View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Ghent University Academic Bibliography

Multi-Fidelity Matryoshka Neural Networks for Constrained IoT Devices

Sam Leroux, Steven Bohez, Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, Bart Dhoedt
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
{sam.leroux, steven.bohez, elias.deconinck, tim.verbelen, bert.vankeirsbilck, pieter.simoens, bart.dhoedt }@intec.ugent.be

***Abstract*—Using deep neural networks on resource constrained devices is a trending topic in neural network research. Various techniques for compressing neural networks have been proposed that allow evaluating a large neural network on a device with limited memory and processing power. These approaches usually generate a single compressed *student* network based on a larger *teacher* network. In some cases a more dynamic trade-off may be desired. In this paper we trained a sequence of increasingly large networks where each network is constrained to contain the unmodified features of all smaller networks. The weight matrix of the largest network has submatrices that correspond to the weight matrices of each of the smaller networks. This technique allows us to keep the parameters of several networks in memory while having the same memory footprint as the single largest network. A trade-off between accuracy and speed can be made at runtime. The proposed approach is validated on two image classification tasks running on a real-world Internet-of-Things (IoT) device.**

I. Introduction

Speeding up the recall phase of a deep neural network (DNN) is a crucial part of deploying a large neural network in practical applications. This is especially important when neural networks are used on constrained devices such as smartphones or Internet of Things (IoT) devices. The processing power and memory available to these local devices is extremely cramped compared to the resources available in high performance cloud servers. It may however still be useful to evaluate a trained neural network on a constrained device locally since this avoids sending data to the cloud which involves costs, latency as well as potential privacy issues.

In this paper we present a technique to train a sequence of similar neural networks with an increasing number of features per layer. A certain network in the sequence has all the features of the smaller networks and some additional features. As a consequence, the weight matrix of the last network in the sequence has submatrices which correspond to the weight matrices of all other networks. This technique allows us to keep the weights of dozens of different networks in memory while having the same memory footprint as the single largest network in the sequence.

It may be desirable in practical applications to support a trade-off between the accuracy and the speed of a deep neural network, especially when the network is used on a memory, energy or processing power constrained device. We can train a sequence of networks where the parameters of the largest network exactly fit in the memory available. We can use a subset of these weights to reconstruct a smaller but less accurate network when a higher throughput is needed or when the network needs to be deployed on a device with fewer resources. This runtime trade-off between accuracy and processing power is highly relevant for robotics. A less accurate network could be used depending on the expected duration of a robot's mission or on the remaining battery power available.

This technique of reusing all the weights of the smaller networks in the largest network is especially useful for hardware implementations of neural networks. Neuromorphic hardware [1] is custom hardware designed for evaluating a neural network. These hardware implementations are much faster and energy efficient compared to neural networks implemented in software. They are also relatively expensive, hard to program and the weights are usually fixed. The same holds for neural networks implemented on Field Programmable Gate Arrays (FPGAs). Although these hardware platforms are able to exploit the inherent parallelism of a neural network, the execution time still depends on the number of parameters in the network [2]. It is usually not possible to configure these chips to store the weights of multiple networks (fast versions and more accurate versions) since the amount of neurons that can be stored on these devices is usually limited. The technique proposed in this paper allows to configure the devices with one set of weights, the weights of the largest (most accurate) network. The weights of the smaller networks are explicitly contained in the weights of the largest network. It is possible to evaluate multiple networks of different sizes with only one set of weights. This allows the device to quickly change the active network without service interruptions.

This paper is organized as follows: In section II we review the existing state-of-the-art in neural network model compression and optimization. In section III we introduce

the core idea and the training algorithm. The experimental validation is presented in section IV, both for fully-connected and for convolutional neural networks. We conclude with a description of future work in section V.

II. Model compression: related work

Various approaches to minimise the memory and computational footprint of neural networks have been proposed before.

One of the first approaches was *Optimal Brain Damage* (OBD) [3] proposed by Le Cun et al. This technique uses second order derivative information to prune unimportant weights from the network. The results prove that neural networks store a large amount of redundant information and that these redundant connections can be safely removed with a minimal impact on accuracy. While Optimal Brain Damage dates back to the 1990s, compressing neural networks only recently became a hot topic in research because of the possibility of using deep neural networks in mobile devices [4].

One elegant technique proposed by Chen et al. reduced the memory overhead of neural networks by grouping the connections into buckets. All connections in the same bucket share the same weight value. The resulting *HashedNets* [5] allow for a large reduction in memory consumption while keeping a similar generalization performance. The idea behind HashedNets is similar to using reduced precision floating point numbers as connection weights. Various works show that 32 bit floating point numbers are not needed, 16 bit [6] [7] or even 8 bit [8] floating points are accurate enough for most deep learning applications.

A similar impressive result was obtained by Han et al [9]. They applied a three-step method to achieve a reduction in the storage and computational requirements by an order of magnitude. The first step in the training algorithm is learning which connections are important and which are redundant. The second step is to prune the less important connections. The network is retrained in the final step to finetune the weights in the remaining connections. The technique is validated on AlexNet and VGGNet on the ImageNet dataset. The number of connections can be reduced up to 13x without loss of accuracy.

It is also possible to train a single *student* network to mimic a larger *teacher* network [10] or even an ensemble of networks [11]. This technique, known as *Knowledge distillation* was extended by Romero et al. Their *FitNets* [12] use the knowledge distillation technique to train a thin but deep student network using not only the outputs but also the intermediate representations of the teacher network. They are able to train student networks that can outperform the teacher network in terms of accuracy while having ten times fewer parameters.

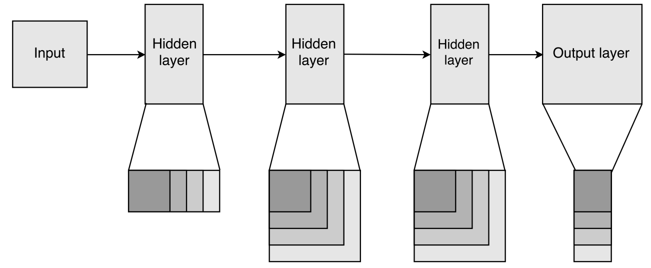

Fig. 1. The weight matrices of each layer have submatrices that correspond to the weight matrices of the smaller networks. All the smaller networks are contained within the largest network just like a Russian nesting doll (a Matryoshka doll).

These techniques allow for an impressive reduction in the needed processing power and memory but they are static solutions, they generate a single compressed version of a large network. A more dynamic trade-off between accuracy and speed is desirable in robotics and IoT use cases.

III. Matryoshka Neural Networks

The core idea is to first train a small fast network with a certain number of hidden layers. Then, a slightly larger network with the same number of layers but with more neurons in each layer. Only the new connections are changed during backpropagation while the weights from the smaller network are frozen. The weights in a fully connected layer are organised in an $m * n$ matrix where m is the number of inputs (neurons in the preceding layer) and n is the number of neurons in the current layer. When a single additional neuron is added to a certain layer l_i, the weight matrix of l_i is extended by one column and the weight matrix of l_{i+1} is extended by one row. Figure 1 illustrates the concept. The darkest gray blocks represent the weight matrices of the smallest network. Additional neurons were added to each hidden layer to obtain the larger weight matrices but the original weights were kept fixed as a submatrix. All the smaller networks are contained within the largest network just like a Russian nesting doll. The dimensionality of the input and the output of the network is fixed. As a consequence, the weight matrices of the first hidden layer and the output layer can only grow in width, respectively in height.

The same technique can be applied to convolutional neural networks. The parameters are stored in four dimensional tensors $n * c * w * h$ where n is the number of filters in the layer, c is the number of input channels (= the number of filters in the previous layer) and w, h are the width, respectively the height of each filter. A layer can be initialised with a certain number of filters and additional filters can be added by adjusting the size of the tensor and by training the additional parameters while keeping the already trained parameters fixed.

We train these networks by modifying the training procedure as follows. At each step in the backpropagation algorithm the error E is calculated. The gradient ∇E of the error function with respect to the weights w_i is calculated and the parameters of the network are updated to minimise the error. Instead of a global learning rate λ, an individual learning rate λ_i is used. This individual learning rate is set to zero for each parameter which is also a parameter of a smaller network. Thus only a small portion of the parameters are allowed to change.

$$\nabla E = \left(\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n} \right) \quad (1)$$

$$\Delta w_i = -\lambda_i \frac{\partial E}{\partial w_i} \quad (2)$$

where

$$\lambda_i = \begin{cases} 0, \text{if } w_i \text{ is a weight of a previous smaller network} \\ \lambda, \text{otherwise, eg. } 0.001 \end{cases} \quad (3)$$

We have implemented this in existing tools by defining a masking matrix for each weight matrix. The masking matrix contains a zero for each parameter that is not allowed to change and a one for each tunable parameter. Before modifying the weights, the masking matrix is multiplied element-wise with the gradient matrix. The mask is only needed during training, no additional computations are needed at inference time.

IV. Experiments

We evaluated our approach on two of the most successful neural network architectures for image classification: fully connected and convolutional neural networks. All networks were implemented in Theano [13] [14]. Training was done on an Nvidia GTX980 GPU. We transferred the trained network to an Intel Edison mobile device for inference. The Intel Edison[1] was specially designed with IoT applications in mind. The Edison includes a 500 MHz Atom processor together with WiFi and Bluetooth connectivity in a package only slightly larger than a standard SD-card. Its size and typical power consumption of less than 1W make it even suitable for wearable applications. All timings reported are measured on the Edison. We processed one test sample at a time without using any batch processing optimizations since this resembles most the practical applications where information has to be processed as soon as it becomes available.

A. Fully connected networks

In the first experiment we used a fully connected neural network to classify images of handwritten digits. The MNIST [15] dataset contains a training set of 60,000 samples and a test set of 10,000 samples. Each sample is a 28 by 28 pixel grayscale image of a single handwritten digit.

[1]http://www.intel.com/content/www/us/en/do-it-yourself/edison.html

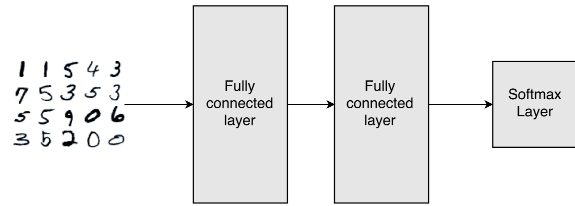

Fig. 2. The MNIST network has two fully connected layers.

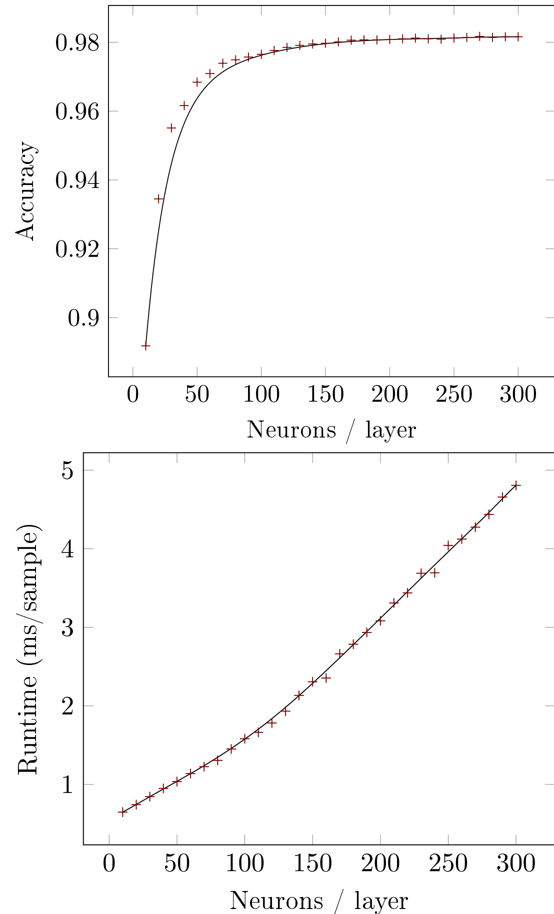

Fig. 3. The accuracy and runtime (ms/sample) of the fully connected MNIST network with an increasing number of neurons in each layer, evaluated on the Intel Edison.

We trained the architecture shown in Figure 2 on the MNIST images. The network has two fully connected layers, each with n neurons. The number of neurons was increased from 10 to 300 with increments of 10. This means that the final network contains the exact weights of 30 networks. All layers use the Rectified Linear activation function (ReLu) [16]. Dropout [17] with a probability of 0.25 was used during training.

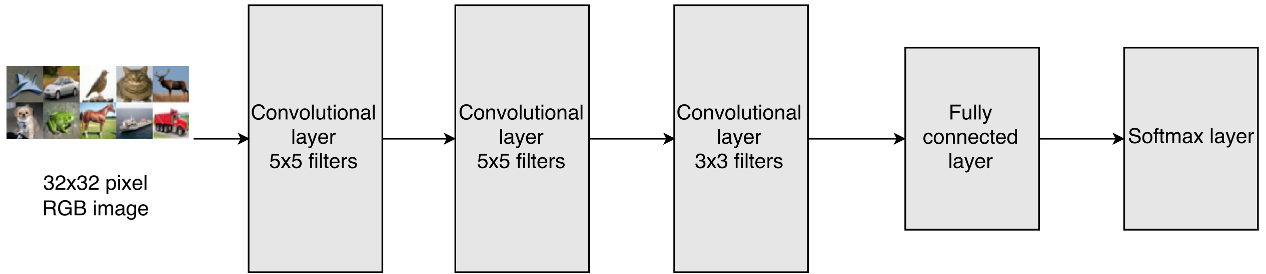

Fig. 4. The CIFAR10 network has three convolutional layers and one fully connected layer.

The smallest network, with two times ten neurons, is able to obtain an accuracy of 89%. The largest network with 40 times as many parameters achieves an accuracy of 98.16%. Constraining the network to contain all the parameters of the smaller networks as submatrices incurs a penalty on the final accuracy since this network has less flexibility to adjust its parameters. We retrained the largest network without any constraints on its parameters to find out how large this penalty actually is. We obtained an accuracy of 98.30%. A small cost to pay for having 30 networks contained in the same set of weights.

B. *Convolutional networks*

Convolutional neural networks [18] are more advanced feed-forward neural networks. These networks are able to exploit the 2D structure of images, unlike the fully-connected networks used in the previous section. Deep convolutional neural networks are currently the state-of-the-art tool for image classification [19].

Each convolutional layer applies trained filters to the input. These filters each respond to certain structures in the input data. The filters in the lower layers correspond to common smaller features such as borders and color transitions while the filters in the deeper layers respond to more complex composed structures such as a human face.

We trained the network shown in Figure 4 on the CIFAR10 [20] dataset. The CIFAR10 dataset is similar in size to the MNIST dataset but contains small color images of objects each corresponding to one out of ten classes such as cat, dog, deer, truck or ship. The network has three convolutional layers. The first two layers have 5x5 filters while the last convolutional layer has 3x3 filters. The network starts with four filters in each convolutional layer. We keep adding filters four at a time until each layer has 64 filters. The resulting network thus contains the weights of 16 networks. We also increase the size of the fully connected layer for each convolutional filter added. When the convolutional layers contain n filters each, the fully connected layer has $32n$ neurons. The Rectified Linear activation function is used both in the convolutional layers and in the fully connected layer.

Training these networks proved to be harder than training the fully connected version. Simply using random initialization yielded suboptimal results. Instead, we trained one large network without using any masking strategy. We then initialized the masked networks with the pretrained filters and fine-tuned them using backpropagation. Dropout [17] was essential to reliably train these networks

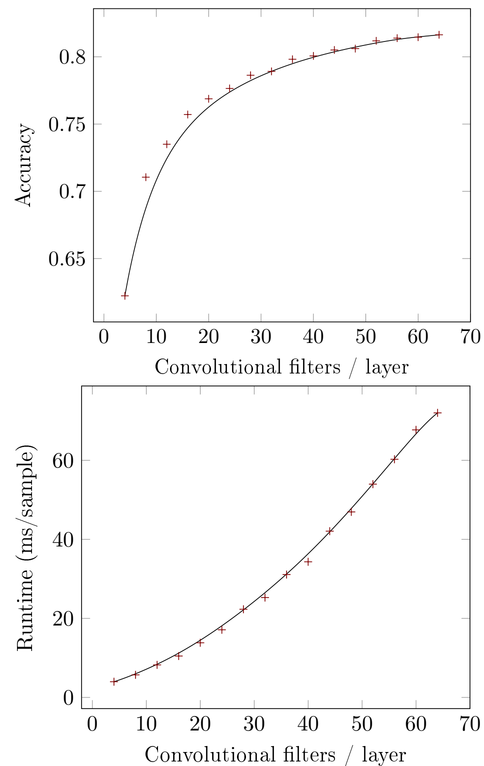

Fig. 5. The accuracy and runtime (ms/sample) of the convolutional CIFAR10 network with an increasing number of filters, evaluated on the Intel Edison.

The narrow network with only three times four convolutional filters and 128 neurons in the fully connected layer obtains an accuracy of 62.23%. The final network with 230 times as many parameters achieves an accuracy of 81.64%. An unconstrained network with the same architecture and similar training procedure obtains an accuracy of 83.3%. We again witness a small penalty in accuracy for the flexibility of having 16 networks all sharing a subset of the parameters.

V. Conclusion and future work

In this paper we presented a technique to incrementally train a series of neural networks, each increasing in size. To reduce the memory footprint of storing all the different networks, we forced each network to contain the exact weights of all the smaller networks as submatrices. A trade-off between accuracy and speed can be made at runtime by selecting the most appropriate network.

We plan to extend the work in this paper by combining it with the techniques presented in the related work section such as the compression technique proposed by Han et al [9]. We will also scale up the networks used to real world large images (ImageNet) and possibly to other application domains besides image classification. Another possible extension is to reduce the penalty in accuracy caused by constraining the weights through careful finetuning of the training procedure.

One practical application where we plan to use the Matryoshka networks is in real-time processing of video feeds. A smaller network can be used to process a certain frame if the previous frames all contained the same object. If the confidence of the network is small when processing a frame we can replace the network by a larger version. We expect this technique to achieve high accuracy while having a good amortized efficiency.

Acknowledgment

Part of this work was supported by the iMinds IoT Research Program. 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 a Tesla K40 GPU, a Jetson TK1 and TX1 used for this and similar research.

References

[1] D. Monroe, "Neuromorphic computing gets ready for the (really) big time," *Communications of the ACM*, vol. 57, no. 6, pp. 13–15, 2014.

[2] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, "Large-scale fpga-based convolutional networks," *Machine Learning on Very Large Data Sets*, vol. 1, 2011.

[3] Y. LeCun, J. S. Denker, S. A. Solla, R. E. Howard, and L. D. Jackel, "Optimal brain damage." in *NIPs*, vol. 89, 1989.

[4] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?" in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 2015, pp. 117–122.

[5] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," *arXiv preprint arXiv:1504.04788*, 2015.

[6] M. Courbariaux, Y. Bengio, and J. David, "Low precision storage for deep learning. arxiv preprint," *arXiv preprint arXiv:1412.7024*, 2014.

[7] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *arXiv preprint arXiv:1502.02551*, 2015.

[8] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1, 2011.

[9] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[10] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Advances in Neural Information Processing Systems*, 2014, pp. 2654–2662.

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

[12] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *arXiv preprint arXiv:1412.6550*, 2014.

[13] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral Presentation.

[14] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio, "Theano: new features and speed improvements," Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[15] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[16] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[17] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[20] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.