Energy Benchmarking of Deep Neural Networks

Synopsis of the Project Report to be submitted
in Fulfillment of the Requirements
for the Award of the Degree of

Bachelor of Technology (Hons.)

in

Electrical Engineering

by

Parth Mahesh Paradkar
18EE10033

Under the supervision of

Dr. Debdoot Sheet

Department of Electrical Engineering
Indian Institute of Technology Kharagpur
April 2022

DECLARATION

I certify that

a. The work contained in the thesis is original and has been done by myself under the general supervision of my supervisor.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in writing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

f. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Parth Paradkar

Contents

1. Introduction

In recent years, deep neural networks have seen great interest and adoption in a variety of fields. With this upsurge in adoption, the amount of compute has increased exponentially. There is an increase in energy costs and carbon footprint associated with the training and inference of deep learning models. Various embedded and mobile systems running deep learning-based AI applications like computer vision, voice assistants, etc. have limited battery capacities. Processors running models with high energy consumption deplete batteries at a higher rate, making the application unfit for use. While contemporary research has focused majorly on accuracy optimizations, there is a need for energy optimization of models and benchmarking of hardware used for deep learning. With this need in mind, the project is aimed at benchmarking the energy consumption of deep learning models on different processors. For the purpose of benchmarking, convolutional neural networks shall be considered. The objective is to design and implement an energy estimation methodology at the processor level for the forward pass computations of a convolutional neural network. Through this, a metric will be created to estimate the per pixel energy consumption of convolutional neural networks on different processors, using which benchmarks for the energy consumption of the processor during inference can be created.

In order to obtain benchmarking metrics, it is necessary to measure CPU-level energy consumption, floating point operations (FLOPS) and memory consumption of the process running the deep neural network program. Intel SoCWatch is a tool that provides CPU-level energy metrics while running a given compiled object file. In the project, a wrapper library written in Python- “DeepMeter”, was built on top of Intel SoCWatch. DeepMeter invokes SoCWatch implicitly after building a neural network program from the given user input. DeepMeter offers a layerwise breakdown of energy consumption of a particular neural network, the data for which is provided in a JavaScript Object Notation (JSON) file.

Keywords: energy measurement, deep neural networks, benchmarking

2. Scope and Objectives

Current optimizations for neural networks focus on saving the number of computations during training and inference at the cost of accuracy. Increasingly, neural networks are finding applications in artificial intelligent systems running on mobile and IoT devices. Inference from trained models has popularly been done through cloud computing. However, with an increased demand for edge computing, new requirements come into the picture-

1. Network access- When implementation of a system using DNNs is dependent on processing taking place in the cloud, the application is restricted to the areas with good internet connectivity. To uncouple the application from the internet, the trained model needs to be placed at the edge device capable of processing inference computations.
2. Privacy- Certain applications like those in healthcare require the data of the end user to be restricted to the local device, without transmission to any external server or database.
3. Latency- Real-time applications like voice assistants, self-driving cars require making rapid decisions based on the incoming data. This makes low latency a must. Decoupling the application from the cloud and carrying out the inference computations on the device itself is required.

Edge devices like mobile devices and IoT devices are constrained by their small battery sizes. Thus, for implementation of DNNs on these devices, energy consumption becomes an important factor to consider.

It is observed that the energy consumed in accessing the data from the memory is many orders of magnitude larger than that consumed in the computations taking place in the processor. Thus, energy consumption is dependent on the way the different processors implement their memory architectures i.e. the RAM, levels of caches, buffers. Due to this, the process for developing energy-efficient models has to take into consideration the memory architecture, and differs from methods used to optimize the number of computations. When developing processors, accelerators and application specific integrated circuits (ASICs) with energy efficiency in mind, benchmarks are required to make comparisons to existing hardware architectures. Thus, benchmarking of different existing hardware architectures is required.

With the variety of hardware architectures available in the market, which includes differing CPU and GPU architectures that have different performances, the choice of hardware required for the specific purpose of the application makes a significant impact on its efficient working. Simultaneously, data from energy analyses can be used to optimize the neural network architecture in implementation. This applies especially for cross-platform applications and applications where the choice of hardware architectures is limited by availability and cost. It can be predicted by FLOPS and memory consumption measurement and also empirically observed after energy measurement that certain types of layers in the neural network architecture consume differing levels of energy. With variation of energy consumption under consideration, various techniques can be applied for its reduction in specific layers. These techniques often have an accuracy tradeoff, which leads to different optimization strategies that are suited to the particular application in question. If the feasibility of the application is reliant on the energy efficiency of the neural network architecture it uses, the accuracy tradeoff can be accepted to a sufficient extent while reducing its energy consumption.

The objective of the project is to develop a energy measurement framework that allows for the benchmarking of hardware architectures for running different neural networks. It aims to establish a reliable unit of measurement for the energy consumption of deep neural networks. The project shall primarily focus on convolutional neural networks (CNNs) and hence aim to establish per pixel energy consumption as a unit of measurement. The project also aims to create a energy measurement library tailored to deep neural networks, which can be used during benchmarking and optimization.

3. Literature Survey

3.1 Relationship between energy, computations and memory accesses

Research regarding the energy consumption of deep neural networks is fairly nascent. Study has been spurred by the requirement of deep learning applications to be energy efficient. One such study is that done by Dr. Vivienne Sze at Massachusetts Institute of Technology. Sze's group at MIT (Yang et al. 2017) has come up with an energy-estimation methodology that is based on their Eyeriss spatial architecture (Chen et al. 2016). The total energy consumption can be divided into two parts- energy consumed in the CPU during computations (E_{comp}) and energy consumed in accessing the data from the memory (E_{data}).

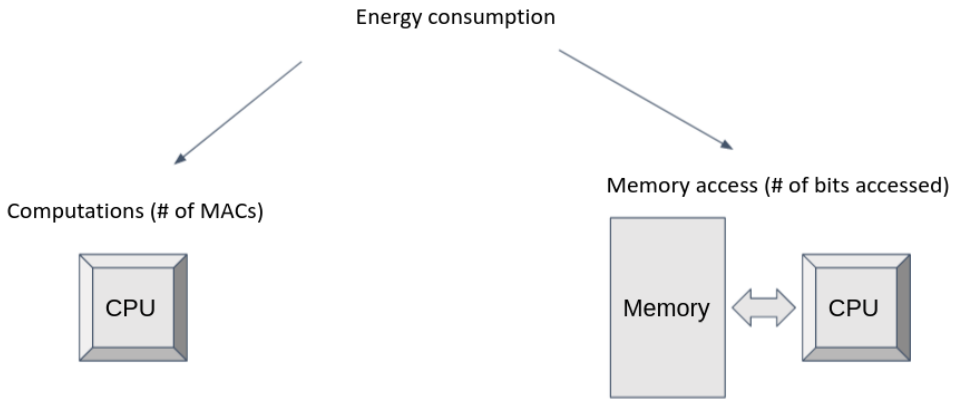

Figure 1: Division of energy consumption in a process

Majority of the computations in CNNs are multiply-and-accumulate operations (MACs). Thus, the amount of energy consumed in performing one MAC operation scaled by the total number of MAC operations will yield E_{comp}.

For estimating E_{data}, the number of bits accessed at each level in the memory hierarchy is measured, it is multiplied with the energy required to access a single bit at that memory level and a sum is taken over all the memory levels.

$$E_{data} = \sum_{i=1}^{N} E_i \times (n_{bits})_i \quad (1)$$

The memory hierarchy consisting of Random Access Memory (RAM), buffers, multiple levels of instruction and data caches is organized such that the memory levels closer to the CPU consume less energy. However, lower levels of memory like caches have smaller capacities. With this tradeoff in place, there are different strategies to optimize energy consumption and space in the respective memory levels. In each of these strategies, the data present in the various memory levels differs. Depending on the strategy being used, energy for the different levels of memory can be calculated and finally accumulated.

3.2 Eyeriss Architecture

With the aim of accurately estimating the energy consumption, Sze's group developed "Eyeriss", an accelerator for deep neural networks that exploits data reuse by parallelizing MAC operations. The memory architecture in Eyeriss comprises of various memory levels of differing energy levels of access. The convolution operations in the CNN are off-loaded to the accelerator, which contains a buffer and a processing element (PE) array.

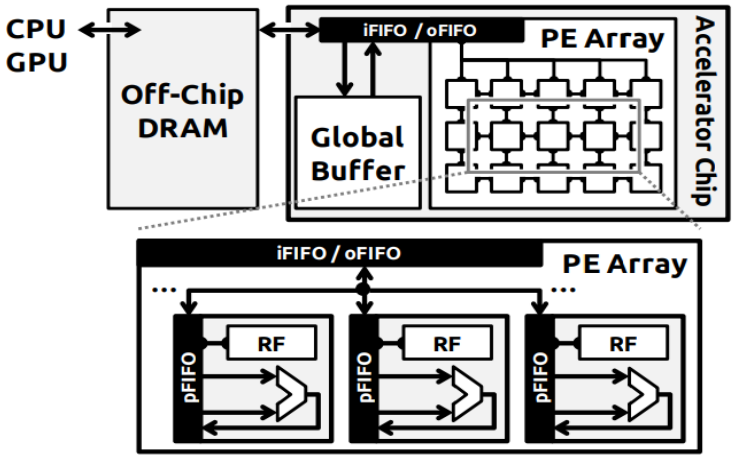

Figure 2: Eyeriss Architecture

Each processing element has its own register file and an ALU, which handles MAC operations in this case. The PEs are connected such that they can inter-communicate the information they

process. This is useful in accumulation operations. In the Eyeriss architecture, there are four types of memory accesses where energy is consumed- Dynamic Random-Access Memory (DRAM), Global buffer, inter-PE communication, Register File (RF). For each of these memory accesses, the energy consumption is measured on-chip and the values are normalized with the energy consumed in one MAC operation.

Table 1: Normalised energy consumption of memory accesses in Eyeriss

Memory hierarchy	Normalized energy consumption	Number of reuses of one data value
DRAM	200x	a
Global buffer	6x	b
Arrays (Inter-PE)	2x	c
RF	1x	d

The values a, b, c, d are assigned to the number of times a single data value is accessed in the respective memory level.

These values differ based on the data reuse strategy being used. With these values in consideration, the total energy consumed is calculated.

$$EC_{total} = a \times EC_{DRAM} + ab \times EC_{Buffer} + abc \times EC_{array} + abcd \times EC_{RF}$$

A web based tool that uses this methodology for energy estimation has been made available.[1]

[1] https://energyestimation.mit.edu/

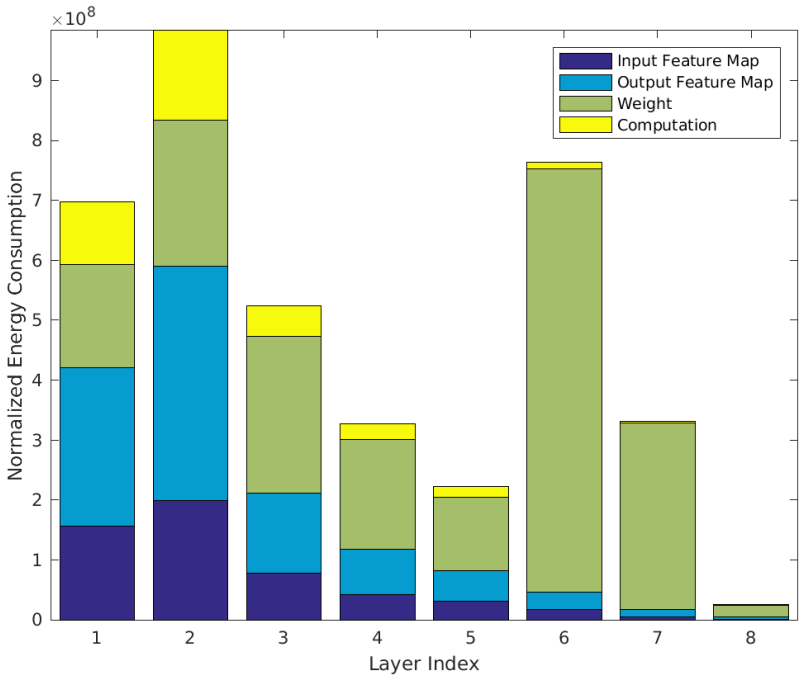

Figure 3: Layer-wise energy consumption in AlexNet

Figure 3 shows a sample result from the tool. Here, layer-wise breakup of the energy consumption in AlexNet calculated using the aforementioned energy estimation methodology is represented graphically, which is further categorised into the energy consumed in the processing of the input feature maps, output feature maps, weights and the computation. Evidently, the energy consumed in computations is less than the energy consumed in accessing the memory.

The Eyeriss project offers insight into the relationship between energy consumption and the memory accesses. It rejects the intuitive notion that energy consumption should scale linearly with the number of FLOPS and emphasizes the role of memory consumption in energy consumption. Since processing of different neural network layers involves different amounts of memory accesses at runtime, this observation helps us explain the variation in the energy consumed by different layers in the upcoming experiments. This needs to be considered during benchmarking a particular neural network on a given hardware architecture. The Eyeriss project illustrates a layerwise approach to energy measurement, which is well-suited for different optimization techniques. The current project also draws inspiration from the easily accessible

web-based energy estimation tool that has been published for free public use. The aim of the current project is to develop a generalised tool that can be used across hardware architectures. The energy in this case needs to be measured and not estimated, since the memory accesses and energy levels at different memory layers in the memory architecture is not known beforehand. Thus, a hardware-level energy measurement tool is required to accomplish the purpose.

4. Energy Measurement Methodology

In order to develop a benchmarking framework, a methodology needs to be established for the measurement of energy consumption in relation to deep neural networks. Specifically, each layer needs to be taken into consideration separately while performing energy measurement.

4.1 Intel SoCWatch

Energy consumption is measured using the Intel SoCWatch tool. SoCWatch is a part of the oneAPI toolkit developed by Intel, which is available as a part of Intel's VTune platform. It is used for energy monitoring and analysis on the system. Intel SoCWatch is accessed from the command line, with the executable file that needs to be monitored as an argument. For the purpose of this project, the power feature of the tool shall be used, which is accessed via the "-f power" command line parameter. This function returns the power and energy consumption of the CPU and DRAM packages present in the system. A sample output from SoCWatch is given in Figure 4.

```
---------------------------------------------------------------------------
Intel(R) SoC Watch for Linux* OS Version 2021.1 [Apr 15 2021]
Build Ref: f8308ca4eef08f738340d9b2dfb7b79a9147b24c
Post Processed using SoC Watch for Linux* OS Version 2021.1 [Apr 15 2021]
Post Processed Build Ref: f8308ca4eef08f738340d9b2dfb7b79a9147b24c
Copyright (c) 2021 Intel Corporation. All Rights Reserved.
Platform power analysis tool for use with Intel processors/chipsets/platforms.
*Other names and brands may be claimed as the property of others.
---------------------------------------------------------------------------

Command line options: -f power -p ./controller/run.py

Program Started: 2022-03-08 10:13:18 GMT
Data Collection Started: 2022-03-08 10:13:18 GMT
Collection duration (sec):0.784405

System Name: miriad1a
Operating System: Linux [4.15.0-153-generic]
CPU: Skylake Server
CPU ID (family.model.stepping): 0x6.0x55.0x4
```

Platform ID: 0b000
Integrated GPU Device: NA
PCH: (0xa1c1)
EDRAM Present: No
Total # of packages: 2
Total # of cores: 48
Total # of logical processors: 96
==========
Package Power
==========
Note: Die-level power is included in package power on platforms with a multi-die CPU topology.
Package Power Summary: Average Rate and Total
Component , Metric Type, Average Rate (mW), Total (mJ)
--------- , -----------, ----------, ----------
CPU/Package_0, Power , 30131.32 , 23626.10
CPU/Package_1, Power , 33905.20 , 26585.45

Package Power Summary: Total Samples Received
Component , Total # of samples, Min sampling interval (msec), Max sampling interval (msec), Avg sampling interval (msec)
--------- , ----------, ----------, ----------, ----------
CPU/Package_0, 6 , 2.28 , 279.49 , 156.16
CPU/Package_1, 4 , 91.41 , 368.50 , 260.25
==========
DRAM Power
==========
DRAM Power Summary: Average Rate and Total
Component , Metric Type, Average Rate (mW), Total (mJ)
--------- , -----------, ----------, ----------
DRAM/DRAM_0, Power , 8636.69 , 6772.16
DRAM/DRAM_1, Power , 10831.17 , 8492.92

DRAM Power Summary: Total Samples Received
Component , Total # of samples, Min sampling interval (msec), Max sampling interval (msec), Avg sampling interval (msec)
--------- , ----------, ----------, ----------, ----------
DRAM/DRAM_0, 6 , 2.22 , 279.49 , 156.15
DRAM/DRAM_1, 4 , 91.41 , 368.50 , 260.25

Figure 4: Sample SoCWatch output

4.2 Experimental Steps

An untrained Lenet-5 model is taken for the purpose of experimentation. The model is written using the PyTorch framework in Python and is present in a single executable Python file.

```
import torch.nn as nn
class Lenet5(nn.Module):
    def __init__(self):
        super(Lenet5, self).__init__()
        self.pool = nn.MaxPool2d(kernel_size = (2, 2), stride = (2, 2))
        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 6, kernel_size = (5, 5), stride = (1, 1), padding = (0, 0))
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size = (2, 2), stride = (2, 2))
        self.conv2 = nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size = (5, 5), stride = (1, 1), padding = (0, 0))
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size = (2, 2), stride = (2, 2))
        self.linear1 = nn.Linear(400, 120)
        self.relu3 = nn.ReLU()
        self.linear2 = nn.Linear(120, 84)
        self.relu4 = nn.ReLU()
        self.linear3 = nn.Linear(84, 10)

     def _call_func(self, func, x):
        return func(x)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = x.reshape(x.shape[0], -1)
        x = self.linear1(x)
        x = self.relu3(x)
        x = self.linear2(x)
        x = self.relu4(x)
        x = self.linear3(x)
        return x
```

Figure 5: Lenet5 model written in PyTorch

To establish correlation between input image size i.e number of pixels and the amount of energy consumed for inference, an experiment is conducted to measure the energy consumed for different input image sizes for each layer in the Lenet5 model. For each iteration of the experiment, an image from the MNIST dataset is taken, resized to the required dimensions and passed through the layer taken in consideration. The Intel SoCWatch relies on sampling the CPU for energy data.

As each layer undergoes optimization separately, the energy measurement framework should ideally provide information in a staggered manner, with the separate energy consumption metrics for each layer. Since FLOPS and memory consumption measurements are often utilized in optimization methods, the framework must be extensible to include these parameters as well.

To ensure that the tool collects enough samples to produce an accurate result, the program needs to have sufficient runtime. For this, inference for each layer is conducted for a batch of 10,000 images. To streamline data collection, the process of running SoCWatch and parsing its output is automated using another Python script. This automation script reads the output data from the tool and saves it as a comma-separated value (CSV) file. This data can be subsequently imported into a spreadsheet for analysis.

5. Experiments

In order to develop a methodology to measure energy consumption of deep neural networks, a series of experiments was conducted. Through these experiments, Python code was developed to adapt the SoCWatch tool to the purpose of measuring energy for each layer.

5.1 Preliminary measurements using SoCWatch

SoCWatch provides the various energy metrics that can be collected either individually or in groups. For the purpose of this experiment, the power and energy metrics are collected.

Results

```
==============
Package Power
==============
Note: Die-level power is included in package power on platforms with a multi-die CPU topology.

Package Power Summary: Average Rate and Total
Component       , Metric Type, Average Rate (mW), Total (mJ)
---------       , -----------  , ----------------  , ----------
CPU/Package_0, Power       , 22027.14          , 763.24
CPU/Package_1, Power       , 20392.05          , 706.73

Package Power Summary: Total Samples Received
Component    , Total # of samples, Min sampling interval (msec), Max sampling interval (msec), Avg sampling interval
(msec)
---------    , ----------------, --------------------------, --------------------------, --------------------------
CPU/Package_0, 3             , 5.01                    , 26.25                   , 15.63
CPU/Package_1, 2             , 31.21                   , 31.21                   , 31.21

==========
DRAM Power
==========
DRAM Power Summary: Average Rate and Total
Component  , Metric Type, Average Rate (mW), Total (mJ)
---------  , ----------, ----------------, ----------
DRAM/DRAM_0, Power     , 6175.33         , 214.05
DRAM/DRAM_1, Power     , 4477.13         , 155.21

DRAM Power Summary: Total Samples Received
Component  , Total # of samples, Min sampling interval (msec), Max sampling interval (msec), Avg sampling interval
(msec)
---------  , ----------------, --------------------------, --------------------------, --------------------------
DRAM/DRAM_0, 3             , 5.02                    , 26.19                   , 15.60
DRAM/DRAM_1, 2             , 31.21                   , 31.21                   , 31.21
```

Figure 6: SoCWatch output

Observations

In Figure 1, measurements of power, energy and the sampling time can be found for two DRAM and the CPU packages that are present in the test machine. It can be observed that the tool could only gather a limited number of samples in the runtime. Thus, programs with larger runtimes can provide better results with more samples. The above output is used by the automation script and is transferred to a CSV file

5.2 Cumulative layer-wise measurement

Initially, a cumulative method was used to measure the energy consumption. Each layer in the network requires different input vector dimensions. During the inference of the model, the output feature map serves as the input feature map of the following layer. Thus, the dimensions are transformed automatically without any external interference. To avoid manual intervention, we use a cumulative method, where we run inference for the layer in consideration along with all the layers that precede it. Individual values are calculated differentially from the cumulative values. The method is outlined in Figure 2.

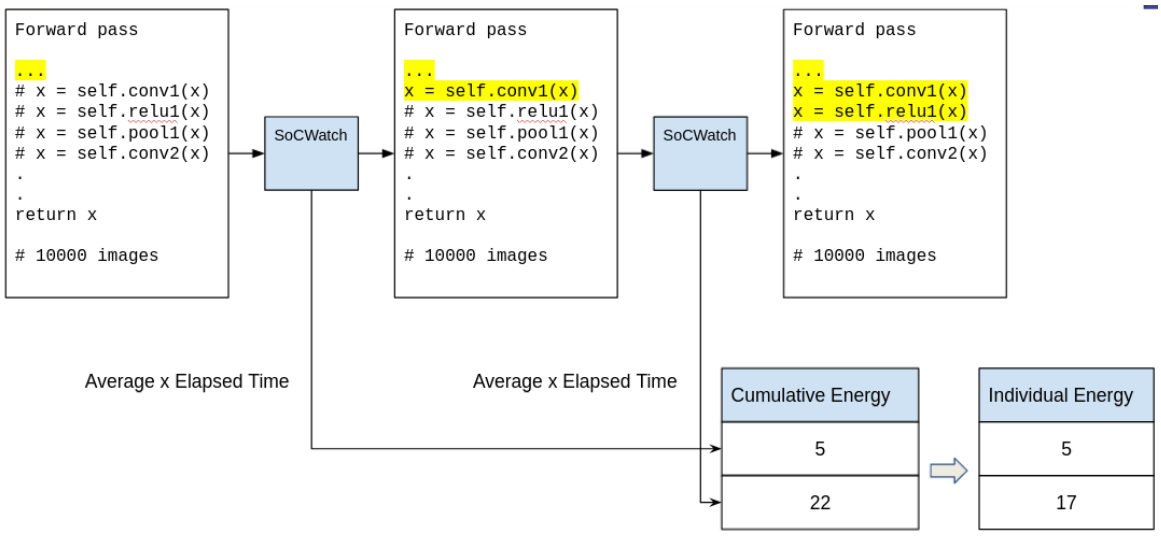

Figure 7: Cumulative measurement method

We consider the cumulative energy consumption in each step, wherein the measured energy consists of the energy consumption of all the previous layers. After the data is collected until the

last layer, the individual energy for a particular layer is calculated by subtracting the energy measured until the previous step. We also measure the energy consumption with no layers. This energy consists of the energy consumed in loading libraries and other initialization tasks.

Results

Table 2: Cumulative energy measurement results

Layer	CPU/Package_0_energy (mJ)	Individual energy CPU 0 (mJ)	CPU/Package_1_energy (mJ)	Individual energy CPU 1 (mJ)	DRAM/DRAM_0_energy (mJ)	Individual energy DRAM 0 (mJ)	DRAM/DRAM_1_energy (mJ)	Individual energy DRAM 1 (mJ)
Baseline	62243	62243	74172	74172	12438	12438	29183	29183
conv1	445943	383700	464263	390092	106920	94481	109165	79983
relu1	455658	9714	478348	14085	110768	3849	118212	9047
pool1	503784	48127	520891	42543	119252	8483	126411	8198
conv2	2584795	2081011	2702210	2181319	862027	742775	849309	722899
relu2	2473235	-111560	2845949	143740	865357	3331	900646	51337
Reshape	2681068	207833	2656503	-189447	866185	828	831582	-69064
pool2	2766228	85160	2578476	-78027	869182	2996	878148	46567
linear1	4449406	1683178	4464221	1885744	1457614	588433	1465472	587324
relu3	4630003	180597	4165568	-298652	1486105	28490	1439632	-25840
linear2	4464019	-165984	4278065	112497	1421035	-65070	1362576	-77056
relu4	4655295	191276	4223219	-54846	1495066	74031	1326617	-35959
linear3	4357337	-297958	4461393	238174	1420736	-74330	1433304	106687

Observations

In Table 1, the energies consumed in two packages of CPU and DRAM are obtained. The packages are structured as shown in Figure 3.

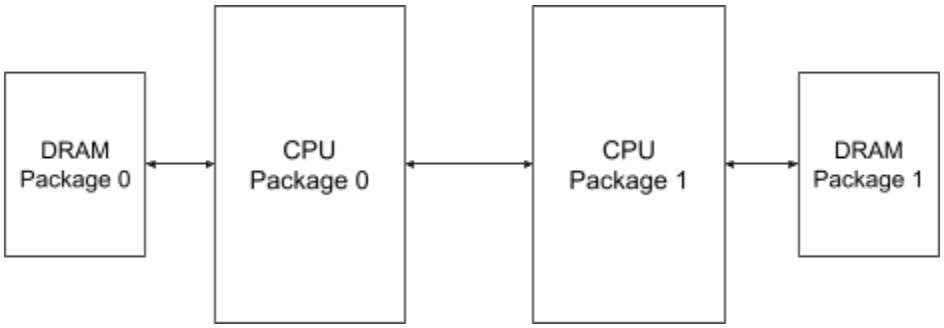

Figure 8: CPU and DRAM Package structure

It can be observed that convolutional layers (conv1 and conv2) and fully connected layers (linear1, linear2 and linear3) consume significantly higher energies as compared to other layers. This is to be expected, given the higher number of memory accesses and computations involved. Erroneous negative values are obtained in some cases (highlighted in red). This is due to the variations in CPU states in between iterations, resulting in different total energies for the same layer. To overcome this disadvantage of the cumulative method, the energy consumption of each layer is measured individually after reshaping the input feature map.

5.3 Individual layer-wise measurement

Now, each layer is considered individually. In order to establish the correlation between input feature size i.e number of pixels and the energy consumed, the dimensions of the input image are increased progressively. The relation between number of pixels and the energy consumption can be inferred from the results.

Results

The results for the first convolutional layer (conv1) with input dimension changing from 32x32 to 1024x1024 are given in Table . The number of pixels in the image is obtained by squaring the image dimension.

Table 3: Individual layer-wise measurement results for conv1 layer

Dimensions	Number of pixels	CPU/Package_0_energy (mJ)	CPU/Package_1_energy (mJ)	DRAM/DRAM_0_energy (mJ)	DRAM/DRAM_1_energy (mJ)	Total Energy for 10k images (mJ)	Total Energy per image (mJ)
32	1024	430591.86	438895.51	107377.99	108857.91	1085723.27	108.57
64	4096	555363.89	577328.19	155497.25	148678.28	1436867.61	143.69
128	16384	926784.91	969018.01	230732.85	208853.21	2335388.98	233.54
256	65536	2182878.48	2247193.97	506746.77	512962.4	5449781.62	544.98
512	262144	7093129.7	7273590.64	1555559.57	1454149.72	17376429.63	1737.64
1024	1048576	23534665.53	25282982.12	5501520.94	6476860.11	60796028.7	6079.60

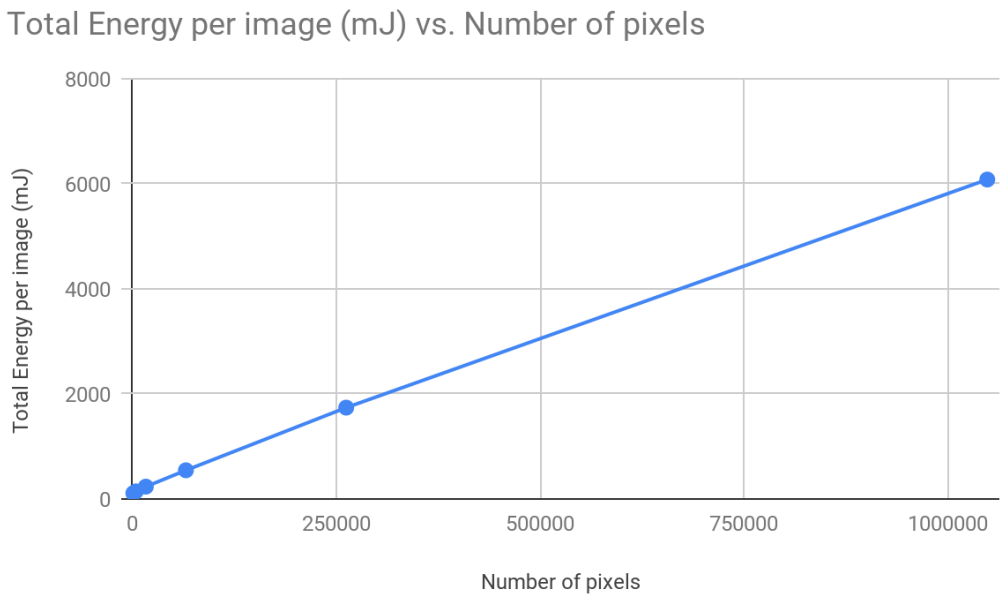

Figure 9: Total Energy per image (mJ) vs. Number of pixels for conv1

Similarly, the experiment is conducted for the first fully connected layer (linear1). Here, one dimension of the input needs to be fixed as per the shape of the layer. In this case, it needs to stay constant at 400. Hence, the number of pixels is obtained by multiplying the image dimension by 400.

Table 3: Individual layer-wise measurement results for linear1 layer

Dimensions	Input Dimension x Output Dimension	CPU/Package_0_energy (mJ)	CPU/Package_1_energy (mJ)	DRAM/DRAM_0_energy (mJ)	DRAM/DRAM_1_energy (mJ)	Total Energy for 10k images (mJ)	Total Energy per image (mJ)
32	12800	657564.82	669120.97	153171.75	127012.21	1606869.75	160.69
64	25600	1010002.75	1037902.4	227673.1	235621.03	2511199.28	251.12
128	51200	1590620.48	1647943.18	360693.54	312618.53	3911875.73	391.19
256	102400	2877407.1	2978843.32	617746.09	606696.23	7080692.74	708.07
512	204800	4912320.19	5035338.56	1035990.23	1051827.45	12035476.43	1203.55
1024	409600	9715623.9	10075784.85	2055656.31	1960552.73	23807617.79	2380.76

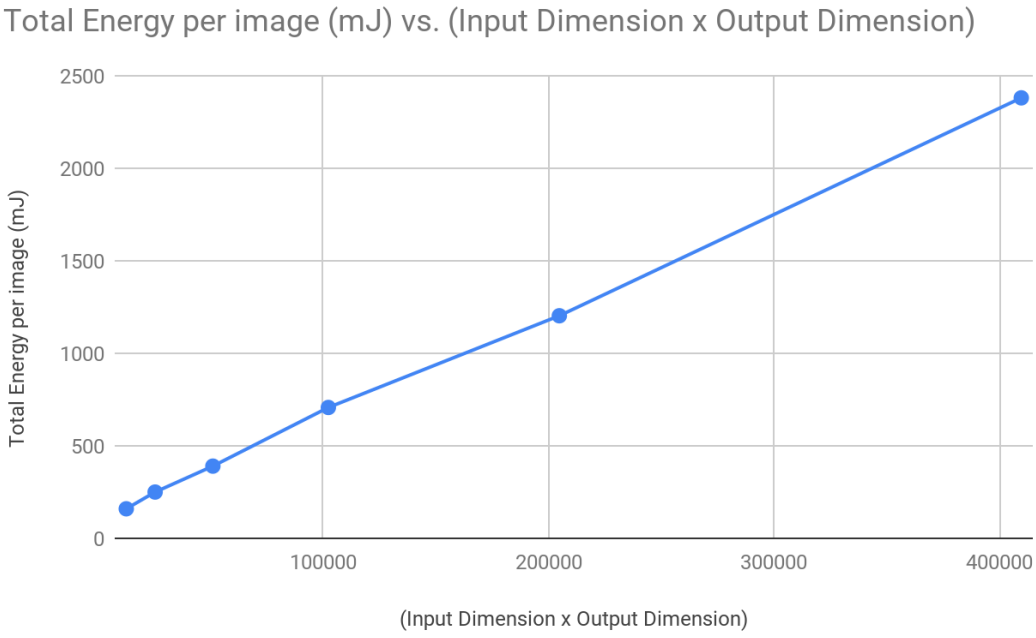

Figure 10: Total Energy per image (mJ) vs. (Input Dimension x Output Dimension) for linear1

To inspect the correlation, linear regression is used to fit a linear model over the resulting energy data for each layer. This gives us the slope, intercept and the coefficient of determination (R^2) for the resulting model.

Table 4: Individual layer-wise measurement results for linear1 layer

Layer	Intercept (mJ)	Slope (mJ/pixel)	Score
conv1	152.17	5.68E-03	0.9995
conv2	94.18	8.13E-03	0.9980
linear1	105.47	5.53E-03	0.9992
linear2	98.57	5.30E-03	0.9997

Observations

From Figure 9 and Figure 10, we can observe the linear dependency between the number of pixels and the energy consumed. This suggests that after offsetting the energy consumed outside of the model inference, the energy consumed per pixel is constant. This constant can be inferred from the value of slope for the corresponding layer given in Table 4. The energy offset, which consists of energy consumed in loading libraries, model initialisation, etc. can be inferred from the intercept value for the respective layer. The coefficient of determination is observed to be high, indicating strong linear correlation between the two variables.

6. Experimental Conclusions

From the conducted experiments, a linear dependency can be observed between the energy consumed and the number of input pixels. Using this, we can define a measure for per-pixel energy consumption of a particular layer in the neural network, and finally for the entire network. To benchmark different hardware architectures for energy consumption, per-pixel energy consumption can be measured for a particular convolutional neural network. With the linear dependency established between the energy consumed and the pixels, the described methodology can be used to develop a tool that can measure energy consumption of individual layers. Also, the linear graph produces an intercept with the Y-axis, which can be used to calculate the energy consumed by the application process when the energy consumed by the neural network is excluded.

7. DeepMeter - A layerwise energy measurement tool for DNNs

DeepMeter is a software tool developed as part of the project, which uses the methodology developed in the aforementioned experiments. It allows the user to inspect the energy consumption of a particular layer in a given neural network. It is built as a Python wrapper around the Intel SoCWatch energy measurement tool to serve the purpose of layerwise measurement.

7.1 Objectives and design requirements

The tool is built with the prospect of releasing it as a library and deploying it as a web service for users to access through a website. This implies two basic requirements on the tool-

1. Availability of a single executable file, which can be run as a process in the deployment server.
2. Easy-to-use input and output data structures that can be easily integrated into web interfaces

Hence, the input to the tool is accepted through a easy-to-use method. The Javascript Object Notation (JSON) format was chosen for this purpose. The user can provide the a JSON file which contains the configuration of the neural network architecture. DeepMeter gives the output in a Comma Separated Values (CSV) file, which contains the information about the energy and power consumption of the selected layer in a tabular format. The CSV file format was chosen due to its easy portability to spreadsheets.

Since the tool is also intended to be a library, the software must be developed keeping in mind the prospective development of an Application Programming Interface (API). Using the DeepMeter API, developers can access the energy consumption data. Hence, it is developed as an installable Python package, that can be imported into the files of the user program. This entails exposing the relevant classes and functions to the user. This is, again, enabled by the modular structure of the codebase.

7.2 Programming philosophy and structure

Keeping in mind maintainability and clarity of the code, the codebase was developed in a modular structure. The code operates on the principle of "separation of concern", which states that independent functions should exist separately, ideally in separate modules (folders in the case of Python). Figure 10 shows the structure of the codebase in a tree format-

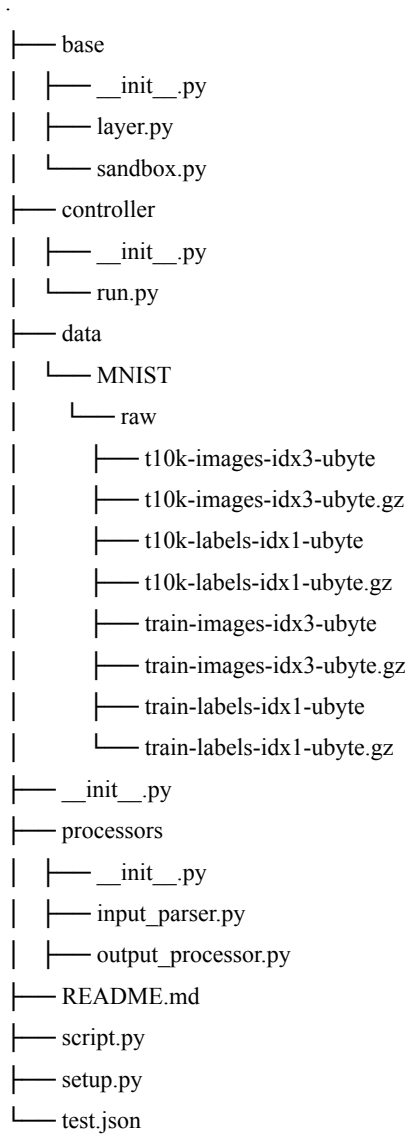

Figure 10: Codebase structure

The codebase is divided into separate module- base, controller and processors. These have been divided according to their functional differences. The base module contains the classes and functions that are used throughout the library, which make up the core functionality of the tool. The controller module contains the code required to run the simulation of the different layers of the model within the tool. The processors module handles parsing the input and extracting the required data from SoCWatch and formatting the final output in the CSV format. The modular structure simplifies debugging and helps in easy maintenance of the codebase. It also ensures that features can be added in the future without changing the existing code significantly. The modular structure also allows different contributors to the tool can access and modify the code at the same time without interfering with someone else's changes.

The entrypoint to the tool is a single Python script- "script.py", which accepts the path to the JSON file that describes the model and the selected layer as command line arguments. If the user if using the tool through the command line, this file can be run by passing the file to the Python runtime.

Currently, the tool has inherited the modified code of the experiments conducted to establish the energy measurement methodology. Hence, the data folder contains the MNIST dataset, which was previously used for testing with the Lenet-5 model. MNIST is a low-complexity data collection of handwritten digits used to train and test various supervised machine learning algorithms. The database contains 70,000 28x28 black and white images representing the digits zero through nine. The data is split into two subsets, with 60,000 images belonging to the training set and 10,000 images belonging to the testing set. DeepMeter currently operates on the MNIST dataset, since the usage is predominantly restricted to Convolutional Neural Networks (CNNs). Option to select a particular dataset can be added in future versions of the tool. At installation, the dataset is not present in the folder. This is due to the large size of the dataset making the tool heavy during installation. At the first run of the tool, the MNIST dataset is downloaded from the internet. Thus, the results obtained from the first run of the tool may be unreliable.

7.3 Dependencies

1. PyTorch

PyTorch is an open source machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR). It is free and open-source software released under the Modified BSD license. This project uses the PyTorch Python library for the purpose of building the deep neural network provided by the user.
PyTorch provides the neural network module (nn), which make functions for different layers available. These include convolutional layers, fully-connected layers, pooling layers, etc. While the layers are provided by PyTorch, DeepMeter implements a single wrapper for different layers. This is a unified layer of abstraction that is built for customised layer selection.
The masic model structure can be seen in Figure 5. A model is a class that inherits the "nn.module" class from PyTorch. It inherits the different functions that are required in the functioning of a DNN, for example- forward pass and backpropogation of the model. The layers are defined in the initialising functions by declaring a series of class attributes obtained by calling the respective PyTorch library functions. The "forward" functions defines the series of steps that are involves in the forward pass of the model. When the model is called, the "forward" function runs and returns the result.

2. Intel SoCWatch

Intel SoCWatch is a data collector for power-related data from the system. It is intended to identify usage issues that may lead to excessive power consumption. Besides energy metrics, the metrics collected include-

- System sleep states
- CPU and GPU sleep states
- Processor frequencies
- Temperature data
- Device sleep states

DeepMeter requires Intel SoCWatch to be installed on the system. Installation instructions can be found on Intel's website[2]. It also requires the user to compile the required kernel modules and enable certain kernel permissions so that the tool can access the CPU registers that are normally not accessible in user space.

7.4 Modules

The codebase is divided into separate module- base, controller and processors. These have been divided according to their functional differences. The base module contains the classes and functions that are used throughout the library, which make up the core functionality of the tool. The controller module contains the code required to run the simulation of the different layers of the model within the tool. The processors module handles parsing the input and extracting the required data from SoCWatch and formatting the final output in the CSV format.

1. Base

The "layer.py" contains the "Layer" class. This is an abstraction over the layer modules in PyTorch.

```
class Layer:
    def __init__(self, kwargs) -> None:
        self.params = kwargs

        self.type_ = kwargs.get("type")
        if not self.type_:
            raise Exception("'type' key missing. Please specify type of action")
        if self.type_ == "layer":
            torch_module = kwargs.get("torch_module")
            if torch_module:
                self.layer_module_name = torch_module.split(".")[-1]
            else:
                raise Exception("Required parameter: torch_module")
            self.activation_functions = ["relu", "tanh", "selu", "leakyrelu"]
            self.layer_module = getattr(nn, self.layer_module_name)
            self.layer_object = self.__get_layer()
            self.reshape_dim = kwargs.get("reshape_dim")
```

[2] Intel® SoC Watch

```
    elif self.type_ == "reshape":
        self.layer_object = self.__buffer()
        self.reshape_dim = kwargs.get("reshape_dim")

def __buffer(x):
    return x

def __get_layer(self):
    try:
        if self.layer_module_name.lower() in self.activation_functions:
            return self.layer_module()
        elif "linear" in self.layer_module_name.lower():
            return self.layer_module(in_features=self.params['in_features'], out_features=self.params['out_features'])
        elif "pool" in self.layer_module_name.lower():
            return self.layer_module(kernel_size=self.params['kernel_size'], stride=self.params['stride'])
        elif "conv" in self.layer_module_name.lower():
            return self.layer_module(
                in_channels=self.params['in_channels'],
                out_channels=self.params['out_channels'],
                kernel_size=self.params['kernel_size'],
                stride=self.params['stride'],
                padding=self.params['padding']
            )
    except KeyError as e:
        raise Exception(f"Missing parameter: {e.args[0]}")
    except:
        raise

def __str__(self) -> str:
    return str(self.layer_object)
```

Figure 11: Layer class

The Layer class selects the required PyTorch module from the input module, which is in the form of a string. It is a necessary abstraction over the PyTorch modules since the modules need to be loaded fron the text input given by the user and eventually, they need to be run in the sandbox model, whose run shall be monitored by SoCWatch.

```
class SandboxModel(nn.Module):
    def __init__(self, layers:List[Layer], layerwise=False) -> None:
        super().__init__()
        self.layers = layers
        self.layerwise = layerwise
        if self.layers:
            self.active_layer = None
            self.__layer_iter = iter(self.layers)
        else:
            raise Exception("Layers not provided!")

    def __iter__(self):
        self.active_layer = self.layers[0]
        return self

    def __next__(self):
        curr = self.active_layer
        if not curr:
            raise StopIteration
        self.active_layer = next(self.__layer_iter)
        return curr

    def __call(self, x, func, reshape_dim=None):
        if reshape_dim:
            x = resize(x, [reshape_dim[2], reshape_dim[3]])
            x = x.expand(reshape_dim)
        x = func(x)
        return x

    def set_active_layer(self, layer_index):
        self.active_layer = self.layers[layer_index]

    def forward(self, x):
        x_ = x
        if not self.layerwise:
            for layer in self.layers:
                x_ = self.__call(x_, layer.layer_object, layer.reshape_dim)
        else:
            x_ = self.__call(x_, self.active_layer.layer_object, self.active_layer.reshape_dim)
```

Figure 12: SandBoxModel class

The SandboxModel class encapsulates the functionality of a model created from PyTorch layers, while allowing for changing the layers dynamically. Thus, the model can be constructed as per the requirement of the user. It has the option of running multiple layers together at a time or one layer at a time. This option can be selected by the calling function. Through the "__iter__" function the class can be converted into an iterator. SandboxModel uses a custom function, "__call" to call the layers, where it resizes the input vector as per the layer. The resize dimension is provided by the user in the input. Thus, SandboxModel acts as a bridge between the PyTorch nn module and the functionality of DeepMeter.

2. Controller

DeepMeter applies the principle of metaprogramming, where it writes a program within itself to simulate the running of the deep neural network. The "run.py" file in the controller module contains the template code to construct such a process. It brings together all the internal components of DeepMeter. The "run.py" file also provides options for the input file path, the batch size and the selected layer for which energy has to be measured. The "script.py" file, which is the entrypoint calls the "run.py" file with the options given by the user.

3. Processor

The processor module contains the code required for text processing in input and output functions of DeepMeter.

The "input_parser.py" file contains the InputInterface class, which is processes the text input file given by the user in JSON format. A sample input to DeepMeter is given in Figure 13

```
[
  {
    "type": "layer",
    "torch_module": "nn.Conv2d",
    "in_channels": 1,
    "out_channels": 6,
    "kernel_size": [2, 2],
    "stride": [1, 1],
    "padding": [0, 0]
  },
```

```
    {
      "type": "layer",
      "torch_module": "nn.ReLU"
    },
    {
      "type": "layer",
      "torch_module": "nn.MaxPool2d",
      "kernel_size": [2, 2],
      "stride": [2, 2]
    },
    {
      "type": "layer",
      "torch_module": "nn.Conv2d",
      "in_channels": 6,
      "out_channels": 16,
      "kernel_size": [5, 5],
      "stride": [1, 1],
      "padding": [0, 0]
    },
    {
      "type": "layer",
      "torch_module": "nn.ReLU"
    },
    {
      "type": "layer",
      "torch_module": "nn.MaxPool2d",
      "kernel_size": [2, 2],
      "stride": [2, 2]
    }
]
```

Figure 13: Sample input to DeepMeter

The input is in hte format of a list of objects, each of which contains the required specifications to construct a layer in PyTorch. These parameters differ according to the layer chosen. The user can also enter objects with type “resize”, which instructs DeepMeter to resize the feature map before moving on to the next step. This step is often required since the feature map dimensions taken from the dataset might not match the dimensions required by the layer at its input. A

“Layer” object is constructed from each object in the JSON list. These “Layer” objects are collected in a list and passed further to the “run.py” file.

The “output_processor.py” file contains the code required to format the output from SoCWatch into a proper format. It takes the text output from the standard output (STDOUT) of the process running SoCWatch and parses it to get the required values. Then, the extracted values are written to a CSV file. A sample output is given in Figure 14.

```
CPU/Package_0_power,CPU/Package_0_energy,CPU/Package_1_power,CPU/Package_1_energy,DRAM/DRAM_0_po
wer,DRAM/DRAM_0_energy,DRAM/DRAM_1_power,DRAM/DRAM_1_energy
30131.32,23626.1,33905.2,26585.45,8636.69,6772.16,10831.17,8492.92
```

Figure 14: Sample output from DeepMeter

The first line in the output is the header line, which is followed by the values of the energy and power measured by the tool for the selected layer in each CPU and DRAM package.

8. Future scope

DeepMeter is highly dependent on the Intel SoCWatch tool. Since Intel SoCWatch tool itself is in its nascent stage, further improvements and feature additions can be made based on how Intel releases further versions of SoCWatch. Documentation for the functionalities of SoCWatch scant at this point of time. Due to limited use, user answers on internet forums like StackOverflow related to bugs and features in SoCWatch are nearly non-existent. This makes development of DeepMeter difficult, since SoCWatch essentially becomes a black box. As usage of the tool picks up, bugs are fixed and new features are added, the development of DeepMeter will become relatively smooth.

The development of DeepMeter in this project has been foundational and has focused on building its functional core. Further development can be done on top of the existing code-

1. Option to select multiple layers for measurement
2. Option to select specific metrics to extract
3. Development of a web-based service that can take user input in the form of a UI or a JSON file and can output the energy measurement metrics for a selected hardware architecture and neural network
4. Feature to generate graphs and analysis reports from the collected data
5. Selection of datasets
6. Collection of common models that the user can select directly without needing to provide a configuration file

9. References

T.-J. Yang, Y.-H. Chen, J. Emer, V. Sze, "A Method to Estimate the Energy Consumption of Deep Neural Networks," Asilomar Conference on Signals, Systems and Computers, Invited Paper, 2017.

Y. Chen, J. Emer, and V. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks,” in ISCA, 2016.