9e5$ (100%)	$4.1e8$ (58%)	16	49.8%	71.7%
16	192	4096	$6.1e7$ (100%)	$1.4e5$ (74%)	$4.9e8$ (70%)	17	50.1%	73.7%
16	192	512	$1.3e7$ (21%)	$1.4e5$ (74%)	$4.5e8$ (64%)	16	46.5%	71.5%
16	48	4096	$6.0e7$ (98%)	$6.5e4$ (34%)	$3.7e8$ (52%)	11	47.0%	71.1%
16	48	512	$1.2e7$ (20%)	$6.5e4$ (34%)	$3.2e8$ (46%)	10	43.6%	68.8%

Table 3: Different configurations of our Alexnet network and the corresponding accuracy. We change the number of convolutional filters in the first and second convolutional layer (C1 and C2) and the number of neurons in the first fully connected layer (FC1) because these layers have the largest impact on the size of the intermediate activations, on the FLOPS and on the number of parameters respectively.

5. Conclusion and future work

Applying deep neural networks to IoT sensor data is an interesting research direction. In this article we argue that because IoT environments are continuously changing heterogeneous environments we need more than one network with a fixed computational cost and corresponding accuracy. We propose to train multiple networks and to constrain them to share parameters. We only need to store one set of weights to be able to deploy multiple versions of the same network. We evaluated our approach on four well known image classification datasets and found that it is indeed possible to have a variety of network configurations contained in one set of weights.

Future work could focus on optimizing the different splits of the network. In this work we provided some heuristics on how to choose these based on a manual inspection of the different layers but it would be interesting to find a method that could find the most interesting splits automatically for any given network architecture.

Acknowledgements

Steven Bohez is funded by a Ph.D. grant of the Agency for Innovation by Science and Technology in Flanders (IWT). We gratefully acknowledge the support of NVIDIA Corporation with the donation of GPU hardware used for this research.

References

[1] W. Hou, Z. Ning, L. Guo, X. Zhang, Temporal, functional and spatial big data computing framework for large-scale smart grid, IEEE Transactions on Emerging Topics in Computing.

[2] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, B. Amos, Edge analytics in the internet of things, IEEE Pervasive Computing 14 (2) (2015) 24–31.

[3] S. Leroux, S. Bohez, E. De Coninck, T. Verbelen, B. Vankeirsbilck, P. Simoens, B. Dhoedt, Multi-fidelity matryoshka neural networks for constrained iot devices, in: Neural Networks (IJCNN), 2016 International Joint Conference on, IEEE, 2016, pp. 1305–1309.

[4] Y. Bengio, et al., Learning deep architectures for ai, Foundations and trends® in Machine Learning 2 (1) (2009) 1–127.

[5] J. Schmidhuber, Deep learning in neural networks: An overview, Neural networks 61 (2015) 85–117.

[6] S. Han, H. Mao, W. J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, arXiv preprint arXiv:1510.00149.

[7] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in neural information processing systems, 2012, pp. 1097–1105.

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

[9] W. Chen, J. Wilson, S. Tyree, K. Weinberger, Y. Chen, Compressing neural networks with the hashing trick, in: International Conference on Machine Learning, 2015, pp. 2285–2294.

[10] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, Y. Chen, Compressing convolutional neural networks, arXiv preprint arXiv:1506.04449.

[11] M. Denil, B. Shakibi, L. Dinh, N. de Freitas, et al., Predicting parameters in deep learning, in: Advances in Neural Information Processing Systems, 2013, pp. 2148–2156.

[12] V. Sindhwani, T. Sainath, S. Kumar, Structured transforms for small-footprint deep learning, in: Advances in Neural Information Processing Systems, 2015, pp. 3088–3096.

[13] S. Changpinyo, M. Sandler, A. Zhmoginov, The power of sparsity in convolutional neural networks, arXiv preprint arXiv:1702.06257.

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

[15] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, Y. Bengio, Fitnets: Hints for thin deep nets, arXiv preprint arXiv:1412.6550.

[16] V. Vanhoucke, A. Senior, M. Z. Mao, Improving the speed of neural networks on cpus, in: Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop, Vol. 1, 2011, p. 4.

[17] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1, arXiv preprint arXiv:1602.02830.

[18] A. Krizhevsky, G. Hinton, Learning multiple layers of features from tiny images.

[19] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A. Y. Ng, Reading digits in natural images with unsupervised feature learning, in: NIPS workshop on deep learning and unsupervised feature learning, Vol. 2011, 2011, p. 5.

[20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, L. Fei-Fei, ImageNet Large Scale Visual Recognition Challenge, International Journal of Computer Vision (IJCV) 115 (3) (2015) 211–252. `doi:10.1007/s11263-015-0816-y`.

[21] V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, Efficient processing of deep neural networks: A tutorial and survey, arXiv preprint arXiv:1703.09039.