Using Hardware Events to Predict the Architecture of Common Deep Neural Networks

Daniel Loran
Dept. of Electrical
and Computer Engineering
University of California, Davis
email: dcloran@ucdavis.edu

Chih-Ho Hsu
Dept. of Electrical
and Computer Engineering
University of California, Davis
email: lchhsu@ucdavis.edu

Tobin Joseph
Dept. of Electrical
and Computer Engineering
University of California, Davis
email: tkjoseph@ucdavis.edu

***Abstract*—In recent years deep neural networks have been used to help solve increasingly complex and sensitive problems. Other research in the security space have focused on determining the architecture of a 'black box' model. In light of this in this paper we describe the process in which we capture hardware event data from running stock DNN architectures from the keras library and use that hardware event data to predict which architecture the DNN in question uses. As a result of our work, we were able to create a classifier that can predict a given architecture with a high degree of accuracy. Our code can be viewed in our git repo: https://github.com/DanLoran/DNN-fingerprint**

***Index Terms*—deep neural networks, hardware performance events, privacy, reverse engineering, machine learning**

I. Introduction

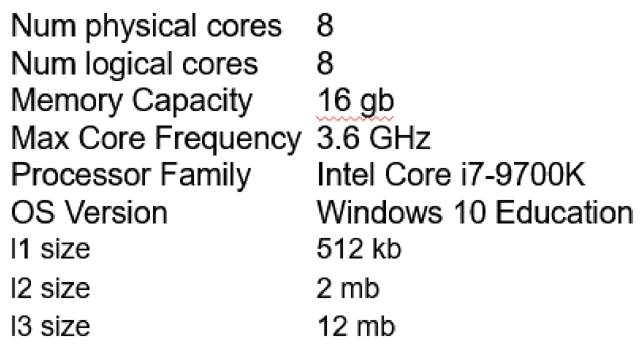
The proliferation of Deep Neural Networks (DNN) have created a new target for cyber-criminal activity through the use of microarchitectural attacks [1]. Previous recent studies have attempted to identify possible attack techniques by trying to identify the underlying DNN layer architecture of systems or rely on previous knowledge of the DNN type and architecture [1]. For our work we used Intel® VTune™ Profiler software [2] to analyze DNN written in python as the Intel® VTune™ Profiler provides support to analyze python programs [3]. Due to technical difficulties with collecting and parsing time-stamped hardware data, we ended up using aggregate hardware data and predicting not the layers of a DNN, but the actual DNN architecture.

In summary, the goal of this project has now become to infer a common DNN architecture given the corresponding aggregated hardware events in real-time. The following content is organized as follows. Section II presents our methodology for collecting data, and details failed attempts to collect timeline data and preprocess it. Section III shows and discusses the results from the evaluation sets. Section IV summarizes this work and give a few potential research direction and challenges.

II. Methodology

A. System Specifications

This experiment was conducted on a Intel Core i7-9700K CPU @ 3.60GHz. What follows is a table of more detailed specifications used to obtain our results:



Num physical cores	8
Num logical cores	8
Memory Capacity	16 gb
Max Core Frequency	3.6 GHz
Processor Family	Intel Core i7-9700K
OS Version	Windows 10 Education
l1 size	512 kb
l2 size	2 mb
l3 size	12 mb

Fig. 1. System specs used to capture hardware data [?]

B. Data collection

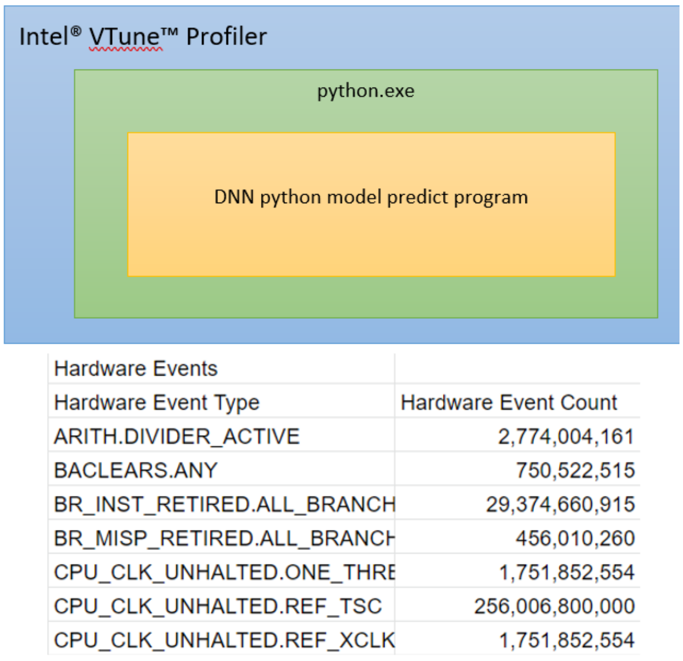
Our methodology initially tried to follow the example set by Dandpati Kumar Bhargav Achary, et. al. [1] to try and identify DNN model layers through micro-architectural hardware events analysis performed in a black box approach using the Intel® VTune™ Profiler software [2]. We intended to use DNN built in python, as there are various systems created in this interpreter.

Our first approach, influenced by Achary, et. al. [1], we tried to create python code to insert between the model layers execution to measure the counts of different hardware events provided by the Micro-architecture Analysis performed with the Intel® VTune™ Profiler.

Our second approach, consisted of using the the Intel® VTune™ Profiler feature to time-slice a previously performed Micro-architecture Analysis into result files for each millisecond of execution using the time-filter option [4]. This required to first run the Intel® VTune™ Profiler for a DNN and after we obtained the results data, create a batch file to execute one Intel® VTune™ Profiler commend for each millisecond of execution.

Our third approach uses a black box strategy by getting the cumulative results of the Micro-architectural Analysis performed by the Intel® VTune™ Profiler on runs for the different DNN used in our work. For each DNN, we executed a python script that ran 100 iterations of the DNN and did VTune analysis on this script. This gives us access to the cumulative hardware events like, for example, branch mispredicts, as well as more general data like runtime and cache hits. As we parsed this data, we realized that the more specific hardware event data yielded better results from the classifier so we filtered down to the most useful hardware event data and used this for the classifier.

We would use these cumulative results as our raw data to feed a DNN classification system to try and identify the type of DNN being used on the program being analyzed. We take programs that try to use the DNN prediction functions, as they would be used by a trained model with weights.



Hardware Events	
Hardware Event Type	Hardware Event Count
ARITH.DIVIDER_ACTIVE	2,774,004,161
BACLEARS.ANY	750,522,515
BR_INST_RETIRED.ALL_BRANCH	29,374,660,915
BR_MISP_RETIRED.ALL_BRANCH	456,010,260
CPU_CLK_UNHALTED.ONE_THRE	1,751,852,554
CPU_CLK_UNHALTED.REF_TSC	256,006,800,000
CPU_CLK_UNHALTED.REF_XCLK	1,751,852,554

Fig. 2. Intel® VTune™ Profiler capturing hardware events counts through a Micro-architecture Analysis from a DNN running in python [?]

C. Data Preprocessing

Following Achary, et. al. [1] data clean up methodology, we planned on using data normalization [5], as defined by the formula:

$$x_{normalized} = \frac{x - x_{minimum}}{R(x)} \quad (1)$$

where $R(x)$ denote the range of x.

As our measures can have magnitudes of difference between different variables, we scale them to a common scale of 0 to 1 to statistically make these comparable,

Also, we planned of using the SMOTE technique for unbalanced samples [6], and the Savitzky Golay filter [7] to eliminate noise and spikes in the data.

III. Results

A. Data Collection

For our first approach at data collection, we intended to use the DNN source code to insert our code to help gather micro-architectural hardware events data. Unfortunately, the source code we had available for our work did not include the DNN model source, so we could not insert code between layers.

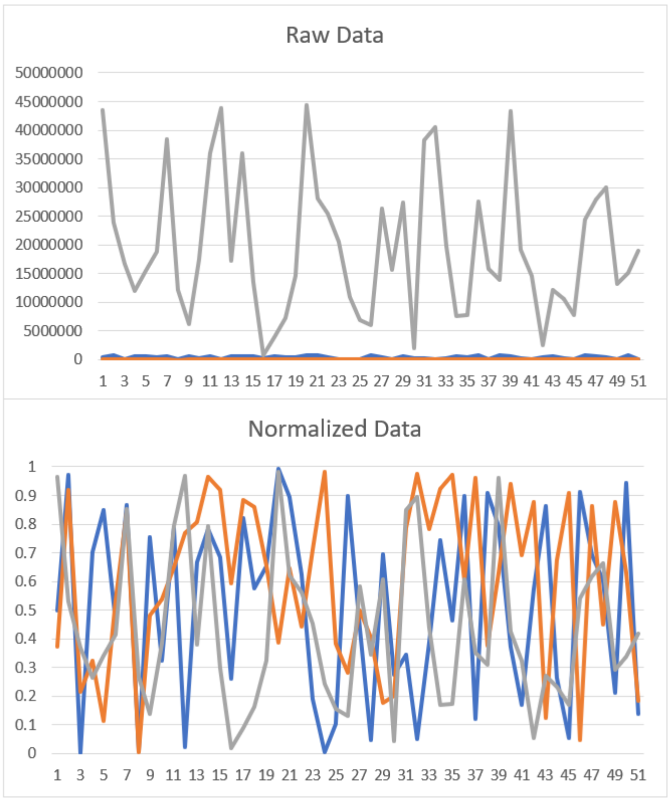
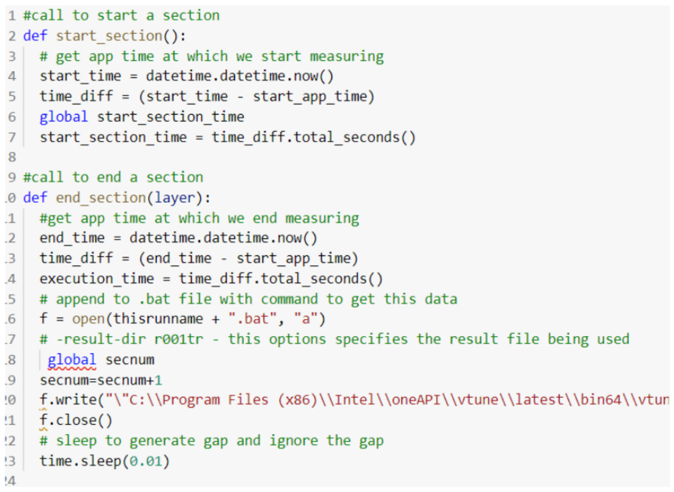


Fig. 3. Example of Data Normalization [?]

We tried doing a loop running the model.predict function and measuring it, but we had pre processing and post processing code we could not ignore its execution for the analysis, so we had to discard this.

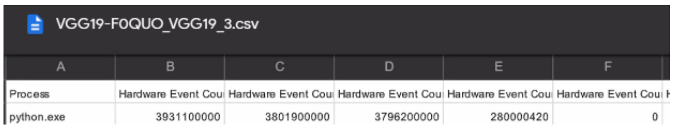


```
#call to start a section
def start_section():
    # get app time at which we start measuring
    start_time = datetime.datetime.now()
    time_diff = (start_time - start_app_time)
    global start_section_time
    start_section_time = time_diff.total_seconds()

#call to end a section
def end_section(layer):
    #get app time at which we end measuring
    end_time = datetime.datetime.now()
    time_diff = (end_time - start_app_time)
    execution_time = time_diff.total_seconds()
    # append to .bat file with command to get this data
    f = open(thisrunname + ".bat", "a")
    # -result-dir r001tr - this options specifies the result file being used
     global secnum
    secnum=secnum+1
    f.write("\"C:\\Program Files (x86)\\Intel\\oneAPI\\vtune\\latest\\bin64\\vtun
    f.close()
    # sleep to generate gap and ignore the gap
    time.sleep(0.01)
```

Fig. 4. Functions to call to mark the start and end of a section in code that we want the hardware events measured for.

For our second approach, partitioning the analysis data into 1 ms time slices, we analyzed the required resources to conduct such experimentation, and concluded that we lacked the hardware resources to run such scenario as, for example, the

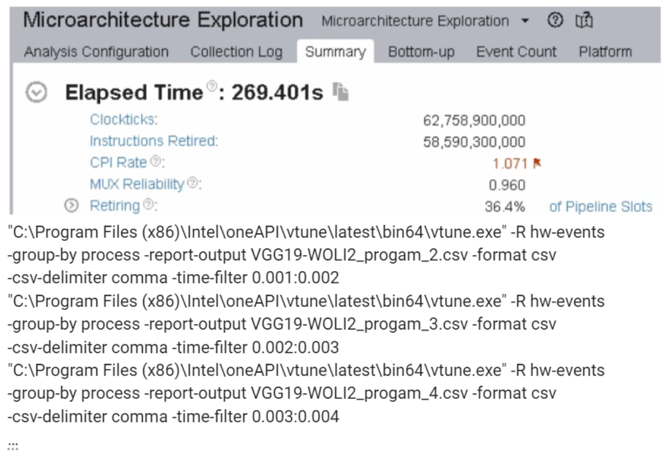


VGG19-F0QUO_VGG19_3.csv

A	B	C	D	E	F
Process	Hardware Event Cou	Hardware Event Cou	Hardware Event Cou	Hardware Event Cou	Hardware Event Cou
python.exe	3931100000	3801900000	3796200000	280000420	0

Fig. 5. Example of raw data collected for a section of code

analysis of just one run of a DNN would require the execution of around 270,000 Intel® VTune™ Profiler executions. The time required to gather the data on the systems we possess would have required more than six months. This time frame was out of the scope of this work.



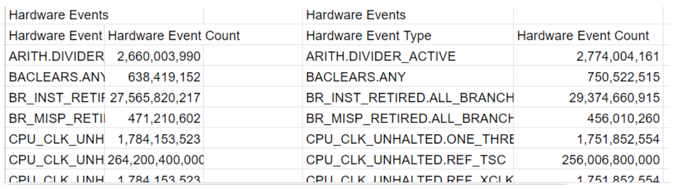
Elapsed Time: 269.401s

Clockticks:	62,758,900,000	
Instructions Retired:	58,590,300,000	
CPI Rate:	1.071	
MUX Reliability:	0.960	
Retiring:	36.4%	of Pipeline Slots

```
"C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\vtune.exe" -R hw-events
-group-by process -report-output VGG19-WOLI2_progam_2.csv -format csv
-csv-delimiter comma -time-filter 0.001:0.002
"C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\vtune.exe" -R hw-events
-group-by process -report-output VGG19-WOLI2_progam_3.csv -format csv
-csv-delimiter comma -time-filter 0.002:0.003
"C:\Program Files (x86)\Intel\oneAPI\vtune\latest\bin64\vtune.exe" -R hw-events
-group-by process -report-output VGG19-WOLI2_progam_4.csv -format csv
-csv-delimiter comma -time-filter 0.003:0.004
:::
```

Fig. 6. Micro-architecture Analysis results for a VGG191 DNN and part of the resulting batch file with the Intel® VTune™ comand to slice it into 1ms results.

Our third approach to data collection was the one that we finally used for our work. We ran the Micro-architecture Analysis with the Intel® VTune™ Profiler and put together the data on a Google Sheets document for storage and further processing.

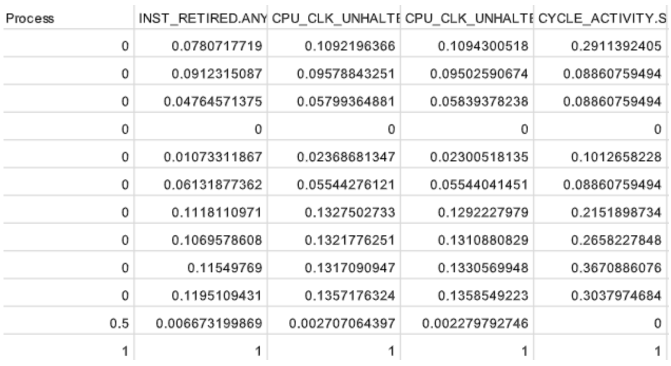


Hardware Events			Hardware Events	
Hardware Event	Hardware Event Count		Hardware Event Type	Hardware Event Count
ARITH.DIVIDER	2,660,003,990		ARITH.DIVIDER_ACTIVE	2,774,004,161
BACLEARS.ANY	638,419,152		BACLEARS.ANY	750,522,515
BR_INST_RETII	27,565,820,217		BR_INST_RETIRED.ALL_BRANCH	29,374,660,915
BR_MISP_RETI	471,210,602		BR_MISP_RETIRED.ALL_BRANCH	456,010,260
CPU_CLK_UNH	1,784,153,523		CPU_CLK_UNHALTED.ONE_THRE	1,751,852,554
CPU_CLK_UNH	264,200,400,000		CPU_CLK_UNHALTED.REF_TSC	256,006,800,000
CPU_CLK_UNH	1,784,153,523		CPU_CLK_UNHALTED.REF_XCLK	1,751,852,554

Fig. 7. Micro-architecture Analysis results for a VGG191 DNN and part of the resulting batch file with the Intel® VTune™ comand to slice it into 1ms results.

B. Data Pre-Processing

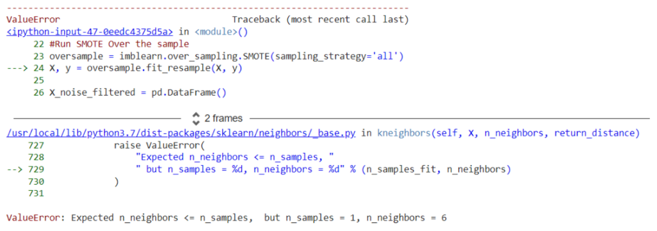
We used the data normalization formula on our collected data, resulting in normalized values from 0 to 1. We do this because our measures are hardware events counts that some can come even into the millions and others into the thousands. To statistically make these comparable, we scale them to a common scale of 0 to 1. This helps to accelerate the training of the DNN [5] that we use in this work.



Process	INST_RETIRED.ANY	CPU_CLK_UNHALTE	CPU_CLK_UNHALTE	CYCLE_ACTIVITY.S
0	0.0780717719	0.1092196366	0.1094300518	0.2911392405
0	0.0912315087	0.09578843251	0.09502590674	0.08860759494
0	0.04764571375	0.05799364881	0.05839378238	0.08860759494
0	0	0	0	0
0	0.01073311867	0.02368681347	0.02300518135	0.1012658228
0	0.06131877362	0.05544276121	0.05544041451	0.08860759494
0	0.1118110971	0.1327502733	0.1292227979	0.2151898734
0	0.1069578608	0.1321776251	0.1310880829	0.2658227848
0	0.11549769	0.1317090947	0.1330569948	0.3670886076
0	0.1195109431	0.1357176324	0.1358549223	0.3037974684
0.5	0.006673199869	0.002707064397	0.002279792746	0
1	1	1	1	1

Fig. 8. Data Normalized (Sample) [?]

When we ran the SMOTE techniques [6] software that we prepared, we ran into the problem that our sample sizes where too small to be able to run this type of technique. So, unfortunately, this technique was not applied to our training and testing data.



```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-47-0eedc4375d5a> in <module>()
     22 #Run SMOTE Over the sample
     23 oversample = imblearn.over_sampling.SMOTE(sampling_strategy='all')
---> 24 X, y = oversample.fit_resample(X, y)
     25 
     26 X_noise_filtered = pd.DataFrame()

2 frames
/usr/local/lib/python3.7/dist-packages/sklearn/neighbors/_base.py in kneighbors(self, X, n_neighbors, return_distance)
    727             raise ValueError(
    728                 "Expected n_neighbors <= n_samples, "
--> 729                 " but n_samples = %d, n_neighbors = %d" % (n_samples_fit, n_neighbors)
    730             )
    731 

ValueError: Expected n_neighbors <= n_samples,  but n_samples = 1, n_neighbors = 6
```

Fig. 9. SMOT error we encountered because of small sample.) [?]

Achary, et. al. [1] used Savitzky Golay filter [7] to eliminate noise and spikes in the time sliced data they used in their work. Since, we ended up using cumulative data of hardware events for an entire run of the DNN, we saw no need to use the software we had prepared for this. In a future work, if the VTune time-filter option [4] is used, then this technique would definitively be of use.

C. Classifier Design

In this subsection, we will discuss the way we design and construct our classifier for DNN prediction based on the collected hardware event.

To begin with, thanks to the effectiveness and efficiency on general classification problem that is brought by recent advancements on the deep learning technique, one can obtain high accuracy on the classification task with less run-time complexity as long as sufficient training data is provided. These advantages motivate us to adopt the DNN technique to address our architecture inferring problem. The design of our DNN is summarized as follows: first, we construct a DNN with two hidden layers. Each hidden layer contains 32 neurons with a Relu activation function. The mathematical form of Relu can be present as follow:

$$f(x) = \max(x, 0) \tag{2}$$

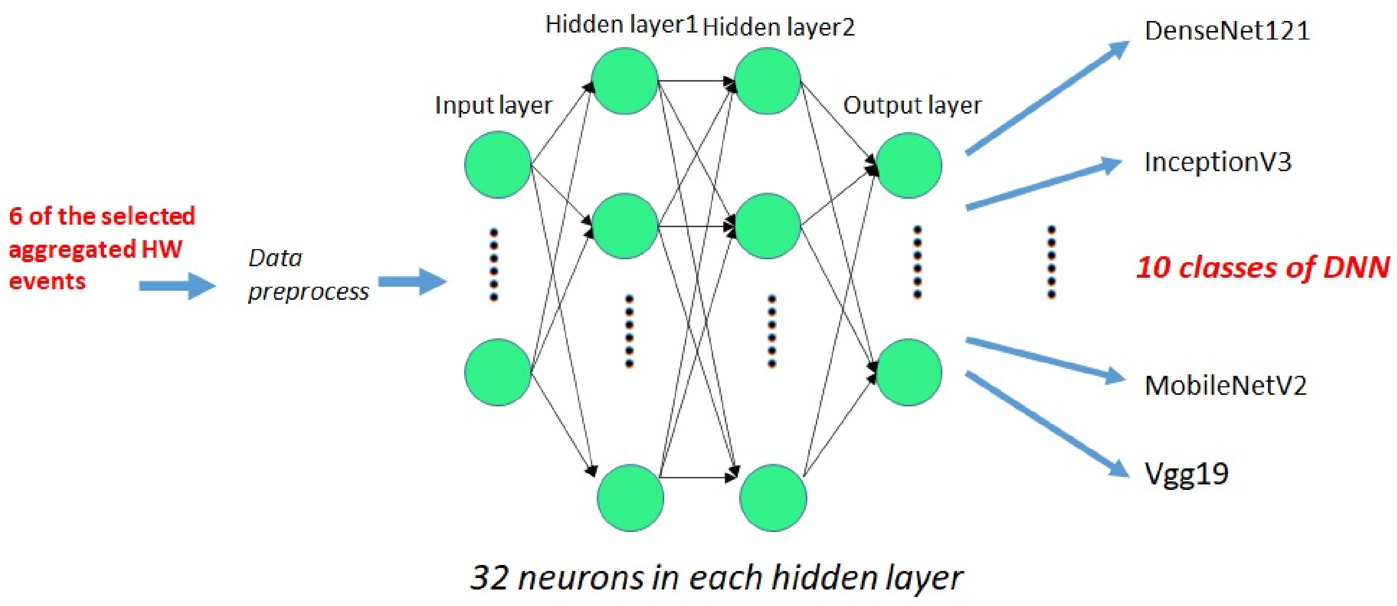


Fig. 10. The DNN framework for inferring DNN architecture based on the hardware event count

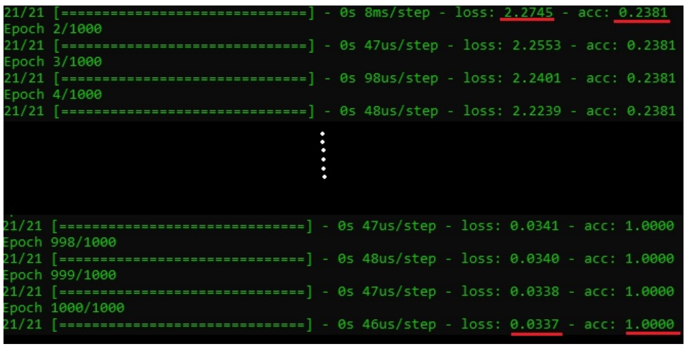
which means the output of a Relu activation is the maximum of the input value and zero, which makes sure the output of a neuron is non-negative. In the last layer, we introduce a flatten layer with the activation function of softmax so that it can perform the classification task. The input of our DNN contains six neurons that correspond to six hardware event matrices we selected:

BR INST RETIRED.ALL BRANCHES
CPU CLK UNHALTED.THREAD
CYCLE ACTIVITY.STALLS MEM ANY
FRONTEND RETIRED.ANY DSB MISS
INST RETIRED.ANY
MEM INST RETIRED.ANY

On the other hand, the input of our DNN contains ten neurons that correspond to ten DNNs we try to classified into (i.e. DenseNet 121, DenseNet 169, DenseNet 201, Inception ResNet, Inception V3, MobileNet v1, Xception, Vgg 16 and Vgg 19 respectively). The first neuron represent DenseNet 121 and the second one represent DenseNet 169 and so on and so forth. Finally, we use categorical-crossentropy as our loss function and adopt Adam as the optimizer. The proposed DNN framework for inferring DNN architecture based on the hardware event count is illustrated in Fig. 10.

D. Classifier Performance

In this section, we present the performance of our designed classifier. To begin with, our dataset contains 50 data points and each of them has seven values, which are the six hardware matrics and one label respectively. We further split them into 40 for training and 10 for testing. We then trained our DNN for 1000 with batch size 40.



```
21/21 [==============================] - 0s 8ms/step - loss: 2.2745 - acc: 0.2381
Epoch 2/1000
21/21 [==============================] - 0s 47us/step - loss: 2.2553 - acc: 0.2381
Epoch 3/1000
21/21 [==============================] - 0s 98us/step - loss: 2.2401 - acc: 0.2381
Epoch 4/1000
21/21 [==============================] - 0s 48us/step - loss: 2.2239 - acc: 0.2381
                                     ⋮
21/21 [==============================] - 0s 47us/step - loss: 0.0341 - acc: 1.0000
Epoch 998/1000
21/21 [==============================] - 0s 48us/step - loss: 0.0340 - acc: 1.0000
Epoch 999/1000
21/21 [==============================] - 0s 47us/step - loss: 0.0338 - acc: 1.0000
Epoch 1000/1000
21/21 [==============================] - 0s 46us/step - loss: 0.0337 - acc: 1.0000
```

Fig. 11. The Training process and accuracy performance on the training data of our implemented DNN

We first present the accuracy of our implemented DNN on the training data. From Fig. 11 we can see that at the beginning the loss value is as high as 2.2745 and the accuracy on training data is only 0.2381. When the training is complete, the loss value is reduced to 0.0337 with 100% accuracy on the training set. This result prove us an insight that our implemented DNN indeed can learn the mapping from the hardware events count to the corresponding DNN architecture.

Next, we evaluate the performance on the 10 testing data, which has the ground-truth labels of DenseNet 121, DenseNet 169, ..., Vgg 19 and is in the same order as the output neuron we discussed above. The result is presented in Table I, where the label column represents the ground truth class of the input testing data, the column Output1 to output9 means the output value of our DNN given the testing data with the corresponding input, which may imply the probability that the given testing belong to this class, and the prediction column denotes the decision of classification, which is obtained by picking up the output column with the maximum value. Also, notice that the bold text denotes the decision of the classification and the red text emphasizes the wrong prediction. We can observe that our DNN model achieve 90% accuracy on the testing data of this task.

An interesting area our classifier struggles in is correctly predicting DenseNet169. We can attribute this to how it is between DenseNet 121 and 201 in terms of size, which might make it difficult for the classifier to differentiate since the hardware events when running the two DNN models would look very similar. This could reveal a potential weakness in our approach of attempting to predict the entire model's architecture instead of layer by layer - if two DNNs are very similar in structure, our classifier will have a hard time differentiating. However, across highly differentiable architectures, our classifier boasts a very high accuracy.

IV. Conclusions and Future Works

Using hardware event count to predict type of architecture is very feasible, we were able to achieve a high accuracy on our selection of models with a relatively small dataset. We infer from our results that it should be very reasonable to expect that if a user turns a well known DNN model into a black box, it should be relatively easy for a third party with access to profiling data to determine the architecture of the DNN model.

It should be noted that while a lot of the other research in this field has to do with predicting individual layers, ours predicts known models. This means that our approach might be less successful in identifying a custom designed architecture - this merits further testing.

With more time to generate more data across more DNN architectures and hardware configurations, we could build a more generalized DNN fingerprinting framework. If such a framework was tested on a wide range of existing DNN architectures, it would not be unreasonable to have a classifier that instead of assigning a known configuration, instead attempts to guess at an unknown custom architecture based on training on a wide array of known architectures. This would require a very large dataset.

Furthermore, in testing on different hardware configurations, it would behoove future research to take place on server hardware, which is more likely to lend itself to a real life scenario. Although one should expect different hardware to produce different execution times and hardware events, using normalization in the data can help the prediction even on platforms we have not tested on.

Finally, we arbitrarily selected the hardware events to use for the classifier. A future research area could be to apply some form of automatic feature selection to the available data to see if there are more accurate sets of input data we could feed to the classifier. Additionally, the classifier itself could be further tuned to see if different classifier architectures would yield better results.

Acknowledgment

The project is done for EEC 270, a class taught by Professor Houman Homayoun who is currently an Associate Professor in the Department of Electrical and Computer Engineering at the University of California, Davis.

References

[1] B. A. D. Kumar, S. C. Teja R, S. Mittal, B. Panda, and C. K. Mohan, "Inferring dnn layer-types through a hardware performance counters based side channel attack," Association for Computing Machinery: The First International Conference on AI-MLSystems, vol. AIMLSystems 2021, p. 7, 2021. [Online]. Available: https://doi.org/10.1145/3486001.348622
[2] "Intel® vtune™ profiler," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtuneprofiler.htmlgs.tvh0oz
[3] "Intel® vtune™ profiler user guide: time-filter," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation /vtunehelp/top/analyze-performance /code-profiling-scenarios/python-codeanalysis.html
[4] "Intel® vtune™ profiler user guide: Python code analysis," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop /documentation/vtunehelp/top/command-line-interface/ command-line-interface-reference/timefilter.html
[5] S. C. S. Ioffe, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," Cornell University.
[6] L. O. H. Nitesh V. Chawla, Kevin W. Bowyer and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique."
[7] R. W. Schafer, "What is a savitzky-golay filter? [lecture notes].
[8] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
[9] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
[10] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
[11] K. Elissa, "Title of paper if known," unpublished.
[12] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.
[13] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
[14] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

TABLE I
PERFORMANCE OF THE DESIGNED CLASSIFIER ON THE TESTING DATA.

Label	Output1	Output2	Output3	Output4	Output5	Output6	Output7	Output8	Output9	Prediction
DenseNet 121	**0.9818**	0.0153	0.0027	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	DenseNet 121
DenseNet 169	**0.7152**	0.278	0.0029	0.0000	0.0005	0.0034	0.0000	0.0000	0.0000	DenseNet 121
DenseNet 201	0.0000	0.1464	**0.8509**	0.0002	0.0007	0.0000	0.0000	0.0000	0.0000	DenseNet 201
Inception ResNet	0.0000	0.0000	0.0000	**0.9868**	0.0004	0.0000	0.0084	0.0044	0.0000	Inception ResNet
Inception V3	0.0000	0.0074	0.0067	0.0032	**0.9694**	0.0000	0.0132	0.0000	0.0000	Inception V3
MobileNet v1	0.0014	0.0000	0.0000	0.0000	0.0000	**0.9986**	0.0000	0.0000	0.0000	MobileNet v1
Xception	0.0000	0.0000	0.0000	0.0152	0.0179	0.0000	**0.9669**	0.0000	0.0000	Xception
Vgg 16	0.0000	0.0000	0.0000	0.0196	0.0000	0.0000	0.0000	**0.9741**	0.0063	Vgg 19
Vgg 19	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0338	**0.9662**	Vgg 19