Fast Optimisation of Convolutional Neural Network Inference using System Performance Models

Rik Mulder
rik@rik.me
University of Edinburgh

Valentin Radu
valentin.radu@sheffield.ac.uk
University of Sheffield

Christophe Dubach
christophe.dubach@mcgill.ca
McGill University

Abstract

The choice of convolutional routines (or primitives) for implementing the operations in a Convolutional Neural Network (CNN) has a tremendous impact over the inference time. To optimise the execution latency for a target system, a lengthy profiling stage is needed – iterating over all the implementations of convolutional primitives in the configuration of each layer to measure their execution time on that platform. Each primitive exercises the system resources in different ways, so new profiling is currently needed when optimising for another system. In this work, we replace this prohibitively expensive profiling stage with a machine learning based approach of performance modelling. Our approach drastically speeds up the optimisation by estimating the latency of convolutional primitives in any layer configuration running on a target system. We reduce the time needed for optimising the execution of large neural networks on an ARM Cortex-A73 system from hours to just seconds. Our performance model is easily transferable across target platforms. This is demonstrated by training a performance model on an Intel platform and transferring its predictive performance to AMD and ARM systems, using very few profiled samples from the target platforms for fine-tuning the performance model.

CCS Concepts: • **General and reference → Performance**.

Keywords: neural network latency, inference optimisation

ACM Reference Format:
Rik Mulder, Valentin Radu, and Christophe Dubach. 2021. Fast Optimisation of Convolutional Neural Network Inference using System Performance Models. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3437984.3458840

EuroMLSys'21, April 26, 2021, Online, United Kingdom
https://doi.org/10.1145/3437984.3458840

1 Introduction

Deep learning has triggered a machine learning renascence. It is now powering numerous applications relying on tasks such as image classification, image segmentation, voice recognition, machine translation, activity recognition, and many others. Most of these applications use the cloud for their data processing, though we are seeing a drift from this paradigm over the last few years towards edge intelligence [19]. This is encouraged by the increasing number of computing capable devices around us (mobile phones, wearables, IoT devices) and growing privacy awareness.

Designing computationally efficient networks for edge intelligence is an active area of research [7, 10]. Producing specialised models by machine learning engineers to target a particular device is a futile approach due to the large number of various devices. Another approach is to take established CNN models and make their execution more edge-computing friendly. Quantization [6] and model pruning [16] are common techniques for reducing the execution complexity. However, these methods alter the structure of the original network, often lowering the estimation accuracy [6].

An alternative approach for improving the execution of a CNN is to select optimal primitives to use for each layer [1]. Convolutional primitives are alternative implementations of the convolution operation, producing equivalent data transformations in different ways. Their performance (execution time) varies due to their unique system resource utilisation (computations, memory access, etc.). Unlike model compression methods, with primitive selection the original estimation accuracy is maintained.

The set of available convolutional primitives includes the im2col variant – performing a data transformation and the layer operation through a matrix multiplication; the direct convolution variant as a deep nested loop without any additional memory allocation; the winograd variant, and others. Their implementation is also characterized by the data input format, the output format, and the employed system support (vectorization, etc.). These primitives have various performances based on the shape of the convolutional layer they implement, and no single primitive dominates the selection.

The main drawback of the primitive selection approach is that it requires a profiling stage to measure the execution time of each primitive in the network configuration [1]. However, this is excessively time consuming due to the many layer configurations in the CNNs, which makes applying

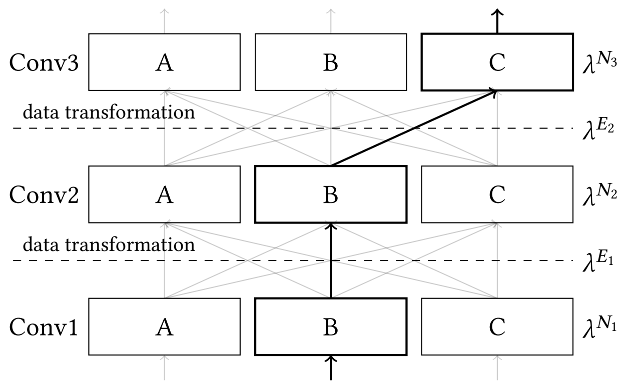


Figure 1. Primitive selection in a three-layer convolutional neural network, choosing from three available primitive implementations (A, B and C). Each node and edge have associated costs: $\lambda^N \in \mathbb{R}^3$ (the execution time of the primitive) and $\lambda^E \in \mathbb{R}^{3\times3}$ (the data layout transformation time) respectively. The primitive selection assigns a primitive to each layer such that it minimizes the total execution cost.

primitive selection difficult. It is unimaginable for application developers to optimise their networks for all the possible devices in the market. Any new network would also need profiling of its layer configurations on many target devices. We find that profiling about one hundred primitives with the layer configurations of ResNet-34 takes more than an hours on an ARM powered Odroid board.

Here, we eliminate the excessive profiling process and replace it with a performance model to estimate the execution time of primitives for any given layer shape. We show that our neural network based performance model is easily transferable to estimate the execution cost on other devices. For this, we adopt transfer learning of a trained model with just a small amount of profiled samples from the target device.

Specifically, we make the following contributions:

1. We design a performance model that accurately predicts the execution time of convolutional primitives for any configuration of convolutional layer.
2. Our experiments show that the performance model offers a substantial speedup in constructing the runtime configuration for a range of neural networks.
3. The performance model is easily transferable across platforms, which we demonstrate by training the model on a x86 system and transferring its predictive capacity to estimate for AMD and ARM systems.

2 Background and Motivating Example

This section presents the concept of primitive selection and introduces a motivating example to justify the need for performance models.

Table 1. Motivating Example. These are the times needed for optimising each CNN using: (i) the estimated execution time of primitives with our performance model (second column); and (ii) the profiled execution times on the device (third column). We do not consider here the training time for our performance model as that is an offline process.

CNN Model	Perf. Model Inf.	Profiling		
		Intel	AMD	Arm
AlexNet	43.6 ms	66 s	189 s	424 s
VGG-11	327 ms	0.20 h	0.67 h	1.72 h
VGG-19	673 ms	0.57 h	1.79 h	4.58 h
GoogLeNet	177 ms	182 s	445 s	0.37 h
ResNet-18	119 ms	242 s	736 s	0.43 h
ResNet-34	194 ms	494 s	0.41 h	0.89 h

2.1 Primitive Selection

By *primitive selection* in a neural network we denote the selection of the optimal primitive (computational routine) to use for implementing the convolution operation at each convolutional layer.

The execution time of each primitive is influenced by the layer configuration and the computing platform it runs on. Figure 1 shows this cost as λ^{N_i} of layer i, with each of the three primitives having their own cost ($\lambda_A^{N_i}$, $\lambda_B^{N_i}$ and $\lambda_B^{N_i}$). Data formats between two consecutive layers may not immediately match (e.g., frequency domain primitives structure their output data in a different format). So an additional computation cost is associated to preparing the data into the expected format for the primitive on the following layer in the network. This is represented as λ^E and added to the cost of using these two primitives.

The primitive assignment is made at layer level such that the entire execution time of the network is minimised. Many solvers can perform this selection. We use the Partitioned Boolean Quadratic Programming (PBQP) solver as it has been used successfully on this problem before [1].

2.2 Motivating Example

The primitive selection task has two stages. First, a cost is associated to each primitive and data format transformation. Second, these costs are used by a solver to determine the optimal primitive at each layer. With given execution costs, Boolean Quadratic Programming (PBQP) performs this selection in under a second even for large neural networks [1].

While the second stage is relatively fast and can be done offline, the first stage of associating a cost to each primitive is excessively time consuming. Currently, this is done by profiling each primitive in each layer configuration on the target device. For smaller devices this can take a long time.

Table 1 shows the profiling time of six deep neural networks on three computing platforms (third column). We can see that profiling all the candidate primitives for each layer configuration can span into the order of hours. By comparison, our performance model estimates the execution time of each primitive in only hundreds of ms (second column). These times also include the time of the PBQP optimiser.

It is clear that replacing the profiling stage with the estimation of a performance model saves substantial time in specialising the CNN to a target device. We show in our evaluation that these estimations are not just fast, but also highly accurate.

3 Related Work

Many approaches have been proposed for reducing the computational load of neural networks. These include lowering the floating point precision for inference [3], producing more compact models with neural architecture search [5, 8, 12, 17] and specializing the computations by employing compiler techniques [14, 18].

Previous works on primitive selection optimize the execution time of convolutional neural networks using empirical measurements of execution time [1] and heuristically selecting the search space with reinforcement learning [4].

Neural networks optimised with primitive selection have been shown to produce large speed-ups over the common frameworks, which use a single primitive across all the convolutional layers [1]. Such an optimisation has also been explored with reinforcement learning [4], which greatly reduces the search space due to heuristics about the cost of similar primitives and scaling costs with layer size. However, primitives may behave similar on one platform, but can have completely uncorrelated behaviour on another device or across layer sizes. There is no guarantee that the same heuristics hold across platform, so the additional cost of tuning the human-driven heuristics is still prohibitive.

Both works [1, 4] rely on a lengthy profiling stage, which can make the method impractical to scale. This work aims to remove the need for a profiling stage by introducing a performance model to predict the execution time of each primitive instead of measuring it on a given hardware.

Machine Learning for Performance Modelling. Previous work have shown that machine learning methods can predict the execution time of algorithms via performance models [9, 13]. Effort has also been invested in predicting the execution time of convolutional networks [2, 11]. Linear regressions [2] and fully connected neural networks [11] produce accurate predictions of execution time of a convolutional network. These are trained and evaluated for a single fix primitive and never considered for estimating the latency of a whole set of primitives.

Transfer learning have also been shown to significantly reduce the number of data points required for training a performance model on benchmark workloads [13]. Our task is harder than adapting to benchmark workloads because of the subtle performance differences across various primitives.

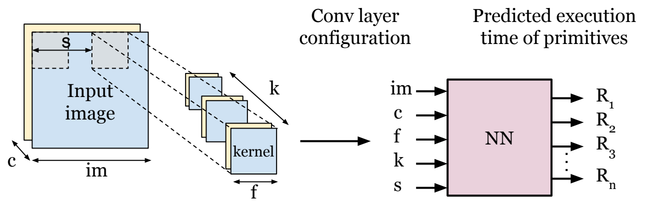


Figure 2. Our performance model is a neural network which takes as input the shape of the convolutional layer that needs to be optimised and outputs the estimated execution time of each primitive and data transformation.

4 CNN Performance Optimisation

We replaces the profiling stage of primitive selection from [1] with a performance model to estimate the execution time of primitives and data transformations.

Our performance model has a fully-connected neural network structure. Latency estimations produced by our performance model are passed to the PBQP solver to determine the final network configuration.

4.1 Convolutional Layer Optimisation

Figure 2 shows the estimation process with our performance model. This takes as input the shape of a convolutional layer: input image size im (height and width), its number of channels c; the kernel size f, stride s and number of kernels k. The performance model predicts the execution time of all the primitives in that layer configuration.

A significant number of highly efficient implementations are available for producing the convolutional operation (i.e., convolutional primitives), each with their drawbacks and benefits. We refer the reader to our previous report [15] for a broader description of the primitives we use in this work.

4.2 Performance Modelling

Performance Metrics. We assess the quality of our performance model estimations using the median relative absolute error (MdRAE):

$$\frac{|\hat{y} - y|}{y}$$

where $\hat{y}$ represents the prediction and y the measured execution time on the target system.

Performance Model Architectures. Given the complex performance of some primitives, a multi-layer fully-connected neural network is our best choice for a performance model. We train a performance model for each target computing platform. This takes as input the shape of convolutional layer and estimates the execution time of all the primitives in one output vector. We exploit batched inference to provide the

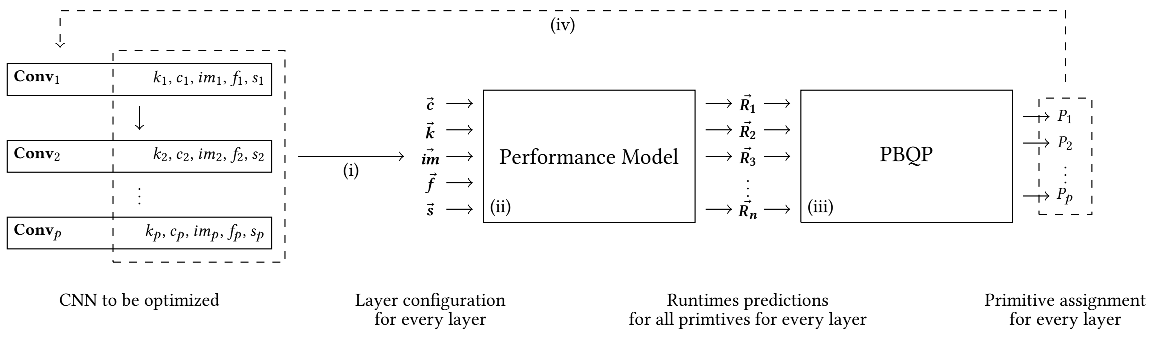

(iv)
Conv$_1$ k_1, c_1, im_1, f_1, s_1
Conv$_2$ k_2, c_2, im_2, f_2, s_2
Conv$_p$ k_p, c_p, im_p, f_p, s_p
(i)
$\vec{c}$ $\vec{k}$ $\vec{im}$ $\vec{f}$ $\vec{s}$
Performance Model
(ii)
$\vec{R}_1$ $\vec{R}_2$ $\vec{R}_3$ $\vec{R}_n$
PBQP
(iii)
P_1 P_2 P_p
CNN to be optimized
Layer configuration for every layer
Runtimes predictions for all primtives for every layer
Primitive assignment for every layer

Figure 3. The primitive selection process with performance modelling. We optimise the execution of a Convolutional Neural Network with p layers. Our performance model takes as input the shape of a convolutional layer and estimates the execution time of the n primitives and data layout transformations that can implement that layer. Each convolutional layer is characterised by its input number of channels c_i, input size im_i, number of kernels k_i, kernel size f_i and stride s_i. These execution time estimations are used by the PBQP optimiser to make the optimal primitive assignment (P_i) across the whole network. The performance model is batched – performing the computation for all layer configurations simultaneously in a single forward-pass of the performance model.

shape of all the convolutional layers of a CNN that needs to be optimised and produce a batch of estimations, one output for each layer of the CNN, in a single forward-pass through our performance model.

Loss Function. Latency prediction is a regression task. As such, we use the mean squared error (MSE) loss function. Some execution times may be undefined, for instance when a primitive cannot be applied to a particular shape of a convolutional layer. We make sure that undefined values have no effect on the training quality by masking out those values and their gradients in the forward pass and in the back-propagation stage respectively. By masking them, their squared error is zero, resulting in no influence over the training process.

Data Sample Normalization. Execution times can be substantially wide in magnitude. The MSE loss function is affected by extreme values. To address this issue, we transform the latency times to a log scale, so that it operates well in both large and small values. To improve the training, we normalise the input values to zero mean and a standard deviation of one.

Hyperparameter Search. Based on empirical exploration, we find the best hyperparameters for training our performance model to be the ones indicated in Table 2. These are used for generating the results presented in our evaluation section.

4.3 CNN Primitive Assignment

We adopt the primitive selection method described in Section 2.1, with the main difference being that instead of using measured times of the primitives, we use the performance estimation produced by our performance model. Figure 3 presents the steps in the optimisation process:

Table 2. The best hyperparameters we found for training the performance model. Early stopping terminates the training at about 250 epochs when no further improvement is observed for the estimation accuracy. In the fine-tuning stage of transfer learning, the learning rate is multiplied by 10^{-1}.

Hyperparameter	**Value**
Optimizer	Adam
Learning Rate	0.001
Weight Decay	1×10^{-5}
Batch Size	1024
Iterations	Early Stopping
Non-Linearity	ReLU
Architecture	$5 \times 128 \times 512 \times 512 \times 128 \times n$

1. For a given convolutional layer, we extract its configuration parameters.
2. The layer configuration parameters are passed as input to the performance model to estimate the execution time of all primitives and data-layout transformations.
3. The execution time predictions are passed to the PBQP solver to produce the primitive assignment across the entire network.

4.4 Profiling Data

We train our performance model from scratch with samples profiled from the target device using the most relevant configurations of convolutional layers. In this profiling stage we collect the tuples:

$$(k, c, im, s, f) \rightarrow (R_1, R_2, \ldots, R_N)$$

where the left side is the layer configuration (k, c, im, f, s) and the right side is the measured times of running N primitives that implement the layer configuration. Not all primitives

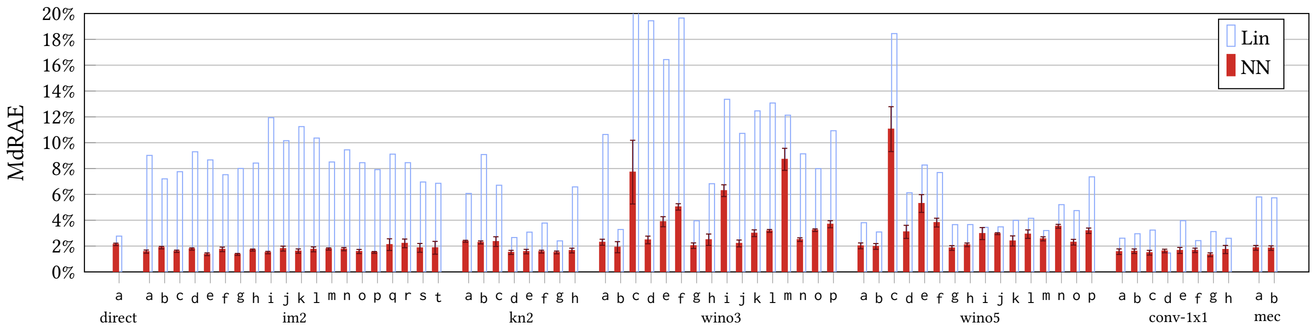


Figure 4. MdRAE (median relative absolute error) of estimated execution times using our NN performance model and a Linear Regression baseline on the Intel test samples. We aggregate the results on primitive class to observe the problematic primitives.

Table 3. The range of values taken by convolutional layer configuration parameters in the most popular CNNs.

Parameter	Meaning	Common Range
k	#kernels	1 to 2048
c	#channels	1 to 2048
im	image size	7 to 299
s	stride	1, 2 or 4
f	kernel size	1 to 11 (odd)

work for every configuration (e.g. a primitive may require a specific kernel size), hence some R_i can be undefined, as mentioned earlier.

We identify the values of convolutional layer configuration parameters in the most common CNNs, as presented in Table 3. For instance, we consider the input image size between 7 and 299 pixels, to match the shrinking image size that is propagated forward in the network.

The set of convolutional primitives is a broad list of known primitives, which were also used in previous work [1]. We take a similar approach for profiling data transformations.

5 Evaluation

This section validates our performance modelling approach by presenting the high accuracy of the performance model in estimating primitive latency and the effect of these estimations over the final solution produced by the PBQP solver. We also show that our performance model is easily transferable across computing platforms requiring minimal profiling.

5.1 Data Collection

We collect primitive execution times from three machines: Intel Core i9-9900K @ 5.0GHz; AMD A10-7850K @ 3.7GHz; and ARM Cortex-A73 (rev2) @ 2.36GHz. The profiled primitives are those used in previous work [1] (commit 8b0e642)[1]. For the ARM system, primitives are cross-compiled.

[1]https://bitbucket.org/STG-TCD/trinnity-benchmarks/

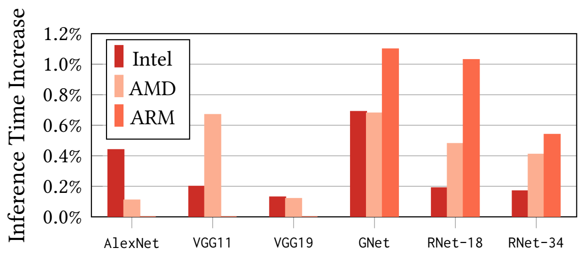


Figure 5. The relative increase in latency when the network is optimized with estimations from our performance model over the latency of optimising with the measured execution times. We see the increase is negligible for 6 deep CNNs.

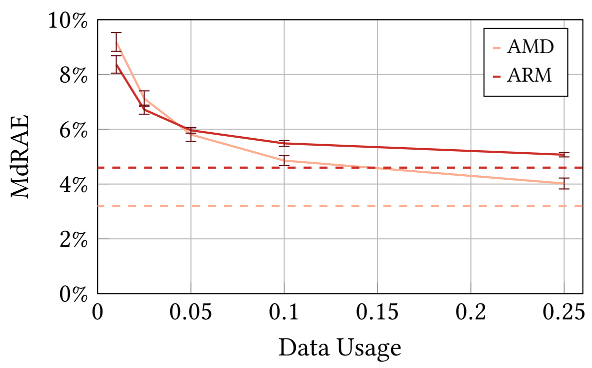
Each primitive is profiled 25 times for each layer configuration and we take the median value for reliability. We find that no single primitive is routinely the quickest, neither is one family of primitives dominant across all layer configurations.

5.2 Performance Modelling

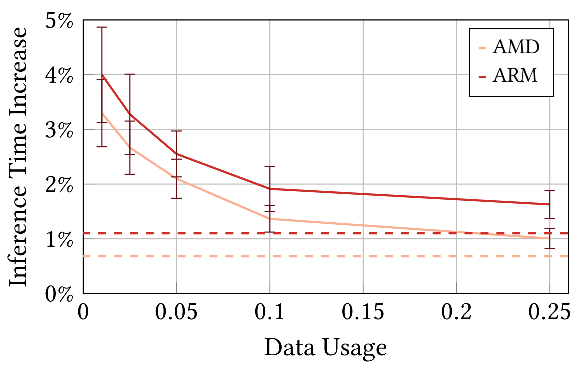
We evaluate a linear regression model (Lin) as baseline and our neural network (NN) performance model constructed as presented in Table 2.

We train the models with 80% of the profiled data from the Intel system. 10% of the data is used for fine-tuning the hyperparameters (validation set) and the remaining 10% for test. Figure 4 presents the performance model accuracy on the test set. We can see that the estimated latency of most primitives is below 2% absolute error. Our performance model outperforms the linear regression baseline by a big margin.

Similar prediction accuracy is observed for modelling the latency of primitives and data transformations on the other two platforms (ARM and AMD), which we present in a separate report [15].



(a) Prediction accuracy of performance model estimations using TL fine-tuned with a varying fraction of the profiled samples.



(b) Primitive selection quality (after applying the PBQP solver) using the estimated execution times.

Figure 6. The predictive and primitive selection performance of the ARM and AMD performance models, fine-tuned using a varying amounts of training data (shown on the x-axis). The pre-trained Intel performance model is transferred and fine-tuned with data from the other two platforms (AMD and ARM). The dotted lines shows the performance of the ARM and AMD performance models when trained from scratch using all the profiled data from these platforms.

5.3 Impact on Primitive Selection

Estimating the latency of primitives with an error of between 2% and 10% may change the ranking of best candidate primitives. Our next experiment exposes the impact of using estimated execution times in the PBQP solver.

Figure 5 shows the increase in execution time of the entire neural networks that is optimised first with the measured execution times, and then with the estimated latency by our model. Using the estimated latency causes a slight increase in the large CNN execution time due to the marginal-opium ranking of primitives. We find that for most cases the increase is negligible, under 1%, compared to direct latency profiling.

5.4 Transferability

It is expensive to train performance models from scratch for a new platform due to the lengthy profiling of a good size training set. Instead, we propose to use a trained performance model and specialize it for other platforms with minimal amount of profiled data points.

We evaluate the transferability of a trained performance model from an Intel machine to other systems (ARM and AMD) by fine-tuning the model with a few profiled samples from the target machines. This is tested on the layers of GoogLeNet due to its large variety in convolutional layers.

Previous work has shown that transfer learning aids the burden of collecting a large training set from the target devices [13]. Our initial performance model is trained on 90k samples profiled on the Intel system. We want to know how much data from another system is actually enough for fine-tuning this pretrained performance model to reach good accuracy. To answer this, we perform 6 separate fine-tuning tests using different fractions of the initial training sets. Samples are randomly selected at 0.1%, 1%, 2.5%, 5%, 10% and 25% fractions of the available training data from the AMD and ARM platforms (90k, 40k samples respectively). The fine-tuning is repeated 25 times using a random subsets of the available data.

Figure 6 shows the accuracy of these fine-tuned performance models, both for latency estimation accuracy and for their effects in the PBQP solver. We see that just 10% of the available training data is sufficient for an accuracy no lower than 2% compared to that obtained when training from scratch using 100% of the available data. A further decreasing below 5% of the training data starts introducing substantial error, but this is still good enough if training data is very hard to obtain, as a trade-off for accuracy.

We present further results in a separate report [15] and make our code and data available[2].

6 Conclusions

We introduce performance modelling in convolutional primitive selection for accelerating the execution of CNNs. Assessing the latency of each primitive that can implement a convolutional layer is essential in primitive selection. Instead of profiling all the layer configurations, our performance model estimates the execution time all the primitives in the given layer configuration. Execution latency estimation reduces the task of primitive selection from hours to just seconds. Our performance model estimates the execution time for any convolutional layer configuration, which makes it ideal for a dynamic environment with many custom neural networks, such as mobile phones running various intelligent applications. The execution time of entire CNNs is off by only 0.39% on average (worse case 1.1%) from the optimal execution time. Our performance model is also viable for transfer learning.

[2]https://github.com/Ubiquitous-AI-Lab/CNN-performance-modelling

This requires only a small amount of profiled execution time samples from a target platform.

Future Work. We will explore strategies for reducing the amount of profiled samples needed in transfer learning from a new target device by determining which candidate primitives and layer configurations are the most informative about the computing system characteristics.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], the European Unions Horizon 2020 research and innovation programme [grant No. 732204 (Bonseyes)], and the Canada CIFAR AI Chairs Program.

References

[1] Andrew Anderson and David Gregg. 2018. Optimal DNN Primitive Selection with Partitioned Boolean Quadratic Programming. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 340–351.

[2] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. NeuralPower: Predict and Deploy Energy-Efficient Convolutional Neural Networks. In *Proceedings of Machine Learning Research*. 622–637.

[3] Philip Colangelo, Nasibeh Nasiri, Eriko Nurvitadhi, Asit Mishra, Martin Margala, and Kevin Nealis. 2018. Exploration of Low Numeric Precision Deep Learning Inference Using Intel FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[4] Miguel de Prado, Nuria Pazos, and Luca Benini. 2019. Learning to infer: RL-based search for DNN primitive selection on Heterogeneous Embedded Systems. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1409–1414.

[5] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. 2018. PPP-Net: Platform-aware Progressive Search for Pareto-optimal Neural Architectures. In *ICLR Workshop*.

[6] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[7] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv* 1704.04861 (2017).

[8] Chi-Hung Hsu, Shu-Huan Chang, Da-Cheng Juan, Jia-Yu Pan, Yu-Ting Chen, Wenli Wei, and Shih-Chieh Chang. 2018. MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning. *ArXiv* 1806.10332 (2018).

[9] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* 206 (2014), 79–111.

[10] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2017. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *ArXiv* 1602.07360 (2017).

[11] Daniel Justus, John Brennan, Stephen Bonner, and A. Stephen McGough. 2018. Predicting the Computational Cost of Deep Learning Models. *2018 IEEE International Conference on Big Data (Big Data)* (2018), 3873–3882.

[12] Ye-Hoon Kim, Bhargava Reddy, Sojung Yun, and Chanwon Seo. 2017. NEMO : Neuro-Evolution with Multiobjective Optimization of Deep Neural Network for Speed and Accuracy. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 419–427.

[13] Preeti Malakar, Prasanna Balaprakash, Venkatram Vishwanath, Vitali Morozov, and Kalyan Kumaran. 2018. Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 33–44.

[14] Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael O'Boyle, and Christophe Dubach. 2020. Automatic generation of specialized direct convolutions for mobile GPUs. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 41–50.

[15] Rik Mulder, Valentin Radu, and Christophe Dubach. 2020. Optimising the Performance of Convolutional Neural Networks across Computing Systems using Transfer Learning. *arXiv preprint arXiv:2010.10621* (2020).

[16] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, José Cano, Elliot J Crowley, Björn Franke, Amos Storkey, and Michael O'Boyle. 2019. Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs. (2019).

[17] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *CVPR*.

[18] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. *SIGPLAN Not.* 51 (2016), 209–223.

[19] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.