IPSJ SIG Technical Report

Vol.2022-ARC-248 No.13
Vol.2022-SLDM-198 No.13
Vol.2022-EMB-59 No.13
2022/3/10

Layer-wise power/performance modelling for single-board CNN inference.

Kuan Yi Ng[1,a)] Aalaa M.A. Babai[1,b)] Satoshi Kawakami[1,c)] Teruo Tanimoto[1,d)] Koji Inoue[1,e)]

Abstract: Intermittent executions and energy harvesting technologies are promising candidates to enable renewable energy on small-scale computer systems like single-board computers, making sustainable computing possible. In this work, we implemented an energy consumption prediction framework for each layer of CNN executing on single-board computers based on NeuralPower as the first step towards enabling energy-efficient intermittent execution of CNN inference on single-board computers. We found that layer hyperparameters cannot explain all the variations in execution time and power consumption when the layer is executed. Model's prediction can be improved with the knowledge of performance counter values, but these values are not available before a layer is executed. Furthermore, our analysis revealed that implementation optimization like sparse matrix multiplication might cause a layer's execution time and power to change with its input values.

1. Introduction

As the environmental impact of computing grows in concern, energy harvesting-based small-scale intermittent computing systems are gaining interest. Since such systems tend to follow the checkpoint-based execution model [10], [15], minimizing checkpoint overhead—which leads to longer execution time and higher energy consumption—is a critical challenge. A promising solution is to skip a particular checkpoint if there is enough energy to execute the following program partition, including the next checkpointing (i.e., the device can decide whether to perform particular checkpoints) based on energy predictions.

Reframing this approach to fit our interest to enable efficient convolutional neural network (CNN) inference on a single-board computer, we propose a layer-wise selective checkpointing scheme with power/performance models. As the first step for this purpose, in this work, we implement a power/performance (or energy) prediction framework based on NeuralPower [4] and evaluate its accuracy. Precisely, it consists of an execution time model and a power consumption model. The following assumptions are applied to simplify the situation of intermittent computing. First, we target Raspberry Pi 4 as a single-board computer and assume it can be turned into an energy harvesting device by switching its power supply to an energy harvesting component and an energy buffer. Second, correct execution of CNN inference can be achieved even when CNN is executed layer by layer. Third, Raspberry Pi 4, with energy harvesting capabilities, has sufficient energy to execute at least one layer of CNN when its energy buffer is full.

The main difference between NeuralPower and our work is the platforms that execute the CNNs; while NeuralPower targets GPUs, single-board computers are our target platform. Also, in addition to implementing and evaluating the prediction framework, supplementary analyses are performed to identify the causes of prediction errors and the possible ways to improve the prediction. Our findings are as follows:

(1) We show that it is possible to estimate the execution time and power consumption of each layer of a CNN with its hyperparameters.

(2) However, our evaluation also shows that prediction with only layer hyperparameters is insufficient in use-cases where high precision of estimations is required.

(3) We found that knowledge of performance counters during prediction can help reduce prediction error but is unavailable before executions.

(4) Our analysis revealed that implementation optimization that exploits the properties of input data and trained weight could affect the execution time and power consumption.

The rest of the paper is organized as follows. Section 2 introduces related work. An overview of the methodology of execution time and power modeling is given in Section 3.

[1] Kyushu Uniersity, Fukuoka, Fukuoka 819–0395, Japan
[a)] kuanyi.ng@cpc.ait.kyushu-u.ac.jp
[b)] aalaa.babai@cpc.ait.kyushu-u.ac.jp
[c)] satoshi.kawakami@cpc.ait.kyushu-u.ac.jp
[d)] tteruo@kyudai.jp
[e)] inoue@ait.kyushu-u.ac.jp

Section 4 describes the experimental setup used, while Section 5 presents the evaluation results. Section 6 discusses the experimental results and future works. Finally, Section 7 concludes this paper.

2. Related Work

Both [9] and [1] presented a power model for Raspberry Pi 2 Model B based on linear regression that estimated the power consumption of Raspberry Pi. In both works, the power model translates CPU utilization and network utilization into Raspberry Pi 2's power consumption. Tools like *stress* command, *cpulimit*, and *iperf* were used to measure the power consumption at different CPU and network utilization. However, we found out that different CNN layers consumed different amounts of power despite each execution having CPU utilization of near 100 %. This observation is likely due to the difference in how *stress* command and CNN's layer utilize the computing and memory components of the single-board computer. Also, we found that *stress* command consumed more power when executed with performance governor mode than when executed with *powersave* governor mode on Raspberry Pi 4. As a reference, the *performance* governor mode tries to keep the CPU cores running at 1.5 GHz while the *powersave* governor mode tries to keep the operating frequency at 600 MHz. Based on these findings, the power consumption of newer single board computers might be more complicated when we take different power modes (i.e., governor mode) into account.

Precious [12] presented an approach of a system that estimates execution time and power draw of convolutional and fully-connected neural networks that execute on Google Coral Edge TPU. We refer to how they generate neural networks to collect training data for execution time and power models. Section 3 explains the details about neural networks generation. There are two main differences between Precious and our work: estimation granularity and the hardware executing the CNNs. From the perspective of estimation granularity, Precious estimated the execution time and power consumption for the whole neural network, while our models estimate each layer of a CNN. In addition, in Precious, the neural networks are executed on Google Coral Edge TPU (an accelerator), while we used Raspberry Pi 4 to execute the neural networks. NeuralPower [4] suggested a methodology to create an execution time and power consumption model for CNN inferences on GPU platforms. Unlike Precious, the estimation models in NeuralPower can predict the execution time and power consumption of each layer of a CNN. The prediction for a whole neural network is calculated from the predicted values of all the consisting layers. We refer to how they use a lasso polynomial regression model to relate layer hyperparameters to their execution time and power.

[5] proposed CleanCut, a tool that assists programmers in splitting their programs into smaller tasks, such that forward execution progress of the program is ensured when executed on energy harvesting devices. CleanCut showed the importance of knowing the energy consumption of program parts beforehand when executing programs on energy harvesting devices. Other works such as Layerweaver [11] and Neurosurgeon [8] showed how knowledge of resource utilization (i.e., execution time and power consumption) allows efficient execution of CNN. Layerweaver used information about CNN's layers to increase the utilization of computing units in accelerators. Neurosurgeon utilizes information of CNN's layers to optimize distributed executions of CNN between mobile devices and cloud data centers for low latency or low energy consumption. Distributed computation of CNN in Neurosurgeon also supports our assumption that CNN can be executed correctly even when executed layers by layers.

Table 1: The hyperparameters of each layer that are used as the input features for the execution time and power model.

	Conv.	**Pool.**	**Flat.**	**FC**
input size (2D/1D)	✓	✓	✓	✓
# channel	✓	✓	✓	
# filters/neurons	✓			✓
kernel/pool size	✓	✓		
strides	✓	✓		
padding	✓	✓		

3. Methodology: Layer-level Execution Time and Power Modeling

Our work chose to model the execution time and power consumption of the convolutional layer (Conv.), max-pooling layer (Pool.), flatten layer (Flat.), and fully connected layer (FC). These layers construct CNN like LeNet, AlexNet, VGG16 and VGG19. Although newer CNN like ResNet and MobileNet, have more complicated building blocks that consist of multiple layers connected in a non-sequential way. For instance, ResNet contains multiple residual units, where each residual unit is a small neural network with a skip connection. The skip connection connects the input of the residual unit directly to its output without passing through the intermediate layers. This structure of residual unit goes against our assumption that CNN can be executed intermittently one layer at a time. So, we decided not to include these CNNs in our targets for modeling.

We created an execution time model and a power consumption model for each layer type, which takes the layer's hyperparameters as inputs. The use of layer hyperparameters as the input allows us to estimate before executing a particular layer. Table 1 lists the hyperparameters that we use as the input features. Our experiments assume that the 2-dimensional hyperparameters, like *input size*, *kernel size*, *pool size*, and *strides*, are squares. For example, LeNet, AlexNet, VGG16, and VGG19 use a square for their 2-dimensional hyperparameters.

We used a polynomial regression model to relate the layer's hyperparameters with execution time and power consumption. A polynomial regression model contains interaction terms formed by the multiplication of layer hyperparameters. These interaction terms can help explain the nonlinear relationship between layer hyperparameters and the model's output (i.e., execution time and power consumption) [4].

The execution time $\hat{T}$ of a layer can be expressed as:

$$\hat{T}(\mathbf{x}) = \sum_j c_j \cdot \prod_{i=1}^{D_T} x_i^{q_{ij}}$$

$$\text{where } \mathbf{x} \in \mathbb{R}^{D_T}; q_{ij} \in \mathbb{N}; \forall j \sum_{i=i}^{D_T} q_{ij} \leq K_T.$$

The model is a K_T-th degree polynomial of $\mathbf{x}$, where each element in $\mathbf{x}$ is the value of a hyperparameter. Different types of layers have different numbers of hyperparameters, thus different value of D_T. q_{ij} is the exponent for the hyperparameter, x_i in the j-th polynomial term. The j-th polynomial term $\prod_{i=1}^{D_T} x_i^{q_{ij}}$ represents an interaction term between different hyperparameters x_i. c_j is the coefficient to learn from training data.

The power consumption $\hat{P}$ of a layer can be expressed in a similar way:

$$\hat{P}(\mathbf{x}) = \sum_j z_j \cdot \prod_{i=1}^{D_P} x_i^{m_{ij}}$$

$$\text{where } \mathbf{x} \in \mathbb{R}^{D_P}; m_{ij} \in \mathbb{N}; \forall j \sum_{i=i}^{D_P} m_{ij} \leq K_P.$$

The model is a K_P-th degree polynomial of $\mathbf{x}$, where each element in $\mathbf{x}$ is the value of a hyperparameter. m_{ij} is the exponent for the hyperparameter, x_i in the j-th polynomial term. The j-th polynomial term $\prod_{i=1}^{D_P} x_i^{m_{ij}}$ represents an interaction term between different hyperparameters x_i. z_j is the coefficient to learn from training data.

3.1 Creating the Models

This subsection introduces the procedure to create the execution time model and power model for each type of layer. The same procedure is taken to model execution time and power consumption for all four types of layers. First, 1-layered neural networks are generated, executed, and measured (Section 3.1.1). Then, polynomial regression models of different degrees are used to fit the training data, and we selected the model with the lowest root-mean-square error (RMSE) (Section 3.1.2). As a result, a total of 8 polynomial regression models, one execution time model, and one power model for each layer type (e.g., convolutional, max-pooling, flatten, fully-connected layer) were created.

3.1.1 Data Collection

To create a regression model that can learn how different combinations of layer hyperparameters relate to the layer's execution time or power consumption, we decided to use the measurements of different 1-layered neural networks as the training data of the regression model. We define 1- a layered neural network as a neural network that contains one intermediate layer (input and output are not counted as a layer). A 1-layered neural network's output is obtained after the only intermediate layer performs operations on the neural network's input. We randomly generated a set of 1-layered neural networks for layer type based on the hyperparameters value ranges listed in Table 2. For instance, if we want to generate a 1-layered neural network of type flatten randomly, we randomly choose an integer from 16 to 256 as the input size and another integer from 64 to 512 as the channel hyperparameter. The 1-layered neural networks generated were then executed on Raspberry Pi 4 and measured to obtain their average execution time—for one inference—and average power consumption. Details on how each measurement is performed can be found in Section 4.3.

Table 2: The values range of hyperparameters of each layer used in 1-layered neural network generation

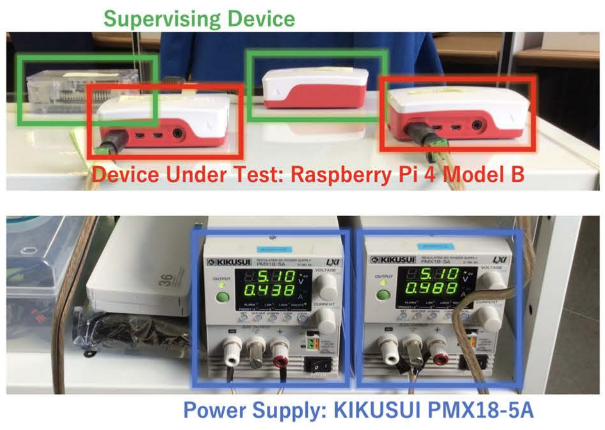
	Conv.	**Pool.**	**Flat.**	**FC**
input size	$[16, 256]$	$[16, 256]$	$[16, 256]$	$[256, 51200]$
# channel	$[1, 512]$	$[64, 512]$	$[64, 512]$	
# filters	$[64, 512]$			$[16, 4096]$
kernel size	$[2, 5]$	$[2, 4]$		
strides	$[1, 5]$	$[1, 3]$		
padding	(yes, no)	(yes, no)		

In our experiments, 180 1-layered neural networks were generated for each layer type (a total of 720 1-layered neural networks). The numbers of 1-layered neural networks can be changed to target more layer types or a wider range of layer hyperparameters.

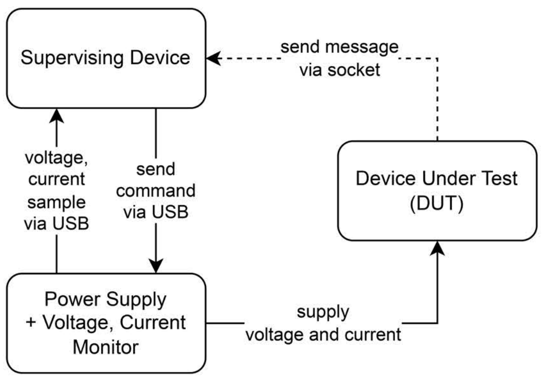
3.1.2 Model Training and Model Selection

A polynomial regression model is used to model each layer type's execution time and power consumption. In addition, we also regularized the polynomial regression model with Lasso regression to perform automatic feature selection and obtain a sparse regression model [7]. The Lasso regression sets the coefficient of least important features to zero during the model's training process.

While the polynomial regression model has degree K as a parameter that decides how many additional polynomial terms to include, Lasso regression has alpha α as a parameter that decides its tendency to set a coefficient to zero. To select suitable values for K and α, we used 5-fold cross-validation to compare regression models with different values of K and α. The values of K and α that we considered were $\{1, 2, 3, 4\}$ and $\{1 \times 10^{-10}, 1 \times 10^{-9}, 1 \times 10^{-8}, \ldots, 1\}$ respectively. In detail, a 5-fold cross-validation randomly splits the training data set into five distinct smaller training sets called folds. It trains and evaluates the regression model 5 times, picking a different fold for evaluation after training on the remaining four-folds each time. As a result, a regression model with a specific K, α values will have five values of RMSE, and the final score of the regression model is equal to the mean of these five RMSEs. Then, we selected a combination of K, α values, such that the regression model with these parameter values has the lowest final score (i.e., mean RMSE from 5-fold cross-validation). Note that, because layer's hyperparameters relate differently to execution time and power consumption, the K, α values for the execution time model and power model of a layer type will be different. Finally, the regression model is retrained


Supervising Device
Device Under Test: Raspberry Pi 4 Model B
Power Supply: KIKUSUI PMX18-5A

(a) Hardwares used in the experiment


Supervising Device
send message via socket
voltage, current sample via USB
send command via USB
Device Under Test (DUT)
Power Supply + Voltage, Current Monitor
supply voltage and current

(b) Devices connection and communication

Fig. 1: Experiment setup

Table 3: Softwares used in the experiments

Role	Software
Raspberry Pi 4's OS	Ubuntu Server 20.04.3 LTS (64-bit)
Deep learning framework (inference)	TensorFlow Lite Runtime + Python 3.8.10
Deep learning framework (network generation)	Keras 2.7 + TensorFlow 2.7 + Python 3.8.10
Sampling program	Python 3.8.10 + Third-party libraries
Analysis and modeling	Python 3.8.10 + Third-party libraries

using all the training data with the K, α values selected with cross-validation.

4. Experiment Setup

This subsection outlines information about the experimental setup, including hardware, software, and experiment techniques.

4.1 Hardware

Figure 1a shows the hardware that is used in the experiments and the setup where all hardware listed is visible, and Figure 1b shows how devices connect and communicate with each other. The DC power supply (PMX18-5A) [6] outputs 5.1V and 3A to the Device Under Test (DUT). It can also monitor the amount of voltage and current supplied to the DUT (i.e., the single-board computer as a whole). The instantaneous power consumption of DUT is later calculated using the equation $P = V \times I$.

The supervising device is connected to the DC power supply via USB, enabling communication. This setup allows us to run a program on the supervising device that performs voltage and current sampling by sending a command to the power supply at regular intervals. Even though we use Raspberry Pi as the supervising device in our setup, any device that can execute the sampling program works fine. The software requirements for the supervising device is explained in section 4.2.

The DUT is the device that performs CNN inferences and is measured. The specification of the DUT used in our experiment is listed in Table 4.

Table 4: The specifications of the DUT

	Raspberry Pi 4 Model B
CPU	Quad core Cortex-A72 (ARM v8)
Main memory	8GB LPDDR4-3200 SDRAM
Secondary memory	64GB Micro-SD card

4.2 Software

Table 3 shows the software used in the experiments. TensorFlow Lite (TFLite) Runtime [13] is used to execute neural networks inferences on a single thread on the Raspberry Pi 4. The program for executing neural network inferences is written in Python 3 (3.8.10). Note that TFLite runtime can only execute TFLite models. Hence, during neural networks creation, we converted the Keras model (neural network generated with Keras API) to a TFLite model before running it on Raspberry Pi 4. When a Keras model is converted to a TFLite model, optimization is performed on the neural network to reduce latency and memory usage with minimal loss in inference accuracy. There are several optimization techniques available during the conversion to a TFLite model [14]. We chose the default optimization technique, which quantized the weights of a neural network from floating points to 8-bit integers. These quantized weights are converted back to floating-point at inference before performing computations.

Devices with Python 3 (3.8.10 or higher) and USB interface can execute the sampling program. Figure 2 illustrates how the sampling program works. Socket communications between the supervising device and DUT are performed so that only the executions of the layer are measured. DUT will send a message to the supervising device right before and after measurement to ensure the measurement session is started and stopped at the right time. As a result, executions unrelated to the layer's inference, for example, the loading of the layer into memory and the preparation of input data before an inference, are not measured.

We used Python third-party libraries to perform data analysis and modeling execution time and power consump-

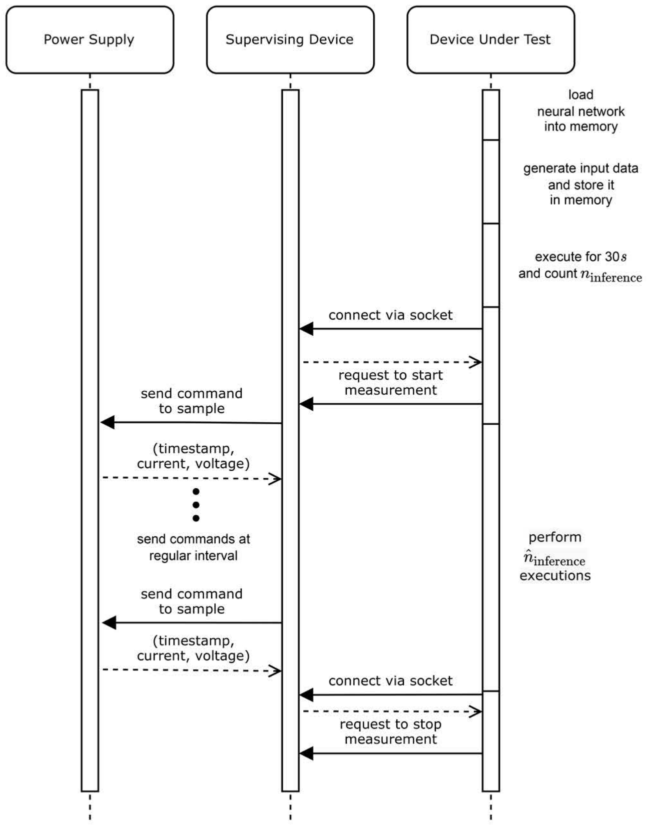

Power Supply
Supervising Device
Device Under Test
load neural network into memory
generate input data and store it in memory
execute for 30s and count $n_{\text{inference}}$
connect via socket
request to start measurement
send command to sample
(timestamp, current, voltage)
send commands at regular interval
perform $\hat{n}_{\text{inference}}$ executions
send command to sample
(timestamp, current, voltage)
connect via socket
request to stop measurement

Fig. 2: A sequence diagram showing how sampling program only measure voltage and current during layer's executions.

tion. For data manipulation and processing, we use *NumPy* and *pandas* libraries. For data visualization, we use *matplotlib* and *seaborn* libraries. Finally, for training and evaluation of the regression model, we use *scikit-learn* library.

4.3 Measurement Techniques

4.3.1 Sampling frequency

The DC power supply we used monitors the voltage and current once every 25 ms, but we sampled once every 50 ms. When we tried to sample once every 25 ms, there were times when the measurement program failed to sample voltage and current.

4.3.2 Number of inferences in a single measurement session

To create an execution time model and power model of one execution for each layer type, we first need to measure each layer's execution time and power consumption. However, the execution time of one layer is shorter than the sampling period of 50 ms. So, we execute the same layer with the same input data multiple times in a measurement session. The average execution time of one execution is then calculated by dividing the total execution time by the number of inferences.

Precisely, for each layer, we first count the number of inferences $n_{\text{inference}}$ executed in 30 s without measuring. Then, the same layer is executed for $\hat{n}_{\text{inference}}$ times while its voltage and current are measured. $\hat{n}_{\text{inference}}$ is a round-up value of $n_{\text{inference}}$ calculated using the equation below. Finally,

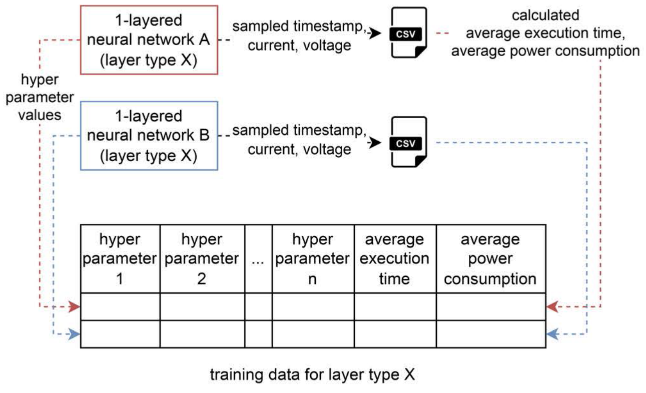

1-layered neural network A (layer type X)
sampled timestamp, current, voltage
CSV
calculated average execution time, average power consumption
hyper parameter values
1-layered neural network B (layer type X)
sampled timestamp, current, voltage
CSV
hyper parameter 1
hyper parameter 2
...
hyper parameter n
average execution time
average power consumption
training data for layer type X

Fig. 3: Hyperparameters of 1-layered neural networks (type X) and measurement data form training data for layer type X

rounding up of $n_{\text{inference}}$ is performed to ensure the executions of each layer are measured for at least 30 s.

$$\hat{n}_{\text{inference}} = (\text{first digit of } n_{\text{inference}} + 1) \\ * 10^{\text{number of digit of } n_{\text{inference}} - 1}$$

4.3.3 Reducing measurement noise

To exclude the impact of dynamic frequency scaling on the execution time and power consumptions measurements, we set the operating frequency of Raspberry Pi 4 at 1.5GHz by setting the governer mode to *performance mode* using the *cpufrequtils* command. Although the performance governer mode will try to keep the operating frequency at 1.5 GHz, it will reduce when the SoC gets too hot. As a result, there exist short durations where Raspberry Pi is operating under 1.5 GHz during the measurements.

4.3.4 Data Preparation

Figure 3 shows how we turn the measurement data into a layer type's training data. After the measurement of one 1-layered neural network, a CSV file containing a series of (1) timestamps where each sample is taken, (2) current measurements, and (3) voltage measurements were obtained. Using the equation below, we calculated the average execution time $\bar{T}_{\text{inference}}$ and average power consumption $\bar{P}_{\text{inference}}$. T_{measured} is the total duration measured, and $n_{\text{inference}}$ is the number of inferences performed in T_{measured} seconds. V_{t_i} and I_{t_i} are the voltage and current measured at time t_i, and T_{sampling} is the duration between 2 consecutive samples which is 50 ms in our case. The average execution time, average power consumption of a 1-layered neural network and its hyperparameters are treated as a row of data in the training set for that layer type.

$$\bar{T}_{\text{inference}} = \frac{T_{\text{measured}}}{n_{\text{inference}}}$$

$$\bar{P}_{\text{inference}} = \frac{1}{T_{\text{measured}}} \sum_{i=2}^{n_{\text{sample}}} T_{\text{sampling}} \frac{P_{t_i-1} + P_{t_i}}{2}$$

$$\text{where } 1 \leq i \leq n_{\text{sample}}; P_{t_i} = V_{t_i} \times I_{t_i}$$

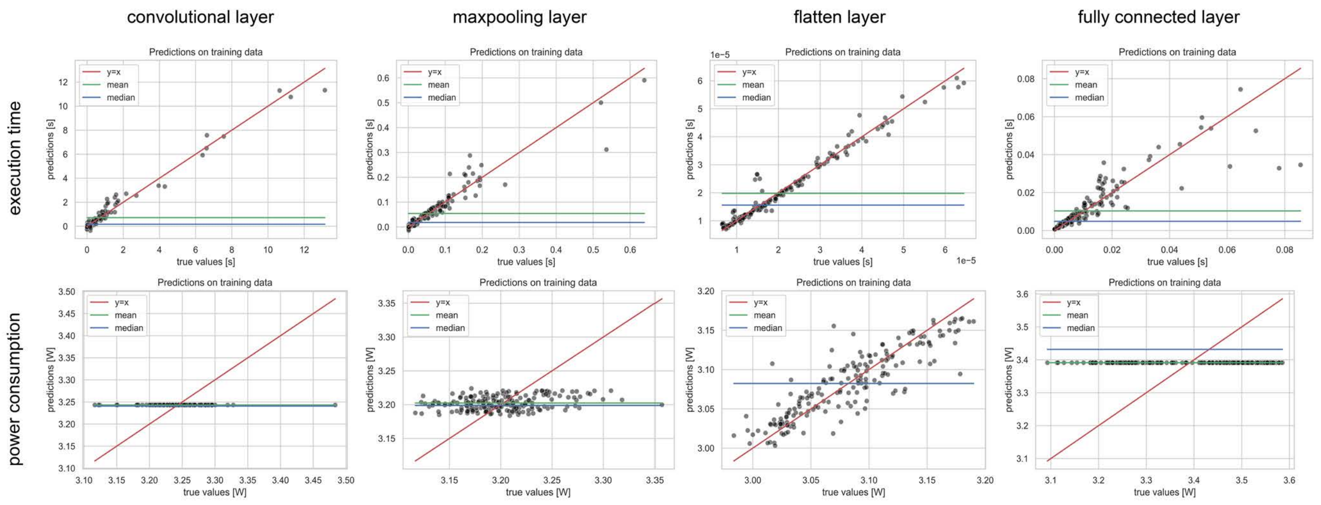


Fig. 4: A graph showing the scatterplots on prediction values and truth values.

5. Evaluation

This section explains how we evaluate our execution and power consumption models and the evaluation results. First, Section 5.1 explains the evaluation metrics used, followed by evaluation results in Section 5.2.

5.1 Evaluation Metrics

Our evaluation used the following metrics to measure how much error is obtained during prediction.

- root mean square error (RMSE)
- root mean square percentage error (RMSPE)
- mean absolute error (MAE)
- mean absolute percentage error (MAPE)

RMSE, RMSPE, MAE, and MAPE are four different error metrics used to measure the difference between the truth values and the regression models' prediction values.

In the equations above, $\mathbf{y}$ is the truth values while $\hat{\mathbf{y}}$ is the predicted values. N is the number of truth values (or predicted values), while y_i and $\hat{y_i}$ represent each pair of truth values and predicted values, respectively.

These error metrics can be categorized in two different ways, (1) absolute value or relative value, and (2) equal or non-equal weights placed on each error (prediction error). Both RMSE and MAE are absolute values, while RMSPE and MAPE are relative values. In other words, RMSE and MAE will have the same unit as the output properties (i.e., s for execution time and W for power consumption). On the other hand, RMSPE and MAPE have values of between 0 to 1, representing values between 0% and 100%. Another difference between these metrics is the weights they placed on each error. From the equation above, we can see that all errors $(y_i - \hat{y_i})$ in MAE and MAPE are equally weighted, but in RMSE and RMSPE, the weight of each error depends on their size (i.e., a larger error will contribute more because they are squared).

Table 5: Prediction errors trained only with layer hyperparameters

(a) Execution time

Layer	**Conv.**	**Pool.**	**Flat.**	**FC**
degree	4	4	3	2
α	1×10^{-3}	1×10^{-4}	1×10^{-7}	1×10^{-4}
MAE [s]	0.175	0.0118	0.000001	0.00308
RMSE [s]	0.295	0.0260	0.000002	0.00691
MAPE [%]	992	112	7.63	97.2
RMSPE [%]	5400	456	14.7	336

(b) Power consumption

Layer	**Conv.**	**Pool.**	**Flat.**	**FC**
degree	1	1	2	1
α	1×10^{-2}	1×10^{-3}	1×10^{-5}	1×10^{-1}
MAE [W]	0.0248	0.0315	0.0179	0.118
RMSE [W]	0.0371	0.0398	0.0239	0.131
MAPE [%]	0.76	0.98	0.58	3.51
RMSPE [%]	1.13	1.23	0.77	3.93

5.2 Predictions on Training Data

The evaluation results of predictions done by the regression models are shown in Figure 4 and Table 5.

In Figure 4, the top and bottom rows show the evaluation results of the execution time predictions and the power consumption predictions, respectively. Columns are grouped by layer types, in the order of convolutional layer, max-pooling layer, flatten layer, and fully connected layer. Each graph is a scatterplot of prediction values on the y-axis and truth values on the x-axis, with s and W as the units for execution time and power, respectively. For instance, the graph at the intersection of the execution time row and flatten layer column shows the scatterplot between truth values and prediction values obtained from the flatten layer's execution time model. In addition, each black point on the graph represents a 1-layered neural network of the same layer type but with different hyperparameters values. Each point is plotted with a 50% opacity so the distributions of values can be visible. For example, in the scatterplot of the convolutional layer's execution time model, the points on the lower left side of the graph appeared darker, indicating that most measured

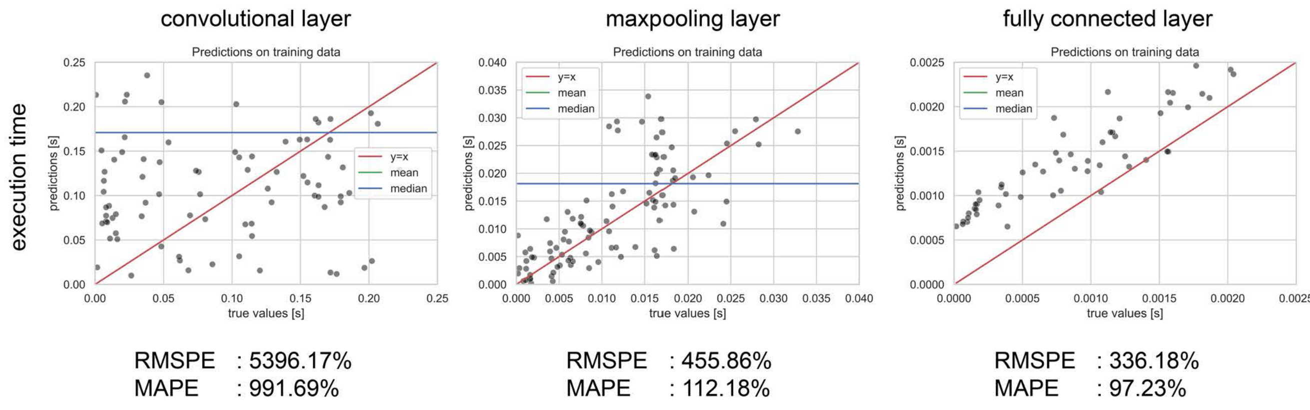


Fig. 5: Zoomed-in version of the scatterplots of prediction evaluations of execution time models for convolutional layer, max-pooling layer and fully connected layer.

execution times lie between 0 s and 2 s. Also, the red line labeled with $y = x$ in each scatterplot represents the ideal situation where all predicted values by the regression model are equal to the true values. The horizontal distance from a black point to the red line refers to the prediction accuracy for a particular 1-layered neural network.

5.2.1 Execution Time Models

Observation 1: From Figure 4, most points are near the red line, suggesting that our execution time models have good prediction abilities. However, the large values of relative errors (i.e., RMSPE and MAPE) in the convolutional layer, max-pooling layer, and fully connected layer columns in Table 5 suggest the opposite.

To explain this observation, we calculated the relative errors of the execution time models' predictions when the actual execution time is relatively short. Precisely, relative errors of the execution time models predictions were calculated for convolutional layers, max-pooling layers, and fully connected layers when the actual execution time is less than 25 s, 0.040 s, and 0.0025 s, respectively. Figure 5 shows the scatterplots of prediction and true values for layers with relatively short execution time. From Figure 5, we found out that the prediction errors when execution time is relatively short contribute to most of the errors reported in Table 5.

The large relative errors when execution time is relatively short can be explained by two possible reasons. First, the execution time models failed to fit well for these 1-layered neural networks—especially when the 1-layered neural network has a short execution time—which caused them to predict badly. The second reason is based on how relative errors are calculated (Section 5.1). As execution time becomes small, the denominator in the equation for RMSPE and MAPE becomes smaller, thus producing a large value of relative error.

5.2.2 Power Consumption Models

Observation 2: From Figure 4, the predictions performed by the power consumption models appeared to be inaccurate, but they have relative errors of less than 5% (Table 5).

One possible reason for this observation is that a bad prediction would not contribute to large relative error due to the lack of variation in truth values of power consumption. For all layer types, the true power consumption values have an average value range of 0.325 W, which magnitude is only about 10% of the power consumption.

Observation 3: We can also obtain from Figure 4 that the power models for the convolutional, max-pooling and fully connected layers produce near-constant predictions.

The polynomial equations for power models of the convolutional and fully connected layers only have the y-intercept (a constant) as the effective term (i.e., term with non-zero coefficient). To further investigated if the power models learned any relationships between layer hyper-parameters and power consumption from the training data, we compare their error metrics to two naive power models. The two naive power models are models that consistently predict the mean value and the median value of the training data, respectively. From this comparison, we found out that the power models of convolutional layer and fully connected layer have the same error metrics as the model that always predicts the mean value, suggesting that they learnt to predict the mean value from the training data.

Another interpretation of these two observations is that external factors (e.g., power modes, the temperature of SoC) have more significant effects on the power consumption of Raspberry Pi 4 compared to the layer hyperparameters.

5.3 Models Trained with Performance Counters

The large error in predictions by the models evaluated in Section 5.2 suggests the limit of layer hyperparameters in explaining the execution time and power consumption. To find out if this hypothesis is true, we evaluated another set of regression models, this time trained with both layer hyperparameters and performance counters values collected during the 1-layer neural networks executions. Because performance counters count the number of events at the hardware level, it is expected to explain execution time and power consumption better than layer hyperparameters.

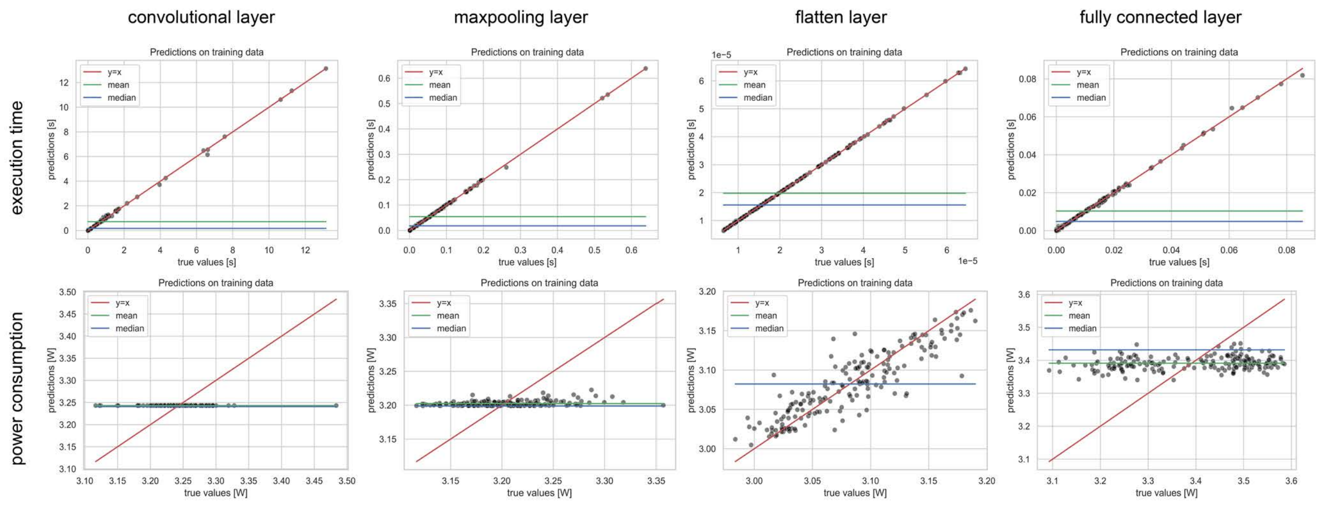


Fig. 6: Scatterplots of truth values and prediction values obtained from polynomial regression models trained with both layer hyperparameters and performance counters values.

Table 6: Prediction errors trained with both layer hyperparameters and performance counters.

(a) Execution time

Layer	**Conv.**	**Pool.**	**Flat.**	**FC**
degree	2	3	2	3
α	1×10^{-3}	1×10^{-5}	1×10^{-9}	1×10^{-5}
MAE [s]	0.0323	0.00108	7.05×10^{-8}	0.000314
RMSE [s]	0.0576	0.00166	1.05×10^{-7}	0.000546
MAPE [%]	117	17.0	0.44	31.3
RMSPE [%]	506	64.3	0.72	177

(b) Power consumption

Layer	**Conv.**	**Pool.**	**Flat.**	**FC**
degree	1	4	4	1
α	1×10^{-2}	1×10^{-2}	1×10^{-3}	1×10^{-2}
MAE [W]	0.0248	0.0314	0.0173	0.114
RMSE [W]	0.0371	0.0402	0.0229	0.128
MAPE [%]	0.76	0.98	0.56	3.37
RMSPE [%]	1.13	1.25	0.74	3.83

Raspberry Pi 4 has seven hardware performance counters in total [3], which consist of one clock cycle counter and six counters that can be set to count any events available in the processor [2]. In our case, we used the performance counters to count the occurence of the following seven events.

(1) *cycles*: number of clock cycles
(2) *instructions*: number of instructions retired
(3) *armv8_cortex_a72/insto_spec*: speculated number of instructions executed
(4) *ldst_spec*: speculated number of load and store instructions
(5) *armv8_cortex_a72/mem_access*: number of memory access
(6) *armv8_cortex_a72/l2d_cache/*: number of L2 data cache access
(7) *armv8_cortex_a72/l2d_cache_refill/*: number of L2 data cache refill

Figure 6 shows the scatterplot of predicted values against truth values of the models trained with layer hyperparameters and performance counters.

5.3.1 Execution Time Models

Observation 4: Comparing Figure 4 and Figure 6, the predicted values by this new set of models are closer to the red line. While being visible in the scatterplots, the error metrics in Table 6 agreed that this new set of models have smaller prediction errors.

5.3.2 Power Consumption Models

Observation 5: Although the power model of the convolutional layer has the same prediction errors like the power model trained without info of performance counters, the power models of other layer types achieved smaller prediction errors. However, from Figure 6, most of the predicted values still form a horizontal line, suggesting that neither performance counters managed to explain the change in power consumption across layers with different hyperparameters well.

5.3.3 Availability of Performance Counters

As performance counters are values counted during the executions, they will not be available for pre-execution predictions. Performing predictions with the knowledge of performance counters when it is not available might be possible if performance counters values are predictable by layer hyperparameters. Finding out a method to estimate performance counters values using layer hyperparameters is one of the possible future research directions.

6. Discussion

This section discusses the validity of using the number of clock cycles in prediction (Section 6.1) and how the input data could affect a layer's execution time and power consumption (Section 6.2).

6.1 Using Clock Cycles as Input Feature

In Section 4.3.3, 1-layered neural networks are executed at a fixed power mode, which will keep the clock frequency constant in an ideal situation. Based on the following equation,

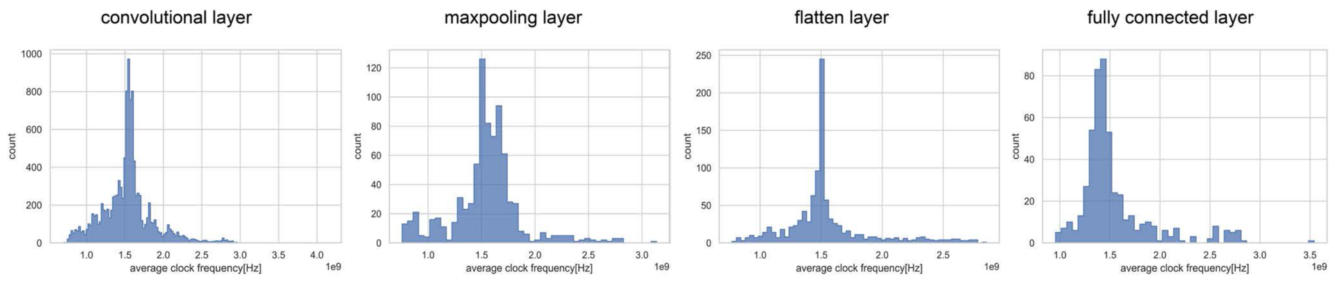


Fig. 7: Histogram of the average clock frequency during the executions of 1-layered neural networks

when the clock frequency is constant, using the number of clock cycles as one of the input features in the polynomial regression model can be considered as "cheating" because the coefficient learned by the regression model is the clock frequency, a value known before training.

$$\text{total execution time} = \frac{\text{total number of clock cycles}}{\text{average clock frequency}} \quad (1)$$

To find out if our models "cheated" during training, we investigated (1) the average clock frequency during the executions of each 1-layered neural network in Section 6.1.1 and (2) the significance of *the number of clock cycles* as an input feature in the regression models in Section 6.1.2.

6.1.1 Average Clock Frequency

Figure 7 shows the distributions of average clock frequency during the executions of 1-layered neural networks. Although all the histograms have a peak around 1.5 GHz—the same as the clock frequency at *performance* governor mode—the spread of values of the average clock frequency is evident, suggesting that the clock frequency during execution is not constant. Note that average clock frequency $\bar{f}_{\text{clk}}$ is calculated using the equation below, where *cycles* is the average number of clock cycles obtained from performance counters in one execution.

$$\bar{f}_{\text{clk}} = \frac{\text{cycles}}{\bar{T}_{\text{inference}}} \quad (2)$$

6.1.2 Significance of Clock Cycles as an Input Feature

In order to investigate the significance of the number of clock cycles as an input feature, multiple linear regression analyses were performed using *statsmodels*, a Python third-party library that focuses on statistical analyses of models.

Hypothesis: If our models "cheated," the results from regression analysis will show that clock cycles is the only input feature required to model execution time. Two metrics, (1) adjusted R^2 and (2) p-value of F-statistic, were used as measures of the significance of an input feature.

Results: From Table 7, the number of clock cycles (represented as *cycles* in the table) as an input feature has a p-value (p-value of F-statistic) of less than 0.05 for all layer types except the fully connected layer. This observation suggests that the number of clock cycles is a good estimate of execution time. However, other input features also have a p-value of less than 0.05, implying that the number of clock cycles is not the only input feature required to estimate execution time (i.e., go against our hypothesis). The changes in R^2_{adj} when we include the number of clock cycles adj in the input features of the regression model also suggest the same.

The results in Sections 6.1.1 and 6.1.2 suggest that on modern computing platforms, the relationship between execution time, clock cycles, and the clock frequency is not as simple as the equation shown above. Moreover, the underlying reasons for these observations and results are not clear. Therefore, more experiments and analysis are required before a concrete conclusion regarding the relationship between execution time, clock frequency, and clock cycles can be drawn.

6.2 The Ratio of Zero in Input Data

Another experiment was conducted to determine if input values affect the execution time and power consumption of 1-layered neural networks. First, we define the term *input data* as the input to a 1-layered neural network, and it consists of *input shape* and *input values*. Input shape decides the number of elements in input data (e.g., height, width, and dimension), and input values decide the values of the elements. In this experiment, each 1-layered neural network was executed with different input values using the methodology explained in Section 4. Precisely, each 1-layered neural network was executed and measured five times, each time with a different input data from the list { 0, 0.25, 0.50, 0.75, 1.0 }.

In this experiment, a different set of 1-layered neural networks was used; instead of randomly generating the 1-layered neural networks, each layer hyperparameter was handpicked. Precisely, the 1-layered neural networks were generated with all the possible combinations of hyperparameter values listed in Table 8. As a result, this new set of 1-layered neural networks contains 2250 convolutional layers, 162 max-pooling layers, 100 flatten layers, and 200 fully connected layers.

Figure 8 shows the distributions of execution time and power across different input values of all layer types. The points scattered vertically of each input value represent layers with different hyperparameters.

Table 7: Results of significance analysis of clock cycles as an input feature; p-values less than 0.05 are shown in bold.

(a) Convolutional layer

R^2_{adj} without cycles	0.939	
R^2_{adj} with cycles	0.940	(+0.0005)
term	**f-score**	**p-value**
y-intercept	207	**0.000**
cycles	10.2	$\mathbf{4.19 \times 10^{-24}}$
filters	9.64	$\mathbf{6.51 \times 10^{-22}}$
instructions	8.45	$\mathbf{3.18 \times 10^{-17}}$
armv8_cortex_a72/inst_spec/	8.26	$\mathbf{1.79 \times 10^{-16}}$
armv8_cortex_a72/l2d_cache_refill/	7.63	$\mathbf{2.50 \times 10^{-14}}$
channels	5.95	$\mathbf{2.76 \times 10^{-9}}$
kernel_size	5.74	$\mathbf{9.54 \times 10^{-9}}$
armv8_cortex_a72/l2d_cache/	5.71	$\mathbf{1.17 \times 10^{-8}}$
armv8_cortex_a72/mem_access/	5.07	$\mathbf{4.01 \times 10^{-7}}$
input_x_size	2.47	$\mathbf{1.34 \times 10^{-2}}$
input_y_size	2.47	$\mathbf{1.34 \times 10^{-2}}$
ldst_spec	2.13	$\mathbf{3.33 \times 10^{-2}}$
strides	1.71	8.68×10^{-2}

(b) Max-pooling layer

R^2_{adj} without cycles	0.924	
R^2_{adj} with cycles	0.926	(+0.0015)
term	**f-score**	**p-value**
y-intercept	67.6	**0.000**
cycles	4.24	$\mathbf{2.5 \times 10^{-5}}$
ldst_spec	3.55	$\mathbf{4.12 \times 10^{-4}}$
armv8_cortex_a72/mem_access/	3.19	$\mathbf{1.49 \times 10^{-3}}$
input_x_size	1.86	6.31×10^{-2}
input_y_size	1.86	6.31×10^{-2}
pool_size	1.41	1.59×10^{-1}
armv8_cortex_a72/l2d_cache_refill/	1.33	1.85×10^{-1}
channels	1.31	1.89×10^{-1}
instructions	1.00	3.13×10^{-1}
armv8_cortex_a72/inst_spec/	0.647	5.18×10^{-1}
strides	0.403	6.87×10^{-1}
armv8_cortex_a72/l2d_cache/	0.095	9.24×10^{-1}

(c) Flatten layer

R^2_{adj} without cycles	0.932	
R^2_{adj} with cycles	0.939	(+0.0063)
term	**f-score**	**p-value**
y-intercept	71.4	$\mathbf{5.31 \times 10^{-261}}$
cycles	7.13	$\mathbf{3.60 \times 10^{-12}}$
armv8_cortex_a72/l2d_cache/	3.82	$\mathbf{1.53 \times 10^{-4}}$
armv8_cortex_a72/l2d_cache_refill/	1.63	1.03×10^{-1}
ldst_spec	0.389	6.97×10^{-1}
armv8_cortex_a72/inst_spec/	0.369	7.12×10^{-1}
channel	0.248	8.04×10^{-1}
armv8_cortex_a72/mem_access/	0.152	8.79×10^{-1}
instructions	0.078	9.38×10^{-1}
input_x_size	0.034	9.73×10^{-1}
input_y_size	0.034	9.73×10^{-1}

(d) Fully connected layer

R^2_{adj} without cycles	0.962	
R^2_{adj} with cycles	0.962	(+0.000)
term	**f-score**	**p-value**
y-intercept	252	**0.000**
instructions	1.96	5.07×10^{-2}
armv8_cortex_a72/inst_spec/	1.86	6.31×10^{-2}
armv8_cortex_a72/l2d_cache_refill/	1.60	1.11×10^{-1}
armv8_cortex_a72/mem_access/	1.31	1.90×10^{-1}
ldst_spec	1.26	2.09×10^{-1}
num_inputs	0.806	4.21×10^{-1}
num_neurons	0.704	4.82×10^{-1}
armv8_cortex_a72/l2d_cache/	0.523	6.01×10^{-1}
cycles	0.300	7.64×10^{-1}

Table 8: The hyperparameter values handpicked for each layer used in 1-layered neural network generation

	Conv.	**Pool.**	**Flat.**	**FC**
input size (2D/1D)	$\{10 + 55n \mid 0 \le n \le 4\}$	$\{10, 20, 30\}$	$\{2 + 2n \mid 0 \le n \le 9\}$	$\{3600 + 600n \mid 0 \le n \le 9\}$
# channel	$\{96 + 64n \mid 0 \le n \le 4\}$	$\{96, 176, 256\}$	$\{96 + 32n \mid 0 \le n \le 9\}$	
# filters/neurons	$\{96 + 64n \mid 0 \le n \le 4\}$			$\{600 + 400n \mid 0 \le n \le 9\}$
kernel/pool size	$\{3, 5, 7\}$	$\{2, 3, 4\}$		
strides	$\{1, 2, 3\}$	$\{1, 2, 3\}$		
padding	{yes, no}	{yes, no}		
activation function	{relu}			{relu, softmax}

When fully connected layers were executed with input values of all zeros, the execution times were much shorter, and the power consumption was smaller than other input values. One of the reasons for this observation is implementation optimization of fully connected layer (e.g., sparse matrix multiplication) which reduces the number of operations required in one inference. This reasoning is confirmed by checking the number of retired instructions in one inference—treated as a reference for the number of operations performed, for simplicity—. The mean number of retired instructions in one inference—across fully connected layers of different hyperparameters—when the input values are all zeros and when the input values are all non-zeros were 5.602×10^4 and 1.055×10^7 respectively. In a more realistic situation, input values of a fully connected layer will be a mix of zero values and non-zero values instead of all zeros. We suspect an association exists between the zeros ratio in input values and execution time or power consumption. Also, the ratio of zeros in the learned weights of the fully connected layer might affect execution time or power consumption, as they are also involved in matrix multiplications. We plan to investigate the relationship between the ratio of zeros and layers' execution time and power consumption in the future.

For the other layers, no apparent differences in execution time can be seen—from the scatterplots—when the layers are executed with different input values. Although differences in power consumption distributions can be observed, there are overlaps between the distributions, so the association between input values and power consumptions of these layers cannot be confirmed without further analysis and experiments.

7. Conclusions

As the need for sustainable computing soars, intermittent executions on energy harvesting devices with effective checkpointing are required. As the first step towards proposing a layer-wise selective checkpointing scheme with power/performance models, we implement a

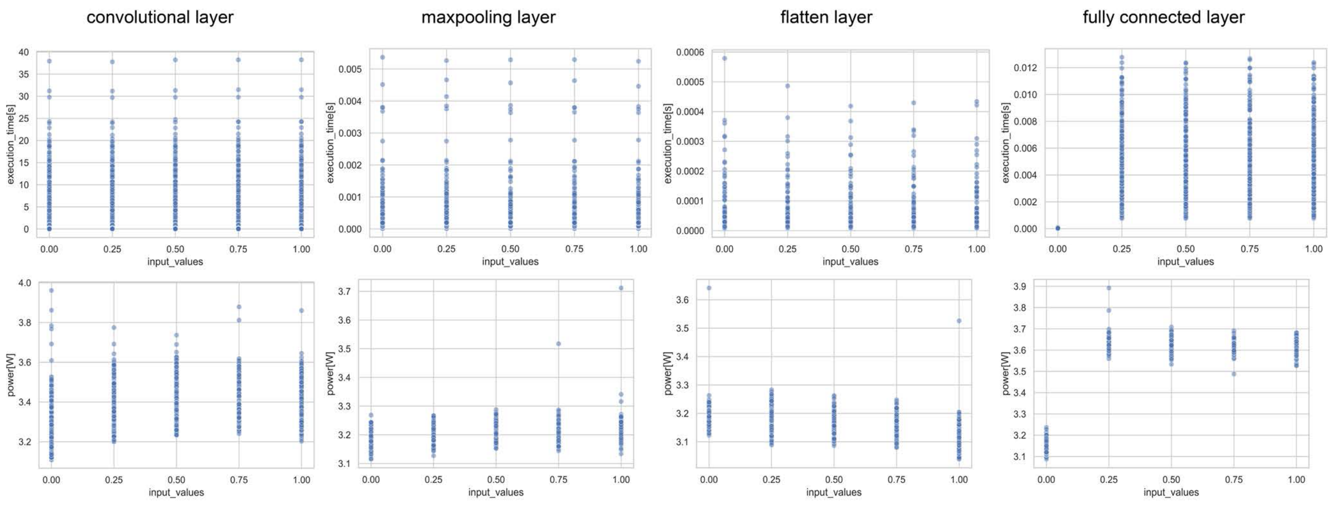


Fig. 8: Distribution of execution time and power consumption with different input values

power/performance prediction framework of CNN inferences on single-board computers based on NeuralPower [4] and evaluate its prediction precision.

We found out that there is a limit in layer hyperparameters' ability to model a layer's execution time and power consumption. Furthermore, although performance counter values can improve the prediction, they are usually unavailable at the pre-execution stage. Moreover, implementation optimization like sparse matrix multiplication gives birth to the dependency of execution time and power consumption on the input values.

Acknowledgments This work was supported by MIC under a grant entitled "R&D of ICT Priority Technology (JPMI00316)".

References

[1] Ardito, L. and Torchiano, M.: Creating and Evaluating a Software Power Model for Linux Single Board Computers, *Proceedings of the 6th International Workshop on Green and Sustainable Software*, GREENS '18, New York, NY, USA, Association for Computing Machinery, pp. 1–8 (online), DOI: 10.1145/3194078.3194079 (2018).

[2] arm Developer: Events, ARM Cortex-A72 MPCore Processor Technical Reference Manual r0p3, `https://developer.arm.com/documentation/100095/0003/Performance-Monitor-Unit/Events?lang=en`, (2021). Accessed: 2022-01-26.

[3] arm Developer: PMU functional description, ARM Cortex-A72 MPCore Processor Technical Reference Manual r0p3, `https://developer.arm.com/documentation/100095/0003/Performance-Monitor-Unit/PMU-functional-description?lang=en`, (2021). Accessed: 2022-01-26.

[4] Cai, E., Juan, D.-C., Stamoulis, D. and Marculescu, D.: *NeuralPower*: Predict and Deploy Energy-Efficient Convolutional Neural Networks, *Proceedings of the Ninth Asian Conference on Machine Learning* (Zhang, M.-L. and Noh, Y.-K., eds.), Proceedings of Machine Learning Research, Vol. 77, Yonsei University, Seoul, Republic of Korea, PMLR, pp. 622–637 (online), available from ⟨https://proceedings.mlr.press/v77/cai17a.html⟩ (2017).

[5] Colin, A. and Lucia, B.: Termination Checking and Task Decomposition for Task-Based Intermittent Programs, *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, New York, NY, USA, Association for Computing Machinery, pp. 116–127 (online), DOI: 10.1145/3178372.3179525 (2018).

[6] Corp., K. E.: PMX-A Series, DC Power Supplies. Accessed: 2022-01-11.

[7] Géron, A.: *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*, O'Reilly Media, Inc. (2019).

[8] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J. and Tang, L.: Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, New York, NY, USA, Association for Computing Machinery, pp. 615–629 (online), DOI: 10.1145/3037697.3037698 (2017).

[9] Kaup, F., Gottschling, P. and Hausheer, D.: PowerPi: Measuring and modeling the power consumption of the Raspberry Pi, *39th Annual IEEE Conference on Local Computer Networks*, pp. 236–243 (online), DOI: 10.1109/LCN.2014.6925777 (2014).

[10] Lucia, B., Balaji, V., Colin, A., Maeng, K. and Ruppel, E.: Intermittent Computing: Challenges and Opportunities, *2nd Summit on Advances in Programming Languages (SNAPL 2017)* (Lerner, B. S., Bodík, R. and Krishnamurthi, S., eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71, Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 8:1–8:14 (online), DOI: 10.4230/LIPIcs.SNAPL.2017.8 (2017).

[11] Oh, Y. H., Kim, S., Jin, Y., Son, S., Bae, J., Lee, J., Park, Y., Kim, D. U., Ham, T. J. and Lee, J. W.: Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling, *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 584–597 (online), DOI: 10.1109/HPCA51647.2021.00056 (2021).

[12] Reif, S., Herzog, J. H. B., Hönig, T. and Schröder-Preikschat, W.: Precious: Resource-Demand Estimation for Embedded Neural Network Accelerators, *First International Workshop on Benchmarking Machine Learning Workloads on Emerging Hardware* (2020).

[13] TensorFlow: TensorFlow Lite runtime in Python, `https://www.tensorflow.org/lite/guide/python#about_the_tensorflow_lite_runtime_package`. Accessed: 2022-01-11.

[14] TensorFlow: Post-training quantization, `https://www.tensorflow.org/lite/performance/post_training_quantization` (2021). Accessed: 2022-02-03.

[15] Umesh, S. and Mittal, S.: A survey of techniques for intermittent computing, *Journal of Systems Architecture*, Vol. 112, p. 101859 (online), DOI: https://doi.org/10.1016/j.sysarc.2020.101859 (2021).