Faculty of Engineering Sciences
Heidelberg University

Master Thesis
in Computer Engineering
submitted by
Constantin Nicolai
born in Bretten, Germany
Day/Month/Year Here

ON ENERGY MODELING OF DEEP NEURAL NETWORK OPERATIONS

This Master thesis has been carried out by Constantin Nicolai
at the
Institute of Computer Engineering
under the supervision of
Holger Fröning

ABSTRACT

The fast broader adoption of ML applications has caused a surge in their energy requirements, necessitating a comprehensive understanding of the tradeoffs and costs. While previous work was focused on time-only or inference-only studies, we provide a more complete picture by covering a wider space or parameters. We contribute: (1) time and energy profiling across inference and training, (2) performance prediction trained on our profiling results and (3) graphical evaluation and validation of profiling and predictor results. Profiling results of A30 performance across several core clocks reveal an energy optimum of 900 MHz, aligning with the manufacturer base clock of 930 MHz. This work provides the tools, which enable the correct choice of target GPU and clock speed for existing and future models.

ZUSAMMENFASSUNG

Die rasche Verbreitung von ML Anwendungen hat zu einem starken Anstieg der Energiekosten geführt und damit ein umfassendes Verständnis der Tradeoffs und Kosten notwendig gemacht. Der Fokus vorheriger Arbeiten liegt auf reinen Zeit- oder reinen Inference-Studien. Hierauf bauen wir auf, indem wir eine breitere Spanne an Parametern abdecken um ein vollständigeres Bild der Lage abzugeben. Unsere Contributions sind: (1) Zeit- und Energieprofiling für Inference und Training, (2) Vorhersagen trainiert auf den Profiling Ergebnissen und (3) eine grafische Auswertung und Validierung der Profiling und Vorhersage Ergebnisse. Messresultate der A30 Performance über einige GPU Clocks zeigen ein Energieoptimum bei 900 MHz, welches sich mit dem Standard Base Clock von 930 MHz deckt. Diese Arbeit stellt die Werkzeuge zur Verfügung um die richtige GPU und den richtigen Clock Speed für aktuelle und zukünftige Modelle zu wählen.

Nice quote here.

— Some Author

ACKNOWLEDGMENTS

Your acknowledgments here if desired

CONTENTS

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Problem Statement 1
- 1.3 Scope 2
- 1.4 Contributions Overview 2
  - 1.4.1 Dataset Collection 2
  - 1.4.2 Prediction Model 2
  - 1.4.3 Validation 2

2 Background 5
- 2.1 Random Forest Regressor 5
- 2.2 XGBoost Regressor 6
- 2.3 Coefficient of Determination 6
  - 2.3.1 Cross-Validation $\overline{R^2}$ 7
  - 2.3.2 Test Set R^2 7

3 State of the Art and Related Work 9
- 3.1 State of the Art 9
- 3.2 Related Work 9
- 3.3 Research Gap 10

4 Dataset Collection 11
- 4.1 Operations 11
- 4.2 Time Profiling 12
  - 4.2.1 Inference 13
  - 4.2.2 Training 13
  - 4.2.3 Proportionality 14
- 4.3 Energy Profiling 14
- 4.4 Profiling Evaluation 17
- 4.5 GPU Clocks 18

5 Prediction 19
- 5.1 Model Selection 19
- 5.2 Prediction Architecture 19
- 5.3 Prediction Workflow 21

6 Validation 23
- 6.1 Dataset Validation 23
  - 6.1.1 Methodology 23
  - 6.1.2 Hardware Platforms 23
  - 6.1.3 Results 24
  - 6.1.4 Uncertainties 24
  - 6.1.5 Tensor Core Real-World Impact 27
- 6.2 Operations Level Predictions 29
  - 6.2.1 Training A30 29
  - 6.2.2 Inference A30 30

6.2.3 Training RTX2080TI 33
6.2.4 Inference RTX2080TI 33
6.3 Neural Network Level Predictions 34
6.3.1 Patterns and Observations 42
6.4 Predictor Latency 45
7 Discussion and Outlook 47
7.1 Discussion 47
7.1.1 Limitations 47
7.1.2 Application 48
7.2 Outlook 49
7.3 Conclusion 50

Bibliography 53

LIST OF ABBREVIATIONS

- DNN - Deep Neural Network
- GPU - Graphics Processing Unit
- MSE - Mean Squared Error
- XGBoost - Extreme Gradient Boosting
- CV - Cross-Validation
- R^2 - Coefficient of Determination
- EDP - Energy Delay Product

1 INTRODUCTION

1.1 MOTIVATION

The global increase in usage of machine learning applications illustrates an acceleration in adoption across both industry and the private sector. The unfathomably large energy costs tied to this broader adoption have already prompted a change in public sentiment towards energy infrastructure. Plans for building trillion-dollar data centers are emerging, necessitating the re-commissioning of previously decommissioned nuclear power plants, which were originally phased out as part of nuclear energy reduction efforts. This reversal of nuclear phase-out policies underscores the significant infrastructural and political pressures exerted by the energy requirements of machine learning technologies.

In this landscape it is more pressing than ever to gain insight into the roots of the energy costs in order to optimize future developments on an informed basis.

In order to facilitate a more informed pairing of workload and GPU we introduce a framework to help guide the decision towards an optimal choice. By allowing for optimization towards the fastest execution time or the smallest energy footprint, our framework enables an informed choice which prevents wasteful computation by optimizing for energy when execution time is not critical.

1.2 PROBLEM STATEMENT

While a considerable amount of previous work has been done in profiling and prediction of neural network performance, no prior work covers both execution time and power consumption across training and inference.

However in most cases where a new model architecture is designed or an existing architecture is adapted, both the training and the inference efficiency are relevant at some point in the model's lifespan. As training is typically performed in datacenters, it can be both time-constrained by long execution times and strict deadlines, as well as energy-constrained by infrastructure limitations in cooling capabilities and energy budget. Inference on the other hand can be latency-constrained by real-time applications but also energy-constrained on mobile or embedded devices.

1.3 SCOPE

The intended scope for this thesis is to collect a dataset, validate its usability, use it to perform predictions and evaluate the prediction quality.
Practical constraints narrowed our study to the Nvidia RTX 2080 TI and the A30 GPUs and to the models available in the Pytorch Torchvision library[1].
The grid-search approach across the parameters revealed some inherent incompatibilities, such as models too large to fit the GPU memory or benchmarks resulting in prohibitively long runtimes due to edge case parameters combinations. Rather than troubleshooting every edge case, we decided to prioritize more practical configurations and to exclude edge cases of limited practical relevance.

1.4 CONTRIBUTIONS OVERVIEW

1.4.1 Dataset Collection

The first contribution introduces a method for automated profiling data collection and processing into a coherent dataset.
In order to ensure repeatability, the time and power readings along with a number of key related metrics are stored together with the Pytorch objects of the profiled operations.
The profiling measurements are executed via a python script that utilizes the Pytorch Benchmark library[2] to conduct the time measurements. The power profiling is conducted by running nvidia-smi[3] in the background during the benchmark execution.

1.4.2 Prediction Model

The second contribution presents our prediction model. By aggregating the predictions for the individual operations of a neural network, we can infer predictions for the complete DNN. This allows us to identify which GPU will have the shortest execution time and which one the smallest energy footprint for any given DNN architecture.

1.4.3 Validation

The third and final contribution consists of two major sections.
The first one investigates the quality of our collected dataset of individual operations. The compositional validity of the dataset is confirmed

1 https://docs.pytorch.org/vision/0.22/
2 https://pytorch.org/tutorials/recipes/recipes/benchmark.html
3 https://docs.nvidia.com/deploy/nvidia-smi/index.html

by comparing the aggregation of the individual operations which compose a neural network against measurements of the full neural network.
The second section assesses the performance of our prediction model both for individual operations and for full neural networks.

2 BACKGROUND

2.1 RANDOM FOREST REGRESSOR

In order to introduce the workings of a random forest regressor model, we first need define to a number of foundational concepts: regression, decision trees, bootstrapping and bagging. On this basis, we can describe the concept of a random forest.
Regression is a supervised learning technique approximating a function to a continuous target variable from a set of input output pairs. In the case of random forests, this function is learned by minimizing the mean squared error over an ensemble of decision trees [4][p. 10].
A decision tree is a predictor which divides up the input space in order to provide specific predictions for each region. In the tree, at each decision node one feature and a corresponding threshold are chosen to split the input space into two regions. This feature and threshold are chosen by considering all features and many thresholds and identifying the lowest MSE combination. At each leaf node, a prediction value is assigned based on the data points within that region [4][p. 307].
Bootstrapping describes the process of sampling with replacement from the training dataset. For each tree in the ensemble, a new training subset is created by randomly sampling data points from the full training dataset, explicitly allowing for duplicates. This results in each tree being trained on a slightly different subset of the training dataset, while preserving the overall data distribution across the ensemble of trees.
The ensemble technique of aggregating the results from the independently trained trees is called bagging. For regression tasks, the individual predictions are averaged, which serves to reduce model variance and improve robustness across the ensemble.
The only step necessary to move from standard bagging to a random forest model, is to adapt the process of building the decision trees. Instead of considering all features at every decision node, in a random forest only a smaller random subset of features is considered at each node. Introducing randomness within the building of the trees increases variety between the trees. This reduced correlation between individual trees is the core improvement over the standard bagging approach. The individual trees being more independent results in reduced variance in the average across the ensemble [4][p. 588].
The twofold introduction of randomness through bootstrapping and the random forest approach is the key to the overall strengths of the

random forest model in strong robustness, little overfitting and good generalization.

2.2 XGBOOST REGRESSOR

XGBoost is another tree based ensemble method for regression. In contrast to the random forest regressor, it builds its trees sequentially instead of concurrently like the random forest does.
Classic gradient boosting proceeds by building an initial tree to fit the original targets. When using MSE, this is typically the mean of the targets. From then on, it works iteratively. Each subsequent tree is trained to predict the negative partial derivative of the loss with respect to the current prediction. The trees are then added to the ensemble and weighted with a learning rate which determines the contribution of each step and controls the convergence behavior of the model [2].
The primary drawback of classic gradient boosting is its tendency to overfit, especially for larger ensembles of trees. XGBoost builds upon classic gradient boosting in order to preserve its strengths and improve its generalization.
One improvement is its inclusion of regularization. It uses a leaf weight penalty term which encourages sparsity by driving leaf weights to zero. It also introduces a minimum loss reduction threshold that controls node splitting. In practice that means, if a split does not improve the loss function by at least the threshold amount, no split occurs [2]. XGBoost also often uses a small learning rate which requires a large number of trees, but improves the training stability. Another extension is early stopping. It evaluates performance on a held-out validation set during training, stopping if no further improvement occurs. The combination of more stable training and early stopping reduces overfitting.

2.3 COEFFICIENT OF DETERMINATION

The coefficient of determination, denoted as R^2, is a statistical measure used to evaluate the performance of regression models. It measures the proportion of the variance in the dependent variable that is predictable from the independent variables.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{2.1}$$

where y_i are the observed values, $\hat{y}_i$ are the predicted values and $\bar{y}$ is the mean of the observed values.
An R^2 score of 1 indicates perfect predictions. A score of 0 indicates

the predictor performs equally to predicting the mean of the training target values. If the model performs worse than predicting the mean, the score ranges below 0.
Before the training process of any model begins, the dataset is split into a training set and a test set. The test set is held out for the final evaluation of the predictive performance on unseen data.

2.3.1 Cross-Validation $\overline{R^2}$

In k-fold cross-validation, the training set is partitioned into k subsets called folds. The regressor is trained on $k-1$ folds and the R^2 score is evaluated on the remaining fold. This process is repeated k times across all folds. The k scores are then averaged into a CV $\overline{R^2}$. This process only simulates a holdout set without actually interacting with the test set. Therefore it is a measure of how well the model works compared to other models and while it can give an indication of the predictive performance, it is not a conclusive measure its performance.

2.3.2 Test Set R^2

In contrast, the test set R^2 score is a measure of the models real-world performance, since its evaluation is performed on a true holdout set. It provides a conclusive measure of the models generalization and its real-world performance.

3 STATE OF THE ART AND RELATED WORK

3.1 STATE OF THE ART

The challenge of predicting neural network performance has invited a multitude of approaches. Apart from the methodological approaches they also differ in a number of other aspects. While execution time is commonly the metric of choice, only few go further and also study metrics like power consumption and memory footprint. Another important distinction is the workload studied in the work, more specifically, whether both training and inference are studied. For practical reasons it is also relevant which machine learning framework is used and what hardware targets are required and can be predicted for. These many dimensions of possibility result in no work covering all possibilities, but allows for many different approaches which have use cases in a given situation.

3.2 RELATED WORK

Kaufmann et al. take an approach of performance modeling by means of the computation graph. They are however limited to the Google Tensor Processing Unit in this work [7].
Justus et al. take an approach exploiting the modular and repetitive nature of DNNs. Given the same operations are repeated over and over in training, often only varying in a few key parameters, the execution time for these base building blocks is measured. This is then done for one batch in the training process and generalized to the whole training process from there. There is however no presentation of the methodology for the execution time measurements [6].
Qi et al. present PALEO which employs an analytical approach towards predicting the execution for both training and inference of deep neural networks. The analytical approach brings both advantages and disadvantages with it. It does not require a dataset of measured execution times as a training set in the same way many other works do, but on the other hand it also is based on more fixed assumptions about the DNN execution than a more data driven approach [8].
Wang et al.'s approach uses a multi-layer regression model to predict execution time for training and inference. Their work is however rather limited in terms of hardware targets and different DNNs studied [11].
Cai et al. focus their work, NeuralPower, on CNNs running on GPUs. For each target GPU, they collect a dataset and fit a sparse polynomial

regression model to predict power, runtime, and energy consumption. While NeuralPower achieves good results, its usefulness has become limited due to its exclusive focus on CNNs, as other DNN architectures have grown in popularity [1].
Gianitti et al. also exploit the modular nature of DNNs in their approach. They define a complexity metric for each layer type, optionally including back propagation terms, allowing them to predict execution times for both training and inference. However, their method faces significant limitations, as the complexity metric is only defined for a specific set of operations, making it incompatible with networks that include layers not covered in the original work. As a result, their approach is essentially limited to classic CNN architectures [3].
Velasco-Montero et el. also take the familiar per-layer approach. Their predictions are based on linear regression models per type of layer, but again for a specific set of predefined operations. Given their focus on low-cost vision devices these restrictions are reasonable, but limit generalizability [10].
Sponner et al. take a broad approach in their work. It works in the TVM framework giving it high flexibility in target hardware and studied metrics. It is in fact the only work to include execution time, power consumption and memory allocation. Given the automated data collection used to create the dataset basis for the predictions, there are also few limitations to the networks that can be studied with this. The predictions are based on an extremely randomized tree (ERT) approach with XGBoosting applied. The only major drawback for this work is its limitation to only study inference, due to TVMs limitation to inference [9].

3.3 RESEARCH GAP

Despite a considerable amount of work on this topic, no single study has covered all possible angles of interest. Given the current landscape of available publications our work will focus on finding the best GPU for a PyTorch job. In order to achieve that, we will cover execution time, power and energy consumption and will provide inference and training predictions for these metrics. Our approach also employs a different method of automatic dataset collection, which allows for a broad field of study. Power readings are collected directly through nvidia-smi. While this limits our work to Nvidia GPUs, this methodology could just as well be applied to any other hardware target which supports reporting power readings.

4 DATASET COLLECTION

This contribution outlines the method used to collect a profiling dataset. The dataset serves as a training set for the prediction model. The parameters covered are various Torchvision models and input sizes, execution time and power, both the inference and the training case as well as multiple GPUs and various GPU clocks.

4.1 OPERATIONS

In order to increase generality, our approach exploits the layerwise structure of DNNs by working at the layer level rather than the model level. In order to ensure rigor we must clarify our terminology.
On the layer level, the workload characteristics are determined by the layer and its input features. On this level, the objects with an identical characteristic workload are layers with specific input feature dimensions and layer settings. We will refer to these objects as operations.
At the model level, the objects with an identical characteristic workload are specific models with specific input dimensions. We will refer to these as model-input sets.
A study like this could have been conducted on the model level, the operations level or the kernel level.
We chose the operations level over the kernel level, because it is closer to the model level and the model level determines the performance of real-world applications. The operations level is still close enough to allow us to infer results for the model level from it. We chose it over the model level itself, because all the combinations of operations open up a wider space of possibilities than the model level ever could.
With these considerations, the units for which we will make predictions later are individual operations. To this end, we aim to collect a dataset of operations and profile the execution time and power for each one. To ensure that the dataset reflects operations encountered in real-world scenarios, we select a number of representative model-input sets from models of the Torchvision library. The set of operations which we will be profling, is the set of all operations occuring in any of these model-input sets.
Extracting this set is automated via scripts, but can just as well be done by hand if so desired. With this set at hand we can dive into the actual profiling.

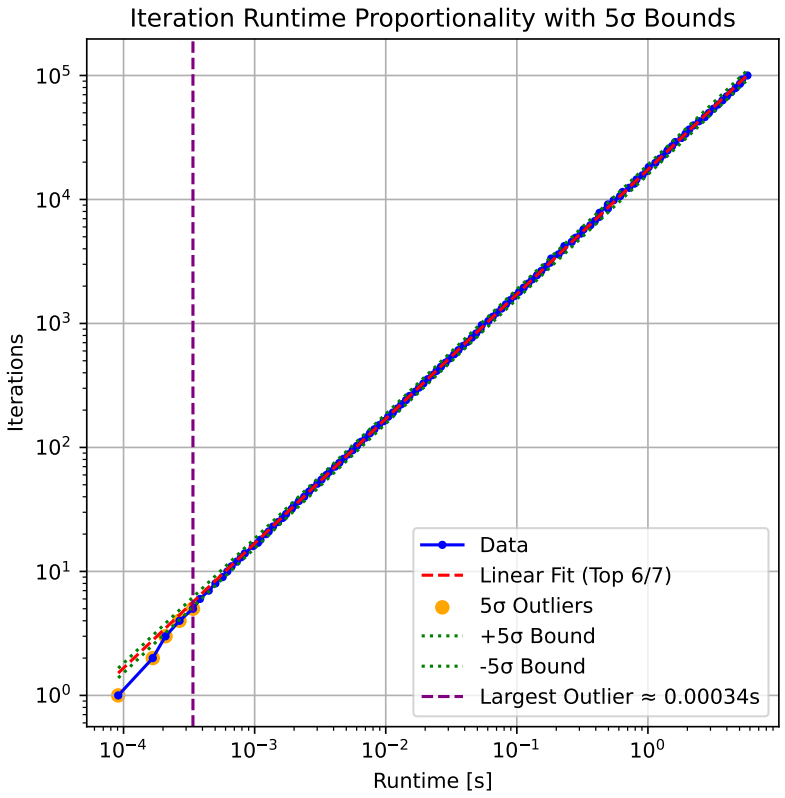

Figure 4.1: This plot shows the proportionality of the runtime with the number of iterations ran for an operation. A linear fit is performed to the $\frac{6}{7}$ of data points with most iterations. From this fit the standard deviation for the linear section is found. This allows us to identify the largest outlier below which the proportionality breaks. In order to do this we search for the largest runtime in the 5σ outliers.

4.2 TIME PROFILING

The time profiling is performed using the benchmark utility from `torch.utils.benchmark` called Timer. This enables us to run our benchmark function with the specific layer and input features as input parameters, allowing us to run it for each operation.
To achieve more stable results and improve the simultaneous power profiling, it is desirable to run the benchmark for each operation for at least a few seconds. This is achieved by setting a minimum runtime for the benchmark to aggregate statistics. Different workload characteristics and different computational capabilities depending on the GPU configuration being profiled result in different requirements for individual profiling runs in order to ensure sufficient repitions in the measurement process. In order to accomodate this requirement, the runtime of the measurement is configurable via a command-line

parameters.
For the initialization of the operations we have to be careful to avoid two extreme cases in order to achieve a sensible approximation of the execution characteristics within a neural network. The first case we want to avoid is initializing one instance of the operation and performing each benchmark loop on that instance. This would result in hitting the cash every time and would not be representative. The second extreme we want to avoid is initializing a new instance for each loop of the benchmark run, which adds the initializition overhead and the larger latency of having to access the GPU main memory every time, skewing our results in the opposite direction. We have therefore chosen the middleground approach of initializing a sizable block of operation instances with random weights and biases, which are then looped over by the benchmark kernel, resulting in some reuse of layers, stressing the memory system in a sensible manner.

4.2.1 Inference

The central difference between inference and training profiling appears in the design of the benchmark function.
For inference profiling we put our operations into `eval` mode and call the operations within a `torch.inference_mode` environment, which is equivalent to the execution in a typical inference forward pass through a model. Within the loop we call the operator on a preinitialized random input tensor for the forward pass. The input tensor is a single instance, as it represents the activations from the previous layer which can be expected to live in high level memory in our memory hierarchy. Since this forward pass is everything we want to profile for inference, this benchmark loop is sufficient.

4.2.2 Training

For the training profiling we put our operations into `train` mode and omit the environment used in the inference case. In order to portray the training process for a single operation, we need to run a forward pass and a backward pass. The forward pass is handled identically. In order to execute the backward pass we need a substitute for the gradients flowing backward trough the model. This is achieved by preinitializing a random torch vector with the dimensions of our operation's output which we can then call the backward pass on. Other than that, we only have to make sure the runtime keeps the gradients for all operation instances and keep resetting the input vector gradients for each loop iteration.

4.2.3 Proportionality

Since we are assuming the number of benchmark iterations and the resulting total runtime to be proportional, we need to test that assumption. In order to do that, we plot their relationship in Figure 4.1. By identifying how short the total runtime needs to become for the linear relationship to break, we can find a lower bound to our desired benchmark duration for each operation we study. Our investigation reveals a lower bound of 100 ms. In our actual measurements every operation is benchmarked for at least several seconds, even though the exact duration may vary depending on the computational intensity of the specific operation.

4.3 ENERGY PROFILING

The energy profiling is conducted by measuring the power in watts. Combined with the time measurements this gives us the energy results.
We are starting `nvidia-smi` as a background process and logging the power readings into a temporary csv file, which is used to calculate the operations's power average.
On a higher level abstraction, a second instance of `nvidia-smi` runs during the complete benchmark process. This log can be compared to the timestamps included in the dataset of profiled operations, in order to investigate surprising annomalies in the results.
Due to the nature of our measurement pipeline, some preparations and processing after the fact are necessary to ensure robust results.
As can be seen in Figure 4.2, showing the power measurement over time for alternating idle times and benchmark calls, the transition is not instant. There are some power readings in between the steady states of idle and benchmark.
In order to illustrate the existance of a startup effect without idle times between the benchmark runs, Figure 4.3 shows five looped benchmark runs of the same benchmark overlayed. For each run we can observe the startup effect.
We do not want to keep this startup effect in our result, because it is a result of our benchmark execution and not representative of the much more integrated execution pipeline in a neural network.
In order to keep it from affecting our results, we use a 3σ channel around the initial mean of the power readings and drop everything outside the channel.
As a second precaution, the script also employs some warmup runs in order to further mimimize the impact of the startup effect.

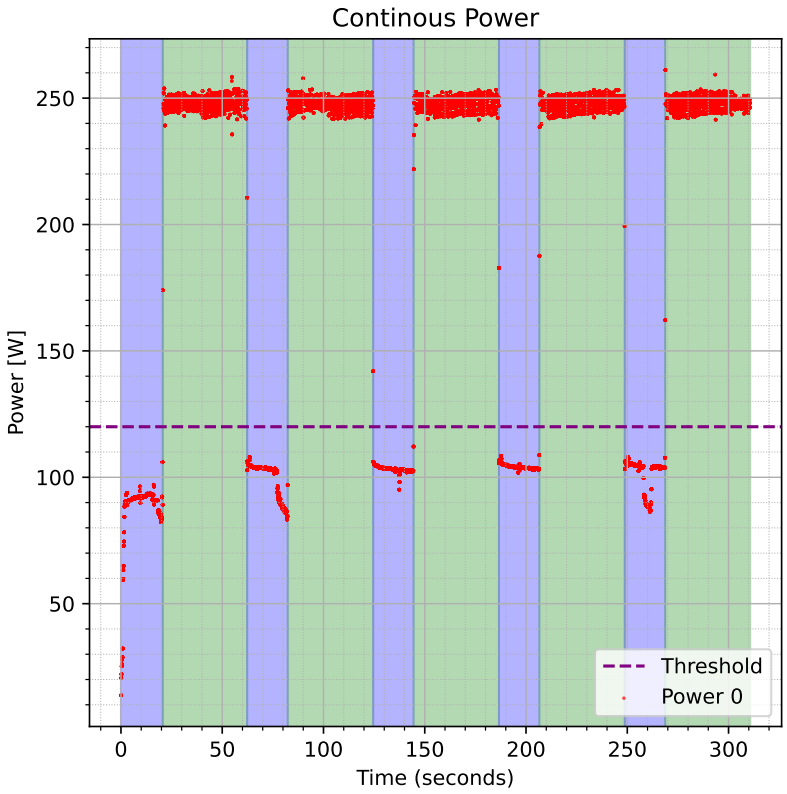

Figure 4.2: Power log on the RTX 2080 TI for alternating sleep and benchmark calls. Benchmark and sleep are marked as coloured sections. At each transition there are a few readings in between the two steady states. Those are the startup effect, that we are filtering out by the use of a 3σ channel around the initial mean and dropping all readings outside.

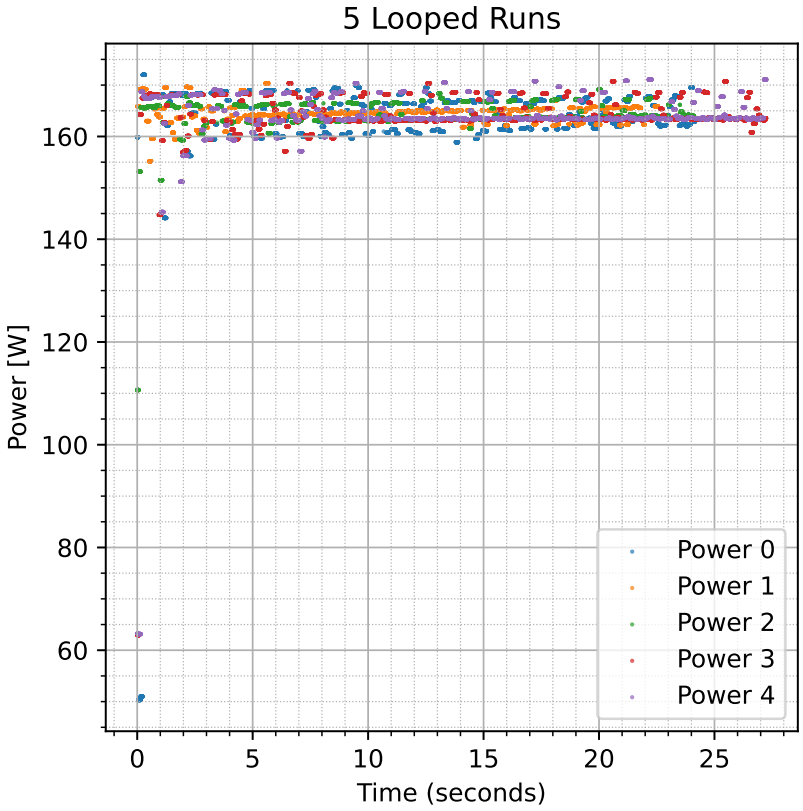

Figure 4.3: Power logs on the A30 for five runs of the same benchmark. They are overlayed to illustrate the reproducable pattern. Since the benchmark was run in the loop, we conclude the startup effect is not a result of the idle time introduced by the sleep calls in Figure 4.2, but is also present in our actual measurement script.

4.4 PROFILING EVALUATION

The following section explains the details of how we arrive at our results from log files collected in the benchmark runs. Below you can see a snippet from one of the log files. The third entry in each row is the power reading.

```
2024/10/10 13:18:58.369, 81, 145.99, 4396, 11264
2024/10/10 13:18:58.407, 81, 182.11, 4396, 11264
2024/10/10 13:18:58.428, 81, 182.11, 4396, 11264
2024/10/10 13:18:58.439, 81, 182.11, 4396, 11264
2024/10/10 13:18:58.490, 81, 178.50, 4396, 11264
2024/10/10 13:18:58.514, 81, 178.50, 4396, 11264
2024/10/10 13:18:58.538, 81, 178.50, 4396, 11264
```

Let us begin with the power evaluation. The log file is read in via `pandas`[1]. Any existing rows containing a non-numerical value are dropped from the dataframe. We then find the standard deviation for the power and drop all rows containing a power reading outside a $3\,\sigma$ range. The following formulae for the mean and standard deviation are used:

$$\overline{W} = \frac{1}{n}\sum_{i=1}^{n} W_i \tag{4.1}$$

$$\sigma_W = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(W_i - \overline{W})^2} \tag{4.2}$$

$$W_{filtered} = W \text{ such that } |W - \overline{W}| < 3\sigma_W \tag{4.3}$$

With n being the number of timestamps and W being the power. The same two formulae are used to find the mean power $\overline{W}_{filtered}$ and standard deviation $\sigma_{\overline{W}_{filtered}}$ of the filtered power.
Both the total runtime t_{tot} and its standard deviation $\sigma_{t_{tot}}$ are provided by `torch.utils.benchmark`.
In the next step we find the total run energy E_{tot} and its error $\sigma_{E_{tot}}$.

$$E_{tot} = \overline{W}_{filtered} \cdot t_{tot} \tag{4.4}$$

$$\sigma_{E_{tot}} = \sigma_{\overline{W}_{filtered}} \cdot t_{tot} \tag{4.5}$$

1 pandas

From this, we find the time per iteration t and the energy per iteration e, as well as the error for the time per iteration σ_t and for the energy per iteration σ_e with the number of iterations being N.

$$t = \frac{t_{tot}}{N} \tag{4.6}$$

$$e = \frac{E_{tot}}{N} \tag{4.7}$$

$$\sigma_t = \frac{\sigma_{t_{tot}}}{N} \tag{4.8}$$

$$\sigma_e = \frac{\sigma_{E_{tot}}}{N} \tag{4.9}$$

The most interesting results here are the time and enery per iteration with their respective errors, as well as the average power.

4.5 GPU CLOCKS

The last parameter we needed to build a test methodology for is the GPU clock. More precisely the core clock of the GPU. A dedicated clocking script[2] takes command line parameters to set the desired clock speed. A range of clock speeds was tested in order to gain insight into the relationships between clock speed, runtime and energy consumption.

2 Because using `nvidia-smi` to change the clock requires lower level permissions, this was done in a seperate script with additional permissions, which was then called from the `sbatch` script used to run the benchmark.

5 PREDICTION

In this chapter we will introduce our model for providing predictions based on the collected dataset. We will go into the idea and decisions in creating it in this specific way and provide insight into the implementation.

5.1 MODEL SELECTION

Since we are interested in time and energy, we need two prediction models. More precisely, we need one for runtime predictions and one for power predictions. By multiplying the two predictions we obtain our energy prediction.
In order to retain as much understanding of the prediction process as we can while ensuring tolerable execution times for the prediction times, we do not want to use a full DNN to perform the predictions. Instead we are looking for a more lightweight solution, closer to classical statistics. These requirements led us to using a random forest predictor model. It is both lightweight and sets restrictions to its input which force us to format our dataset in a way that provides some insight into cause and effect for the predictions.
From the same family of tree based ensemble methods, we also ran some tests with Extremely Randomized Trees, but did not find improved results results.
A similar attempt was made with Extreme Gradient Boosting. Using XGBoost resulted in similar performance to the random forest model. Both models outperformed the other one is some scenarios. For the sake of simplicity we will continue forward with the random forest model. It is more simple in concept and exhibits better scalability characteristics due to its inherent parallelisation.

5.2 PREDICTION ARCHITECTURE

We are using an existing random forest implementation provided by sklearn[1]. Therefore most work went into formatting and preprocessing the dataset. In order to give the model as much useful input information as possible we needed to provide the operator name, the input size and the GPU clock along with all potentially useful attributes of

1 sklearn random forest implementation

```
[0 1 3 1 64 1 7.0 7.0 1.0 3.0 3.0 1.0 2.0 2.0 1.0 -1 0 -1 0
-1 0 -1 0 -1.0 0 -1.0 0 -1 0 -1.0 0 -1.0 0.0 -1.0 0.0 -1.0
0.0 -1.0 0.0 -1.0 -1.0 0.0 False False False True False
False False False False False False 32.0 1.0 3.0 1.0
224.0 1.0 224.0 1.0 1440]
```

Figure 5.1: This is an example of the final format of the input vector for the sklearn predictor.

the operation's pytorch object to the predictor.
Because random forest models only use numerical inputs, we made sure to format the input vector in a suitable way. The final format can be seen in Figure 5.1.
For the operator names, this means we have a fixed number of categories in our dataset and can therefore use one-hot encoding to identify each type in a way that is readable to the random forest model. Another challenge arose from the fact, that different operations do not always have the same attributes. Along the same lines, the length of the input size tuple also is not equal for all operators. We do however want to support all operators in one predictor model, which in turn means, we needed to find a solution for this asymmetry in attributes for the different operators. The approach we ended up using, was to introduce a Boolean flag for each attribute entry in the input vector, signifying whether is applicaple for this operator.
For example, a linear layer expects an input tensor of the shape: `(batch_size, in_features)`, but a `Conv2d` layer expects: `(batch_size, in_channels, height, width)`. With that, the input size tuple for for a linear layer with `(batch_size=32, input_features=128)` would be encoded as `(32,1,128,1,-1,0,-1,0)`, whereas the input size tuple for a `Conv2d` layer with `(batch_size=32, in_channels=16, height=256, width=256)` would result in `(32,1,16,1,256,1,256,1)`. This way we can construct a meaningful input vector for the random forest model, which has a constant length and meaningful entries. We use -1 as our entry for non applying fields, as there are no negative input sizes. For the flags, a 1 signifies the entry being applicaple, while a 0 signifies it not being applicable.
In a similar fashion, there is a field in the input vector for `stride`. For convolutions, this carries the information of the stride, but for other operators like linear layers or ReLUs, it simply carries a minus one, with a zero flag in the next entry signifying it does not apply for this operator.
This encoding approach is used on a predefined list of attributes which are likely to have an impact on the computational characteristics of the operation. The choice of these parameters was conducted in a heuristic fashion, based on our understanding of the operators in question. However, further work could try to minimize the number of necessary attributes by quantitatively investigating their computational impact

and including or excluding attributes accordingly. With the heuristic approach used here, we have attempted to err on the side of rather including more attributes than necessary instead of too few.
The last important input vector entry we need to mention is the GPU clock speed. Even though the initial implementation trained runtime and power predictors for each individual clock speed in our dataset, treating the GPU clock as a parameter for the model reduces unnecessary complexity without having a negative impact on our prediction accuracy. This way it is just another parameter for the predictor, simplifying both the training of the random forest models, as well as their utilization, since it is no longer necessary to locate the specific model for the GPU clock speed of interest, and can instead just be fed into either the inference or the training predictor.

5.3 PREDICTION WORKFLOW

As apparent from the architecture, our prediction model operates on the operations level. In case this is what is desired for a given study, the current state is already sufficient. One can simply provide the operations of interest with their respective input size and GPU clock to preprocessing and prediction script and will receive predictions.
Nevertheless, we expect most users to want to make predictions for full neural networks. In that case the workflow is very similar to the dataset creation. It starts with extracting the operations and how often each operation occurs. Afterwards all unique operations will be run through the predictor and the results will be summed up according to their number of occurences within the model.
Critically, due to the modular nature of the approach, there is no need for the neural network being predicted to be actually executed or even exist in functioning form. As long as the operations are available and it is known how often they occur predictions can be made.

6 VALIDATION

While the previous contributions provided insight into the building blocks of this work, this chapter will serve to present its results. By providing quantitative results we can validate the methodology and provide an informed impression of both its capabilities and limitations.

6.1 DATASET VALIDATION

In this first section our focus is to ensure the datasets we collect for training are reasonable accurate. This is necessary in order to prevent the introduction of a strong bias due to dataset inaccuracies.

6.1.1 Methodology

Our measurements of the individual operations should resemble how they behave when executed within a complete DNN. Since our individual operations are merely building blocks, we cannot compare them directly to the entire neural network. Instead, we aggregate the measurements of the operations which compose the DNN and compare that total to the execution of the full model for validation.
Our measurements for the complete DNN executions are performed using the same pipeline used to measure the individual operations. This is desirable because using the same pipeline for all measurements ensures comparable results.
We also built a script that extracts which unique operations are present in a specific DNN in a way that it also tracks how often each unique operation occurs. This way, we can sum the results from our collected dataset of operations accordingly.

6.1.2 Hardware Platforms

The two hardware platforms studied here are the Nvidia RTX 2080 TI and the Nvidia A30. The Nvidia RTX 2080 TI is based upon the Turing architecture from the year 2018 and features 4352 CUDA cores and 544 first generation tensor cores with `FP16` support. The Nvidia A30 is based upon the Ampere architecture form the year 2020 and features 3584 CUDA cores and 224 second generation tensor cores with `TF32` support.
Given the capability of working with `FP32` values using the `TF32`

datatype on the A30, we decided to probe its performance characteristics between having its tensor cores enabled and disabled. We cannot make the same differentiation for the 2080TI, because its tensor cores do not support FP32, which we use in all our benchmarks.
This leaves us with three configurations for the dataset validation. The 2080TI with default settings, the A30 with default settings and the A30 with its tensor cores disabled.

6.1.3 Results

RTX 2080 TI Our results for the 2080TI are mixed. Agreement between measured and summed results does look rather promising for larger model-input sets. However, for smaller ones, there are instances where the agreement weaker. The model-input sets displaying this behavior are the EfficientNetB0 (32, 3, 224, 224), the ResNet18 (32, 3, 32,32) and the ResNet34 (32, 3, 56, 56). In these instances, the summation overestimates both runtime and energy. However, the overestimation is more pronounced for runtime than for energy.
A30 Tensor Cores Disabled Our results for the A30 with its tensor cores disabled are already more precise than the 2080TI's. While the same trends are visible, they are much less pronounced and our summation yields a closer approximations of the measurements overall.
A30 with Tensor Cores Enabled Our results for the A30 with its tensor cores enabled are very promising. While the earlier trends did not completely vanish, they are even less pronounced than for the A30 with disabled tensor cores. This hardware configuration yielded the most precise summation of the three configurations.

6.1.4 Uncertainties

The standard deviation in our results is very small, both for the summation and for the measurements. It is clear that the discrepancies between the two cannot be solely caused by statistical noise. The larger contributor necessarily are systematic uncertainties which dominate the statistical noise remaining in our measurements. The specific systematic contributors were not identified. Given this situation, statistical errors are omitted in the evaluations following the dataset validation.

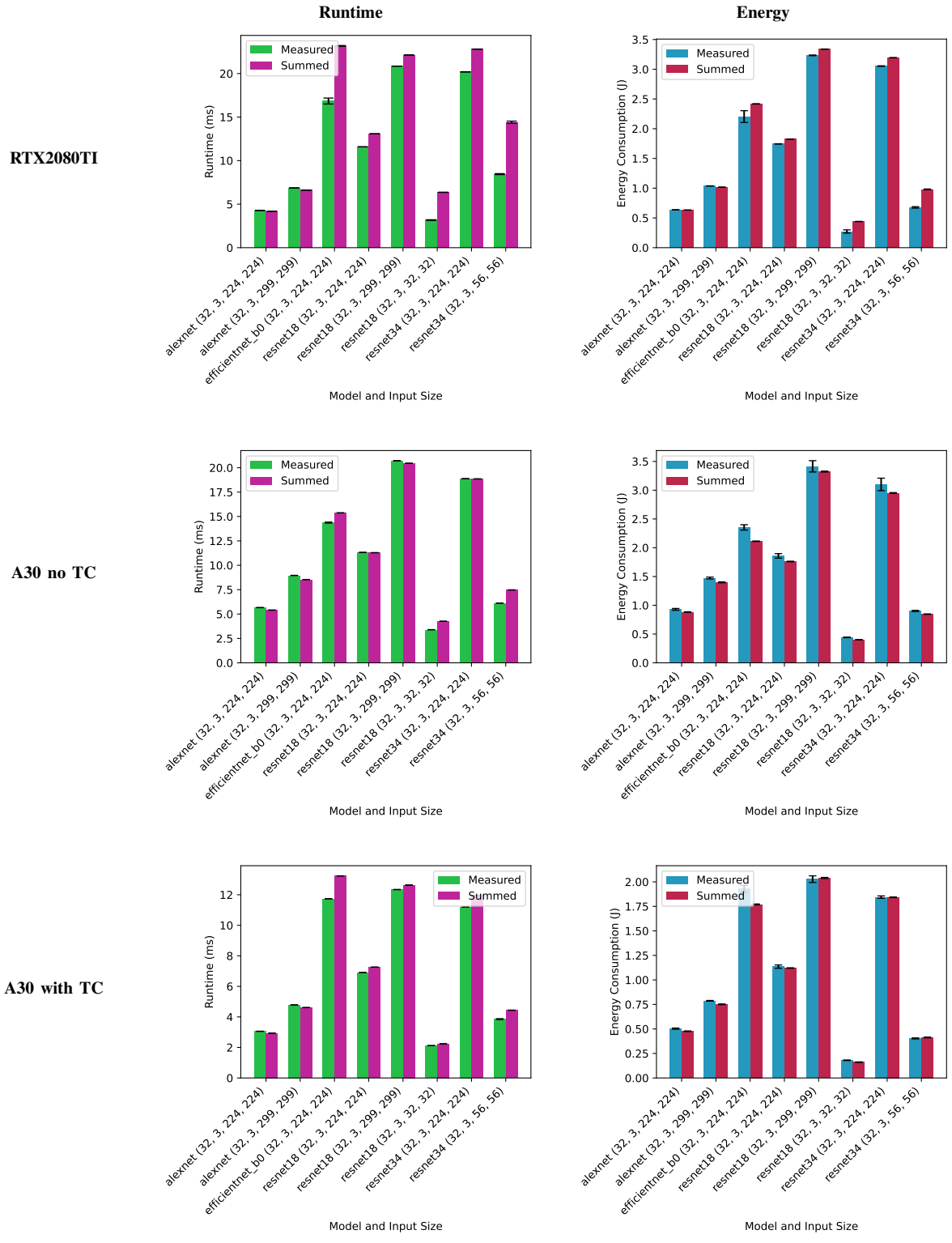

TABLE I: Measured and predicted runtime and energy consumption for smaller model-input sets. The left column shows the runtimes we measured with full model-input runs compared to our approximations. The right column shows the same comparison for the energy consumption. The approximations were obtained by summing the individual findings for all operations within the model-input set. The error bars show the standard deviation for the full model-input runs and the result of error propagation of the individual standard deviations for the summed approximation.

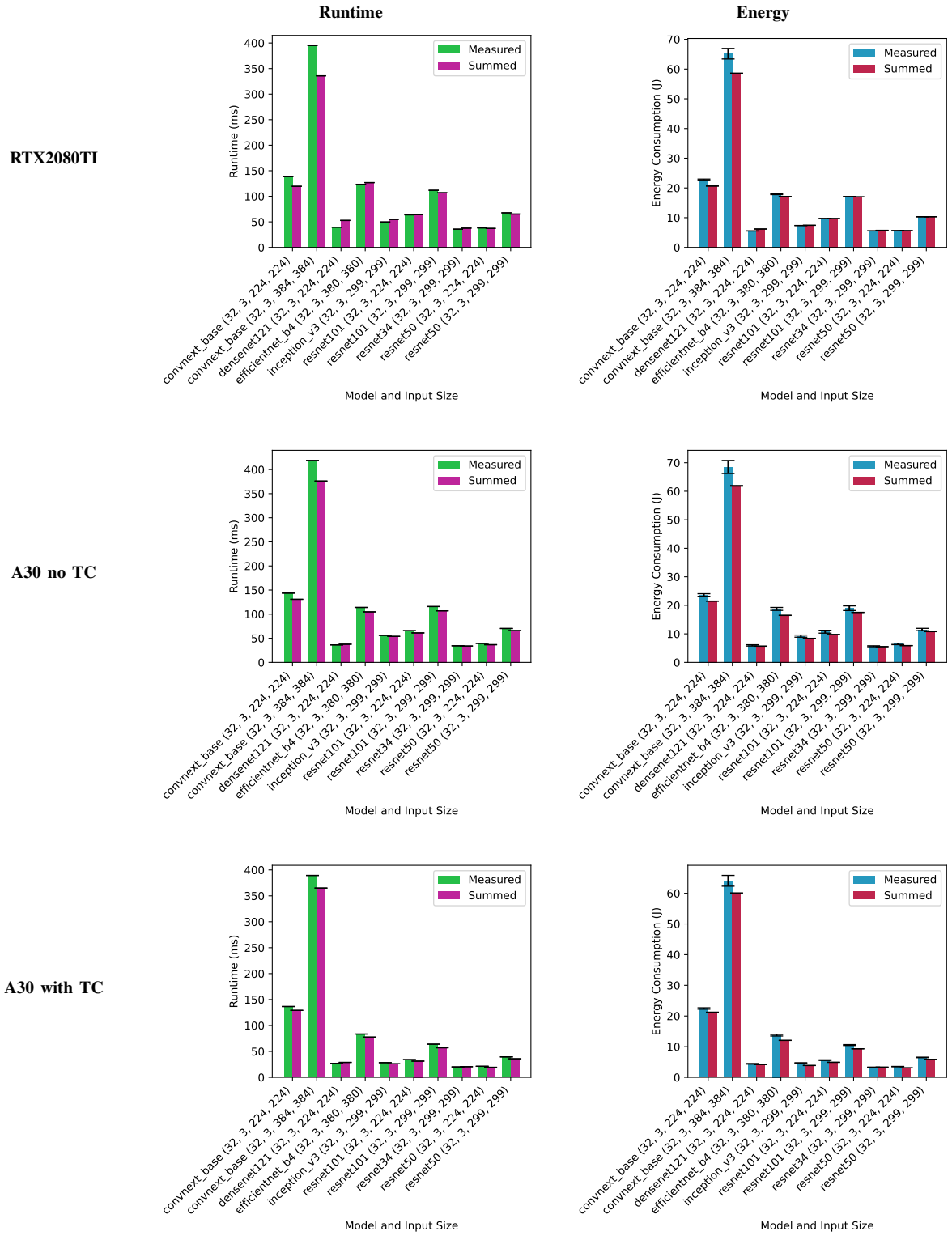

TABLE II: Measured and predicted runtime and energy consumption for larger model-input sets. The left column shows the runtimes we measured with full model-input runs compared to our approximations. The right column shows the same comparison for the energy consumption. The approximations were obtained by summing the individual findings for all operations within the model-input set. The error bars show the standard deviation for the full model-input runs and the result of error propagation of the individual standard deviations for the summed approximation.

6.1.5 Tensor Core Real-World Impact

As can be seen in Figure 6.1 and Figure 6.2 showing the measured energy for the full model-input set runs on all three GPU configurations, tensor cores do have a significant impact on the energy efficiency of running PyTorch models. This is illustrated by the difference between the results for the A30 with tensor cores and without tensor cores.
This difference is more pronounced for smaller model-input sets and appears to become continually smaller for larger and more complex ones. But the difference does not appear to simply be proportional to the model's energy cost either. At first glance and without studying the individual model architectures in detail, it would appear that the difference decreases with the model's dependency complexity.
In this context, dependency complexity refers to the depth and quantity of dependencies in a model. Depth is defined as the number of layers a dependency spans when it goes beyond a sequential connection between adjacent layers. Skip connections are a prime example of this phenomenon.
When comparing the results for different flavors of ResNets to the results for model architectures with higher dependency complexity such as ConvNext, EfficientNet and DensetNet, it can be seen that the results are much closer for the latter ones, while for the ResNets the tensor cores get to show their potential.
Taking a step back from studying the impact of the tensor cores, there are also interesting findings in comparing the results for the 2080TI to the other GPU configurations. We find worse energy efficiency for the 2080TI compared to the A30 running with tensor cores for all models. When the tensor cores are disabled this trend gets reversed. Overall the difference between the 2080TI and the A30 without tensor cores is smaller than the difference between the 2080TI and the A30 with its tensor cores enabled. However, the pattern of the energy efficiency being best on the A30 with tensor cores, the 2080TI occupying the middle position, and the A30 without tensor cores having the worst energy efficiency remains the same for all model-input sets.

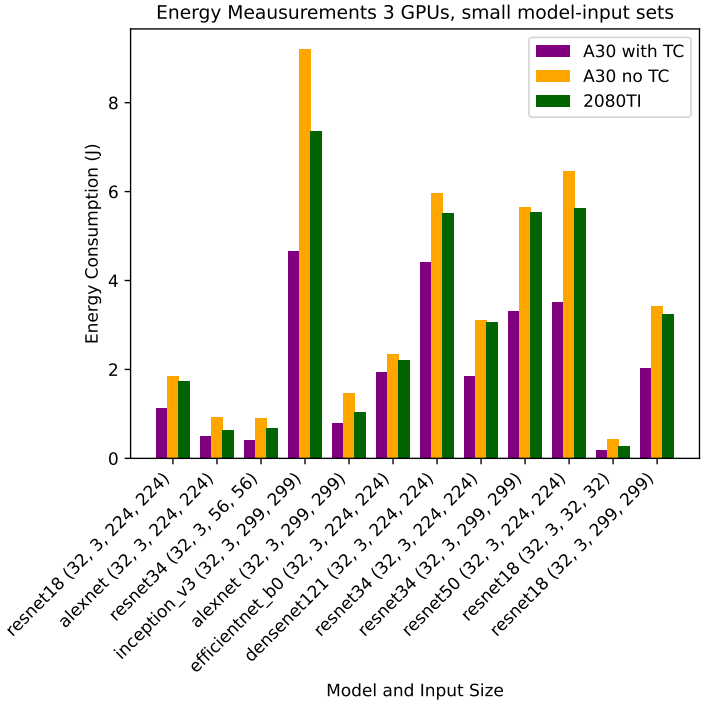

Figure 6.1: Comparison of energy measurements for the 2080TI and the A30 with tensor cores once disabled and once enabled. The resulting ordering is identical for all model-input sets. However, the relative differences show a lot of variation, being more pronounced for these smaller model-input sets.

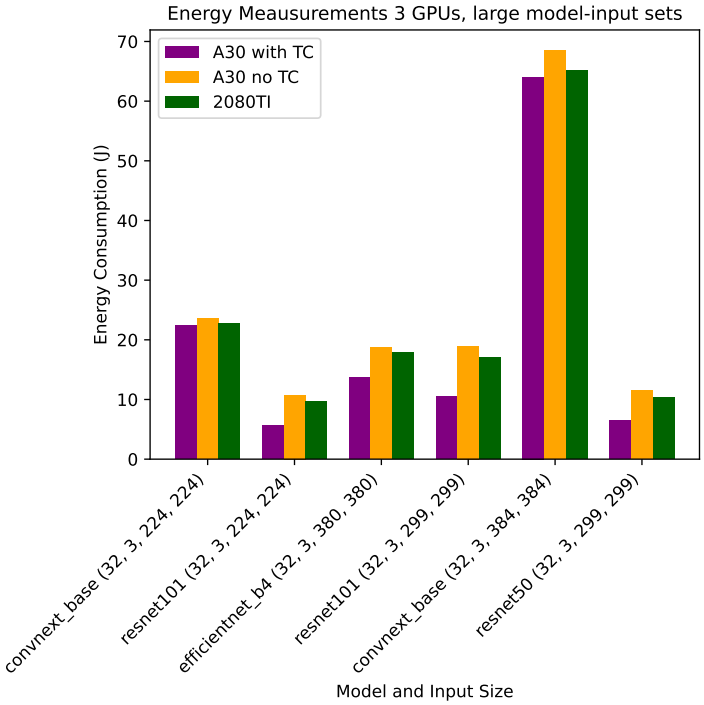

Figure 6.2: For these larger model-input sets, we maintain the ordering, but we observe far weaker relative differences between the hardware configurations. For model-input sets with a high dependency complexity, like the ConvNext Base model, we find the most similar energy results across the configurations.

6.2 OPERATIONS LEVEL PREDICTIONS

We begin by evaluating our prediction model on the operations level. We are using the implementations from the `sklearn` and `xgboost` libraries and have investigated XGBRegressor[1], ExtraTreesRegressor[2] and RandomForestRegressor[3] to use as our prediction model. Initial tests showed clearly that ExtraTreesRegressor performed worse than the others in terms of predictive power, so we focused on the other two.
We chose to use the coefficient of determination, also known as R^2, as our metric to evaluate the prediction accuracy.
In order to provide a more well rounded representation, both the R^2 score for a test set separate from the training set, as well as the mean R^2 score for a 15 fold cross validation are given.
The inference and training datasets are conceptually separate and measured independently, therefore the prediction models are separate as well. Naturally, their evaluation is split up too.

6.2.1 Training A30

Training A30	Random Forest	XGBoost
CV $\overline{R^2}_{time}$	0.876 ± 0.073	0.912 ± 0.04
Test Set R^2_{time}	0.9036	0.9018
CV $\overline{R^2}_{power}$	0.977 ± 0.007	0.985 ± 0.003
Test Set R^2_{power}	0.9797	0.9868

Table 6.1: Operations level results for the random forest and XGBoost A30 predictors. Cross validation and test set score are in close agreement.

The resulting values for the model predicting training performance can be found in Table 6.1. The scores show a small amount of variation depending on the random seed used for the predictions models and the test set, training set split.
For A30 training predictor, the XGBoost model performs slightly better in terms of prediction performance. However the random forest model has the advantage of being more simple in nature and is expected to exhibit better scalability properties in general. For this reason and in order to keep the predictor model type constant between different predictors, all results past the operations level evaluation were acquired using the random forest model.
We also included Figure 6.3 in order to convey a visual impression of

1 xgboost XGBRegressor
2 sklearn Extra Trees Regressor
3 sklearn Random Forest Regressor

the prediction performance. For a random subset of operations from the test set, it shows the predicted value alongside the measurement value.
Both the R^2 score results as well as the visual interpretation tell a similar story. We clearly have a stronger predictive power for the power model than for the runtime model. For either model prediction performance and prediction latency requirements will determine whether the performance we can provide is suitable for any given application.

6.2.2 Inference A30

The resulting values for the model predicting inference performance can be found in Table 6.2. Our resulting R^2 scores agree with the ones for the training model.
The slightly lower scores of the runtime model with XGBoost and the random forest test set score are explained by inherent uncertainties. However one possible explanation going beyond uncertainty originates from the smaller absolute runtimes for inference. Smaller runtimes with the same measurement methodology result in similar absolute uncertainties and therefore in larger relative uncertainties. These larger relative uncertainties cause the slightly weaker predictive performance we observe for the inference predictor.
All power model scores both for the random forest model, as well we the XGBoost model show identical performance to the training model, taking into consideration the additional uncertainty introduced by the random seed affecting the predictive performance slightly.
Given the very similar results, both in these metrics and in the visual interpretation of Figure 6.4, our conclusions for the inference model are the same as for the training model. The lower predictive power of the runtime model compared to the power model will result in a limit of its suitable applications, but it remains application dependent.

Inference A30	**Random Forest**	**XGBoost**
CV $\overline{R^2}_{time}$	0.890 ± 0.66	0.898 ± 0.05
Test Set R^2_{time}	0.8827	0.8924
CV $\overline{R^2}_{power}$	0.981 ± 0.003	0.988 ± 0.002
Test Set R^2_{power}	0.9853	0.9895

Table 6.2: Our inference predictor for the A30 does very well. With runtime scores just below 0.9 and power scores higher than 0.98, it is performs just as well as the A30 training predictor.

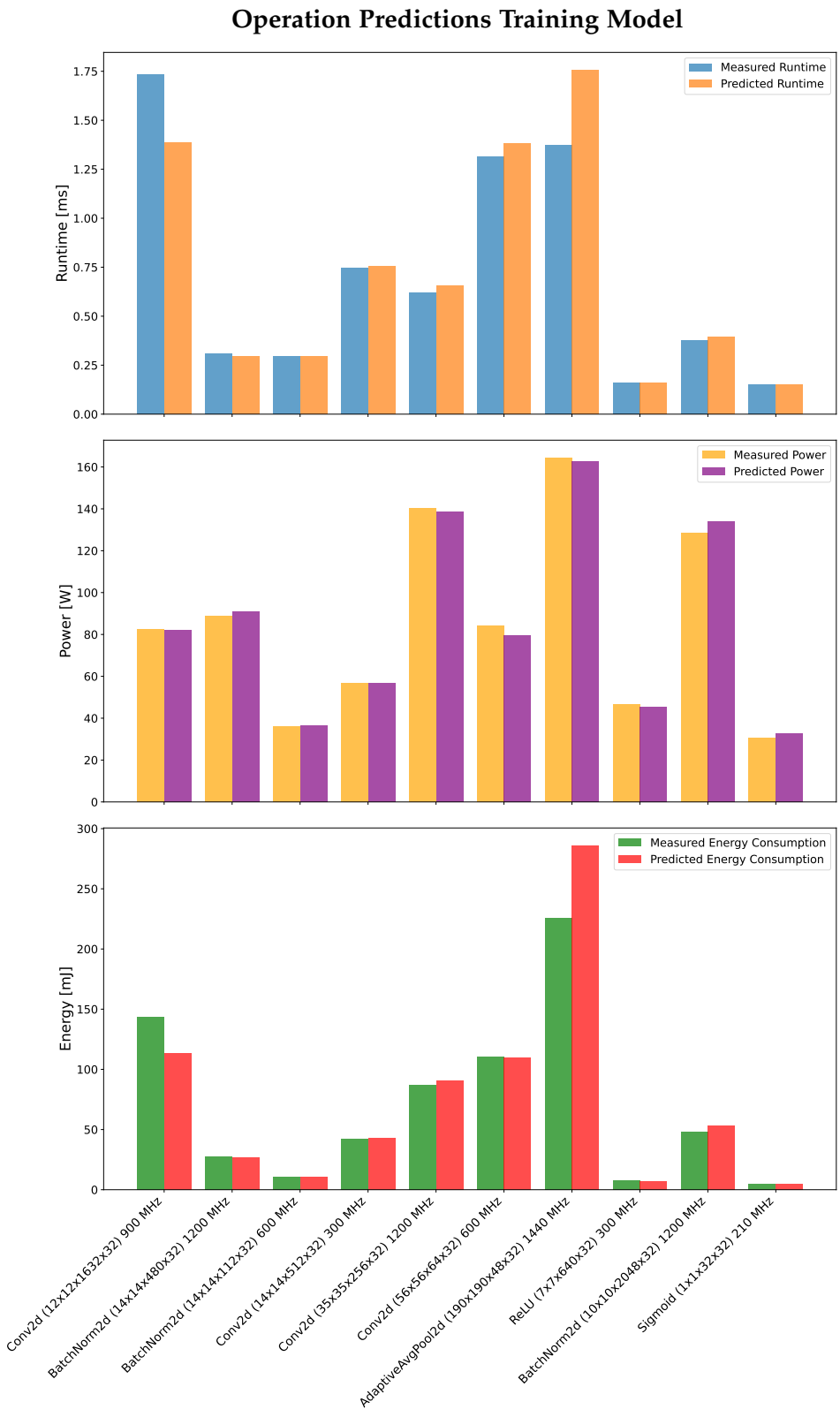

Figure 6.3: Comparison of predictions to measurements for ten operations from the test set for the A30 training prediction model. While we have outstanding agreement for most of them, we can see the limits of our predictive power in the results for operations 2 and 9. This illustration can give us a more intuitive impression of the R^2-error of 0.87 for the runtime and of 0.97 for the power.

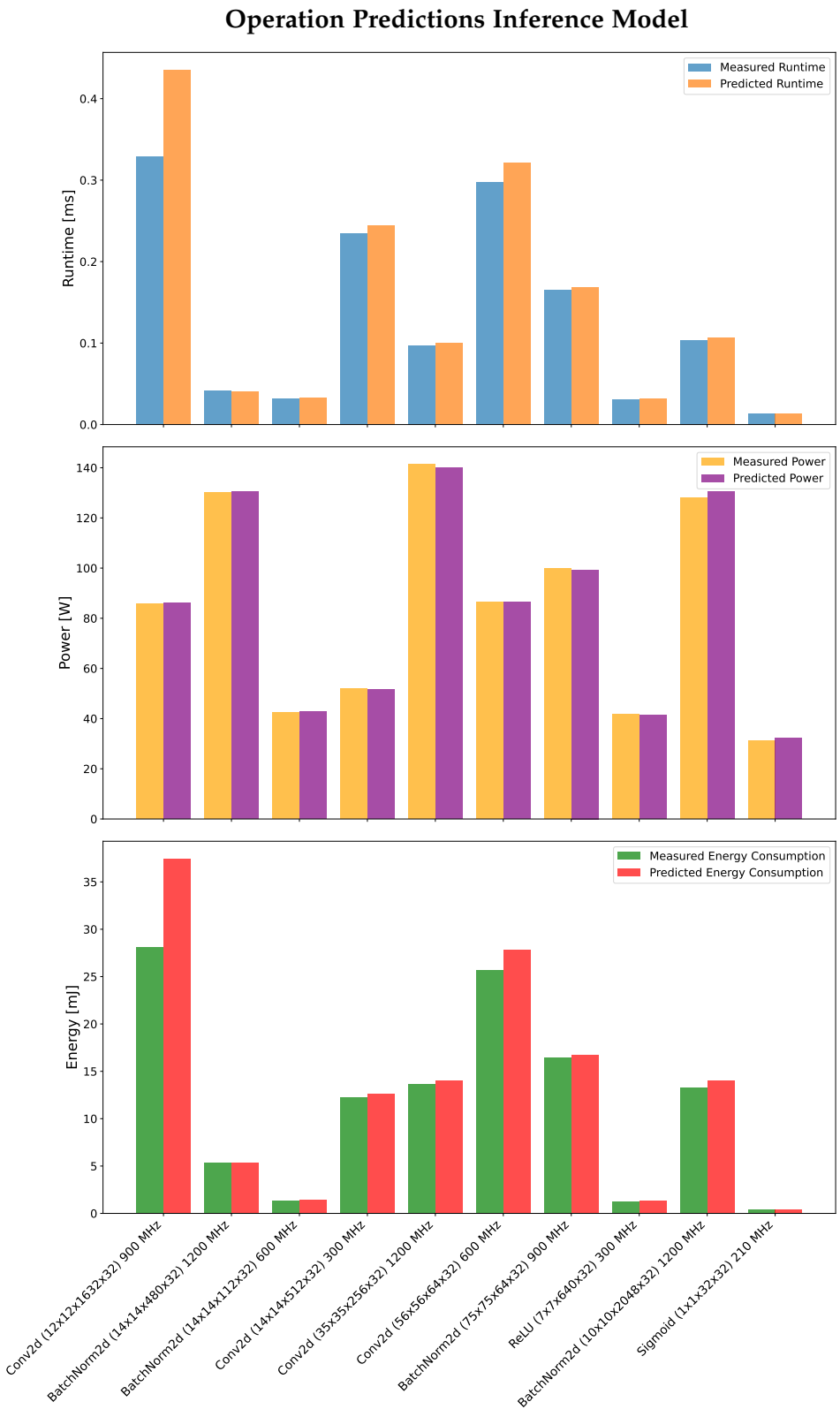

Figure 6.4: Comparison of predictions to measurements for ten operations from the test set for the A30 inference prediction model. The different orders of magnitude make the plot challenging to read, but we can see decent agreement between predictions and measurements. We do however see slight overpredictions for the resulting energy for operations 1 and 6.

6.2.3 Training RTX2080TI

The prediction model for the RTX2080TI does not provide predictions for specific clock speeds. Instead it is trained on a dataset collected running the GPU in its default configuration. The prediction performance is not as strong as for the A30. The results can be seen in table 6.3. This predictor breaks the pattern of XGBoost performing slightly better. Here we find some cases where it performs better and some where it performs worse. A possible explanation for this is the property of random forest models of generalizing pretty well, even with smaller amounts of training data. This is the case here, since this is a dataset only for the default auto clock speed. In comparison, the A30 datasets are six times as large, because they cover six different clock speeds.

Training 2080TI	**Random Forest**	**XGBoost**
CV $\overline{R^2}_{time}$	0.755 ± 0.122	0.684 ± 0.359
Test Set R^2_{time}	0.813	0.793
CV $\overline{R^2}_{power}$	0.920 ± 0.058	0.921 ± 0.054
Test Set R^2_{power}	0.866	0.869

Table 6.3: The training predictor for the 2080TI scores notably lower than the A30 predictors. It also reverses the A30 predictor trend of the XGBoost models performing slightly better. For the 2080TI predictors, XGBoost performs a little worse. The smaller training set due to the lack of a clock speed study for the A30 and the smaller focus on consistency of a consumer GPU compared to the A30 might contribute to the weaker predictor performance here. It results in a runtime score of over 0.75 and a power score of around 0.9.

6.2.4 Inference RTX2080TI

The resulting R^2 scores for our 2080TI inference predictor can be seen in table 6.4. For the runtime models we see a steep drop in R^2 score from the random forest model to the XGBoost model. But even for the better performing random forest model, this is still the worst performing predictor out of our four predictor models. This can be explained by a combination of multiple factors. It is trained on a smaller dataset, since it is not a multi clock model and it is an inference predictor, which means it has to predict smaller absolute values, which will have larger relative errors in the training set. Lastly, since the 2080TI is a consumer GPU and not a datacenter GPU, its behavior is tuned to focus on the best burst load performance, but not necessarily on the greatest consistency and stability compared to a datacenter GPU such as the A30.

Inference 2080TI	Random Forest	XGBoost
CV $\overline{R^2}_{time}$	0.735 ± 0.088	0.607 ± 0.079
Test Set R^2_{time}	0.769	0.613
CV $\overline{R^2}_{power}$	0.939 ± 0.034	0.944 ± 0.026
Test Set R^2_{power}	0.908	0.885

Table 6.4: The inference predictor for the 2080TI is our worst performer. It shares the challenges of the 2080TI training predictor, while having to work with a dataset of smaller values, which naturally have larger relative errors. The low scores of the runtime predictors using XGBoost support our decision to use the random forest model going forward. With the random forest predictors, we have a runtime score of 0.73 and a power score of 0.94.

6.3 NEURAL NETWORK LEVEL PREDICTIONS

In this next step, we will evaluate the prediction performance of the random forest model for the A30 on the neural network level.
Since this prediction model is capable of providing predictions for different clock speeds, we will conduct the validation for each clock speed. As this is a graphical validation, we will provide the measured results for each clock speed with the predicted results below. This way both the direct comparison between measurement and prediction is visible, as well as the behavior across different clock speeds.
Results are given for both runtime and energy, as well as for both inference and training. Additionally, plots of the energy delay product, defined in Equation 6.2, are provided to illustrate a trade-off between optimizing for either time or energy on their own. For the EDP, the results for each model are normalized, because the orders of magnitude between the different models are to large for this plot to remain readable otherwise.
6.5 As expected for the inference runtime measurements, the runtime decreases with an increase in clock speed for all tested models.
6.6 The corresponding inference runtime predictions manage to maintain the ordering between different clock speeds. This allows the predictions to be used to determine the optimal clock speed.
6.7 As opposed to the monotonous relationship between runtime and clock speed, the optimal clock speed for the inference energy measurements is 900 MHz.
6.8 As expected from the operations level results, we can see that the predictions for the inference energy are closer to their corresponding measurements than the inference runtime predictions are to their corresponding measurements. They also maintain the clock speed optimum found in the measurements, making these predictions suitable for energy optimizations.
6.9 For the training runtime measurements we observe the same be-

havior we saw in the inference case. Lower clock speeds lead to longer runtimes.

6.12 In the training runtime predictions, we see a similar prediction performance to the inference case. The predictions provide the correct ordering in this case too.

6.11 Identically to the inference case, the optimal clock speed in terms of training energy measurement lies at 900 MHz. The absolute energy costs for training are between 2 and 5 times as high as the inference costs.

6.12 Comparing these training energy predictions to the corresponding measurements just above illustrates how capable our predictions are at reproducing the real world behavior. These predictions are the most precise ones among the presented cases. The ordering is maintained in this case as well.

6.13 Different from the 900 MHz optimum for energy, we find the optimum for the inference measurement of the energy delay product at 1200 MHz. Being the product of runtime and energy it provides a more balanced metric to optimize along.

6.14 For the inference predictions of the energy delay product, we find larger discrepancies between measurement and prediction, than for the energy predictions. From the operations level evaluation, we know the power predictions perform better than the time predictions. The fact that time contributes to the energy delay product both as a factor in the energy and then again being multiplied with the energy prediction to form the prediction of the energy delay product, explains the observed larger discrepancies.

6.15 For the training measurements of the energy delay product, there is not a lot of change to be observed compared to the inference measurement results in the normalized plot. This indicates a roughly proportional scaling of the energy delay product between inference and training.

6.16 In the training predictions of the energy delay product, we see much larger deviations between measurement and predictions compared to the inference case. Runtime predictions are less precise than power predictions. Absolute times are several times larger for training than for inference. This effect is amplified by the squared contribution of time towards the energy delay product. This squared contribution of a larger absolute value which is less precise causes the larger deviations.

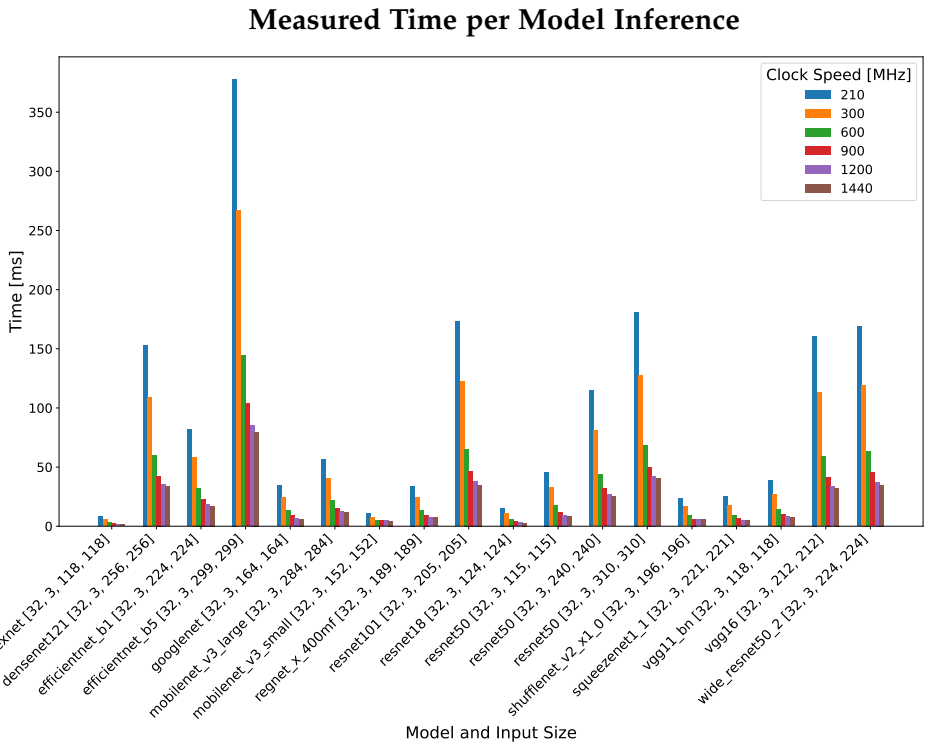

Figure 6.5: Inference time measurements for 18 model-input sets across six clock speeds.

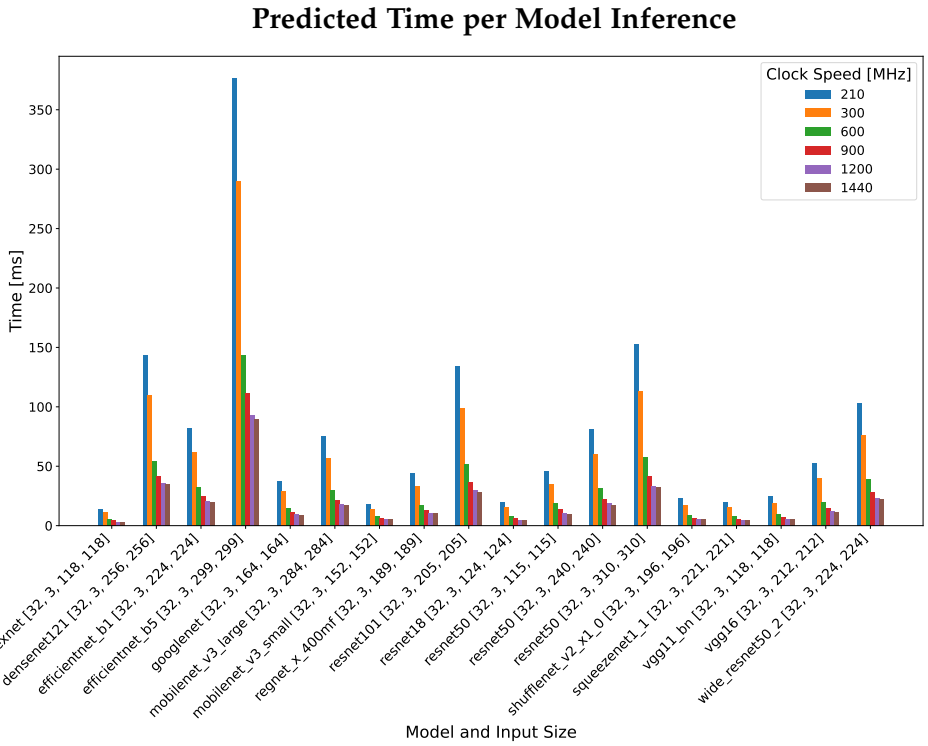

Figure 6.6: Inference time predictions for 18 model-input sets across six clock speeds.

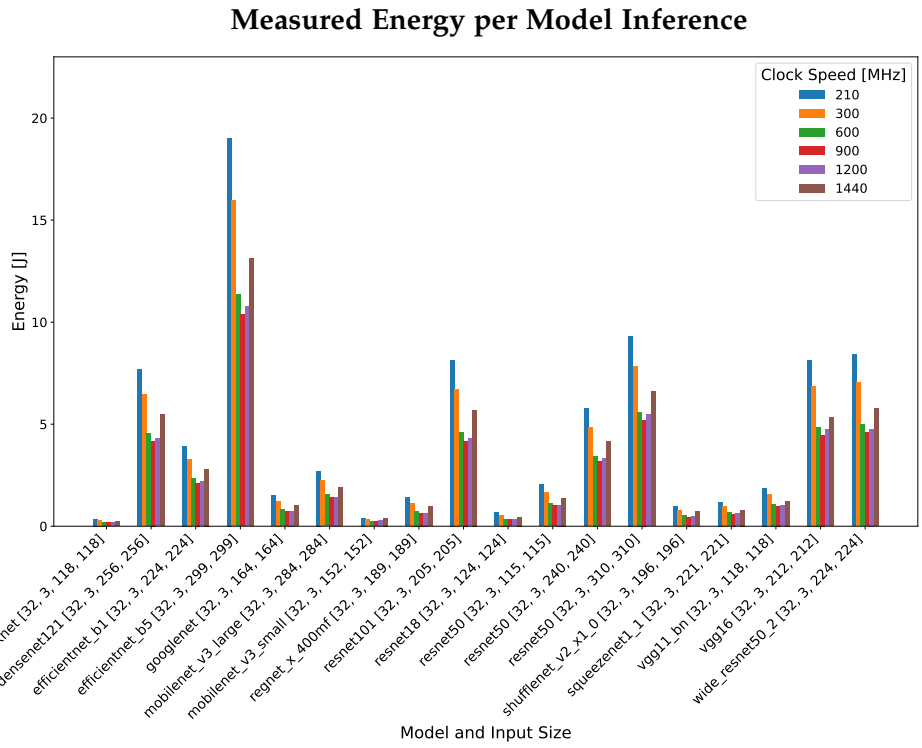

Figure 6.7: Inference energy measurements for 18 model-input sets across six clock speeds.

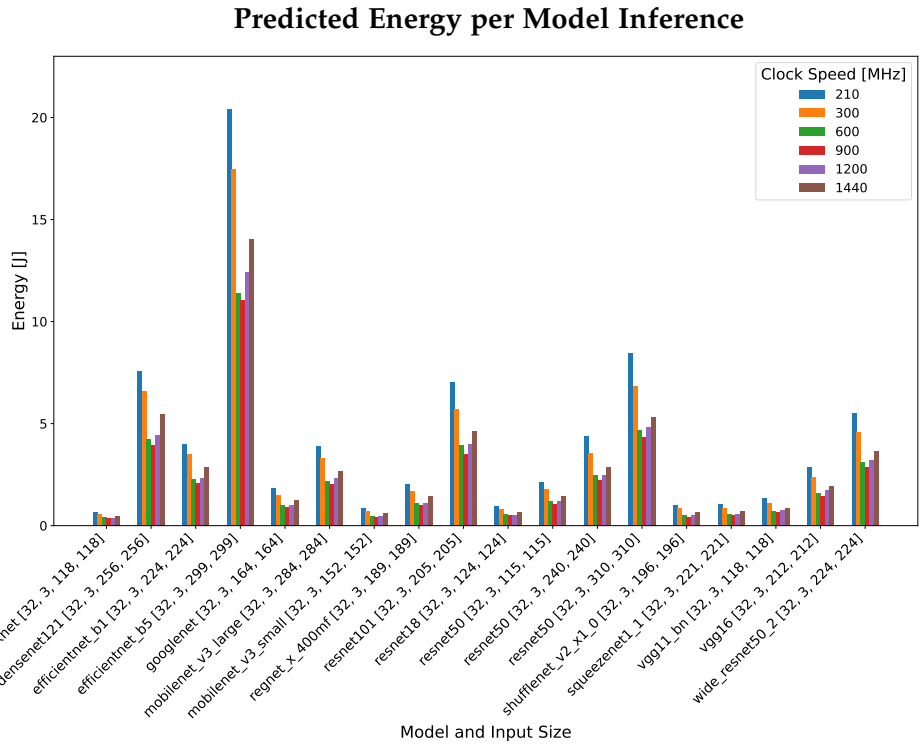

Figure 6.8: Inference energy predictions for 18 model-input sets across six clock speeds.

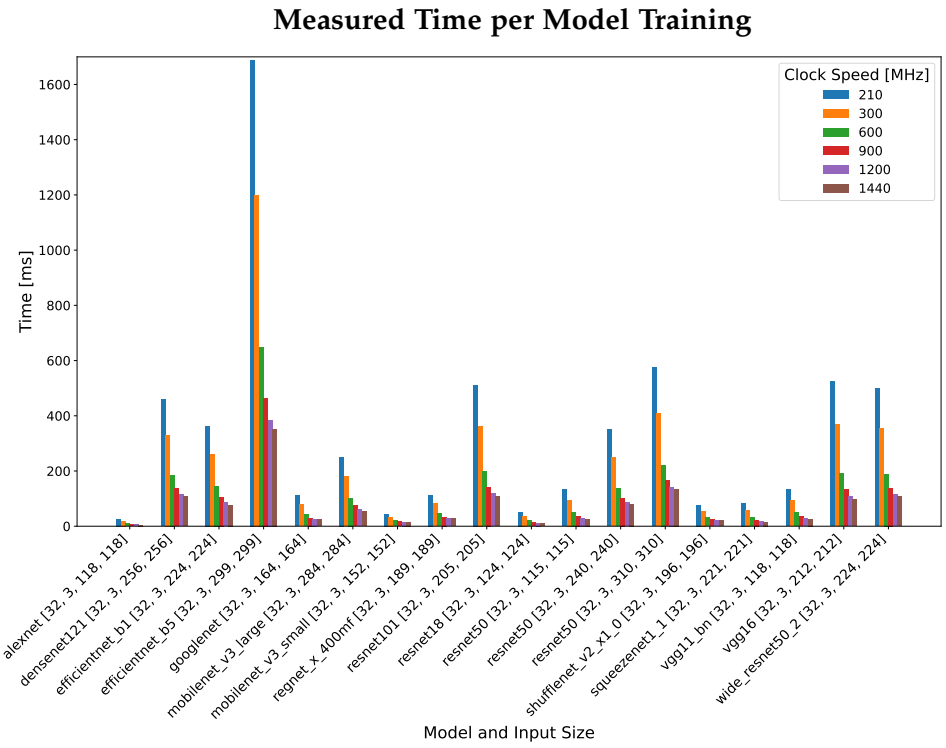

Figure 6.9: Training time measurements for 18 model-input sets across six clock speeds.

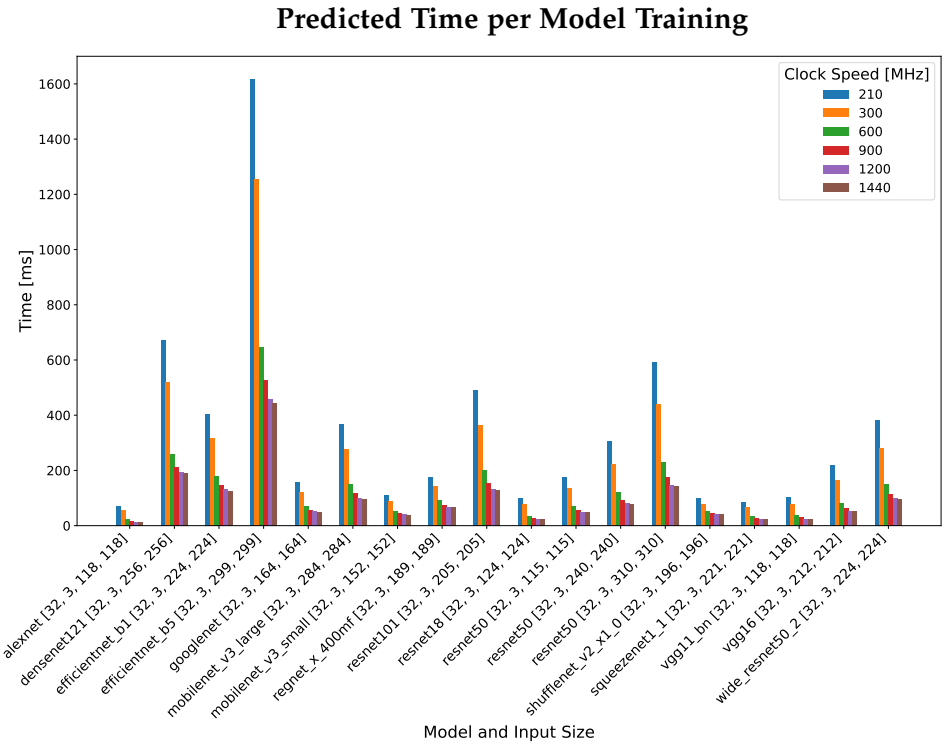

Figure 6.10: Training time predictions for 18 model-input sets across six clock speeds.

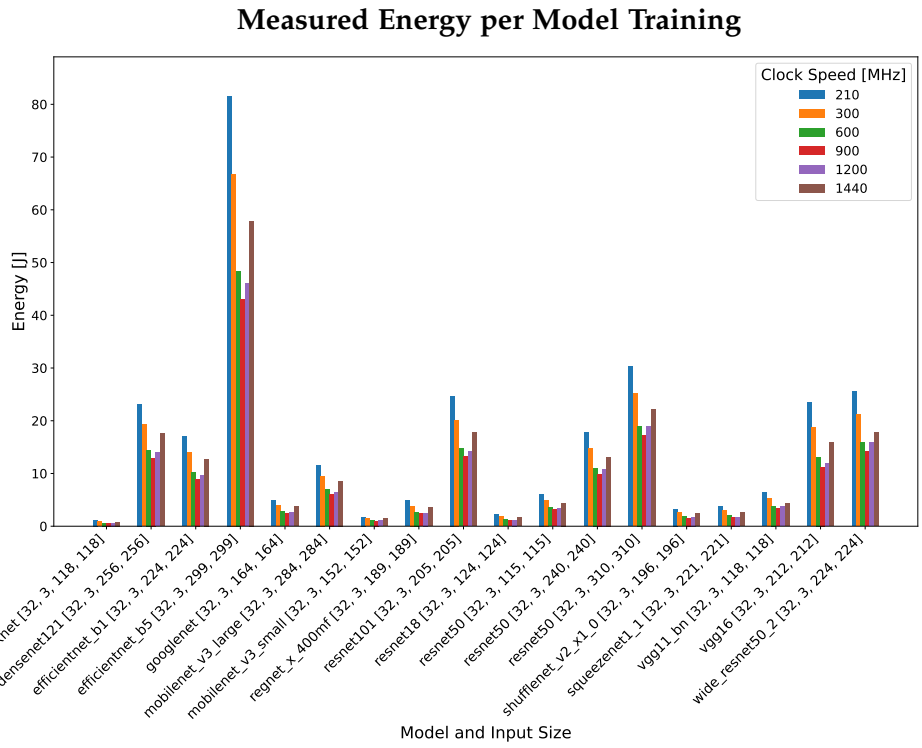

Figure 6.11: Training energy measurements for 18 model-input sets across six clock speeds.

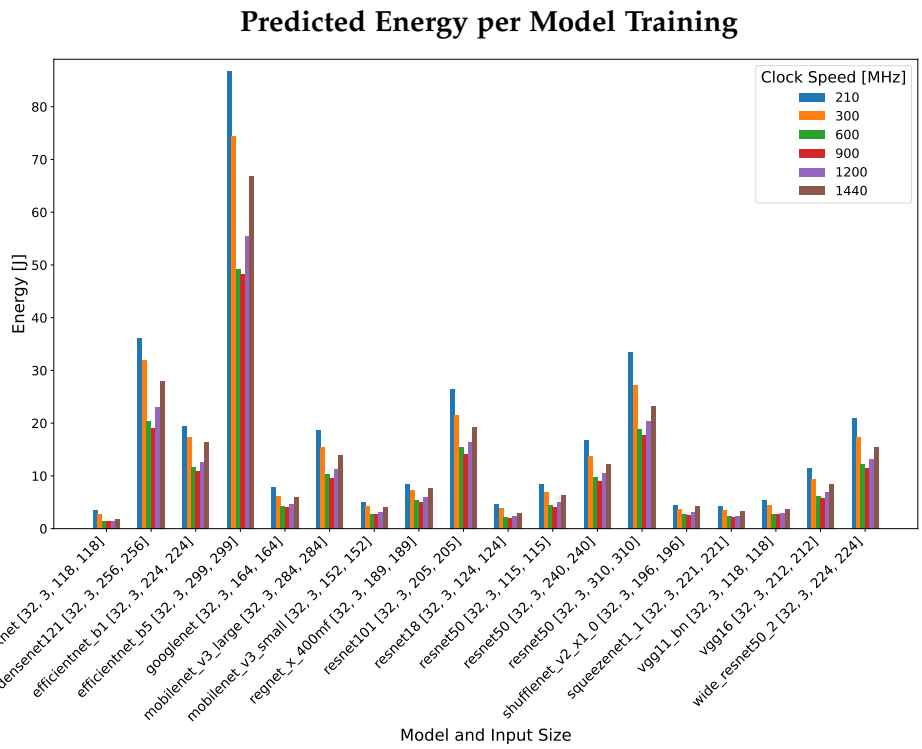

Figure 6.12: Training energy predictions for 18 model-input sets across six clock speeds.

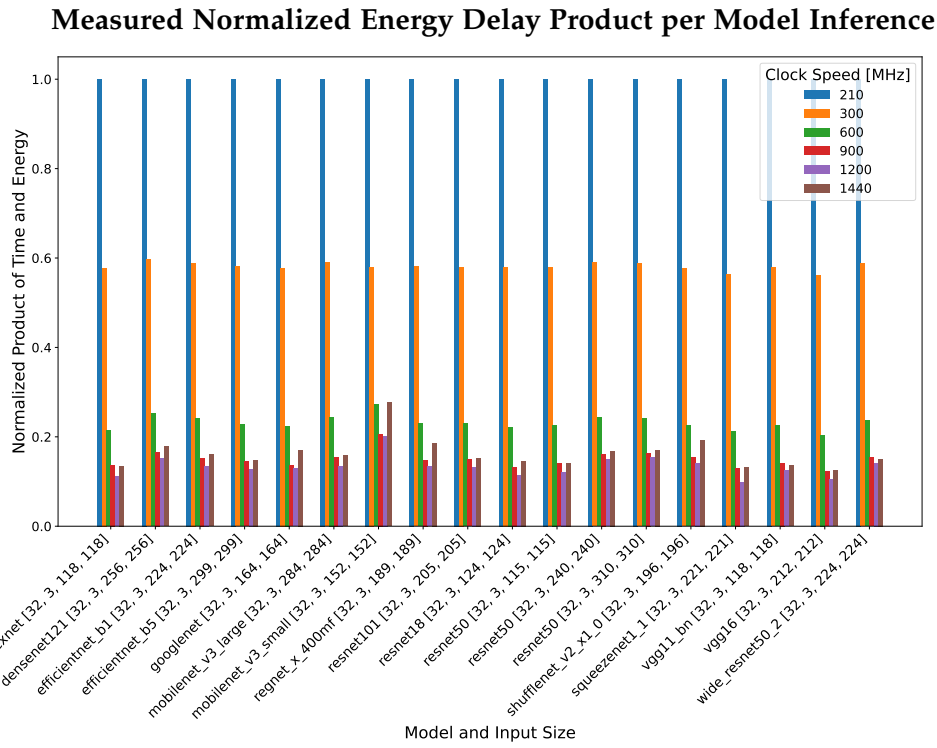

Figure 6.13: Inference energy delay product measurements for 18 model-input sets across six clock speeds.

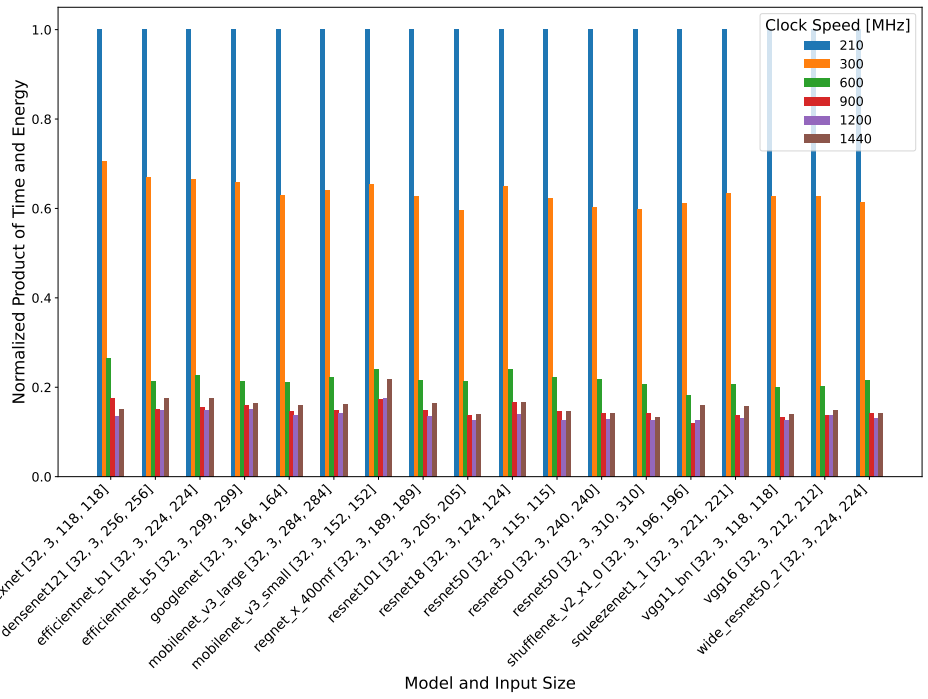

Figure 6.14: Inference energy delay product predictions for 18 model-input sets across six clock speeds.

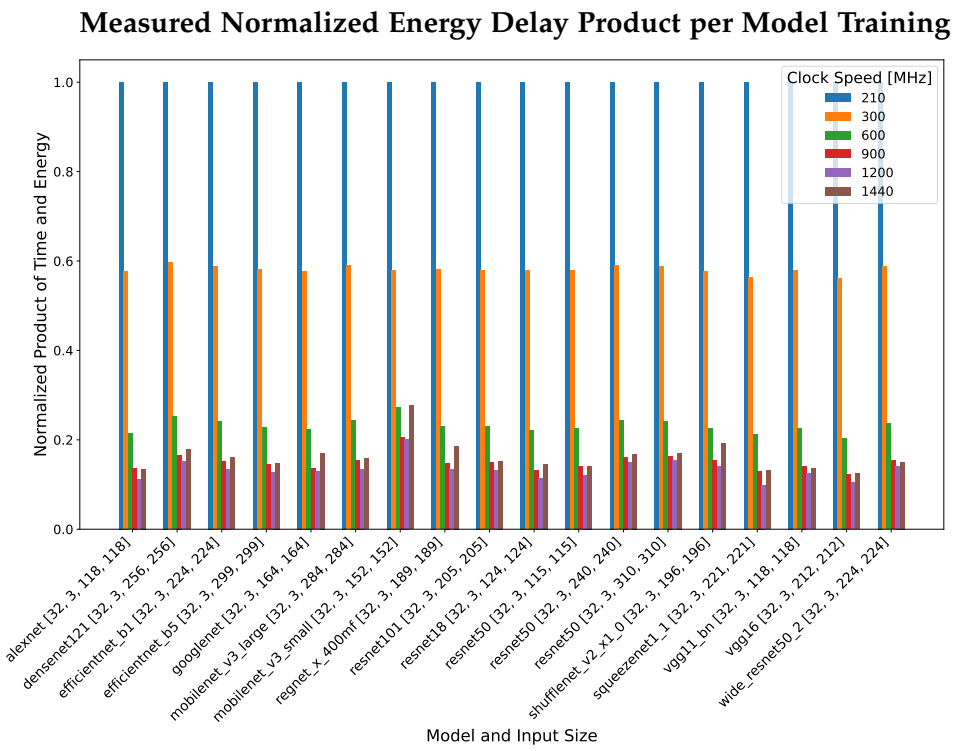

Figure 6.15: Training energy delay product measurements for 18 model-input sets across six clock speeds.

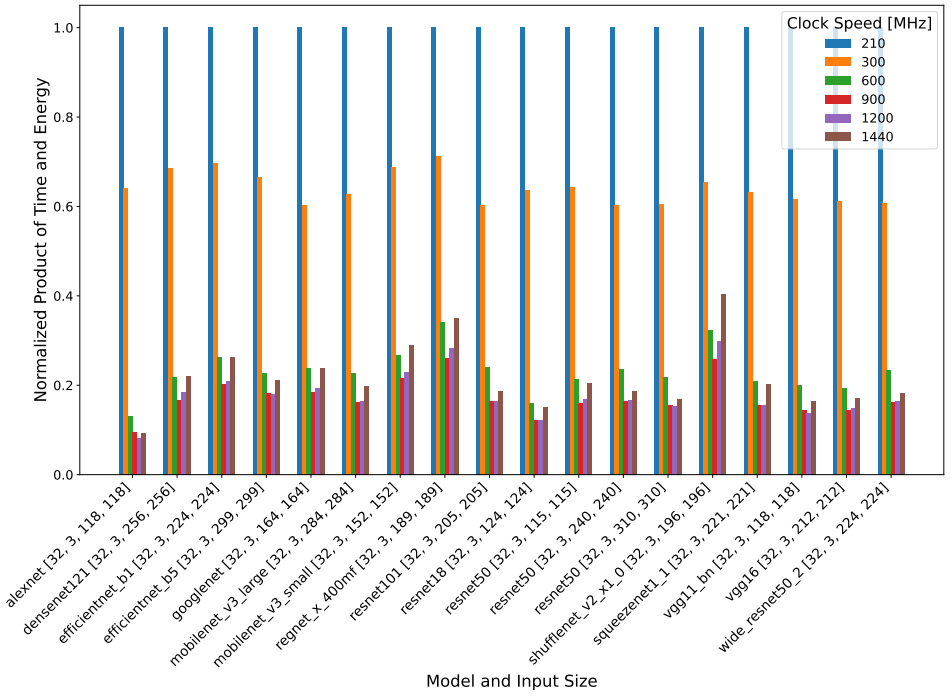

Figure 6.16: Training energy delay product predictions for 18 model-input sets across six clock speeds.

6.3.1 Patterns and Observations

Given all our profiling data on the A30, we studied the results in search for the emergence of interesting patterns.
We were not surprised to find higher clock speeds resulting in lower runtimes, but apart from this obvious causality we did not find any other curious patterns in our execution time profiling.
We did however find an interesting pattern in our energy profiling. Both for the training and the inference case we observed a "U" pattern across the clock speeds. Figure 6.17 shows very high energy costs for very low clock speeds and high energy costs for very high clock speeds. We find the energy minimum at a more moderate clock speed of 900 MHz. This pattern remains stable across both training and inference and for all tested models.
The fact that the highest clock speeds are not the most efficient choice is explained by the power consumption formula for transistors [5].

$$P \propto \frac{1}{2} C V^2 f \tag{6.1}$$

Where P is the power, C is the capacitative load, V is the voltage and f is the frequency.
For a whole GPU this is scaled by its number of transistors. The proportionality remains intact. In order to maintain stability at the highest clock speed settings, it is necessary to increase the voltage. The squared contribution of the voltage towards the power consumption makes it very sensitive to larger voltages. This is the reason why the highest achievable clock speed is rarely the most efficient one.
We do however observe high energy costs for low clock speeds as well. This is where another effect comes into play. Every GPU has an idle power consumption that is independent of the current workload. This means that because a GPU running a workload at a very low clock speed will take a long time to complete it, a larger fraction of the energy cost will be caused by its idle power consumption.
These two counteracting effects result in the energy optimum lying at neither extreme. So even if a future DNN had a slightly different optimum, the optimum we found here would be a good starting point when optimizing for energy efficiency.
While energy efficiency favors moderate clock speeds, in practice we often require a different balance between energy and runtime optimization. The energy delay product defined in Equation 6.2, quantifies the tradeoff between energy and runtime by squaring the time contribution. Optimizing for the energy delay product shifts our optimum towards a higher clock speed. In Figure 6.13 and Figure 6.15 we observe a similar "U" pattern in our EDP profiling results, with its optimum at 1200 MHz across both inference and training.

The shift of the optimum towards higher clock speeds is explained by the definition of the EDP:

$$\text{Energy Delay Product} = P \cdot t^2 \tag{6.2}$$

Where P is the power and t is the time.
The optimum is still the result of the same two counteracting effects. However, since the contribution of the time is squared, the optimum moves to a higher clock speed.
Our predictions do not reproduce this pattern perfectly for the training case. Since the runtime has a squared contribution to the EDP and the absolute values for training are larger, this combination leads to a scenario where our prediction precision for the runtime becomes insufficient to predict the correct optimum.

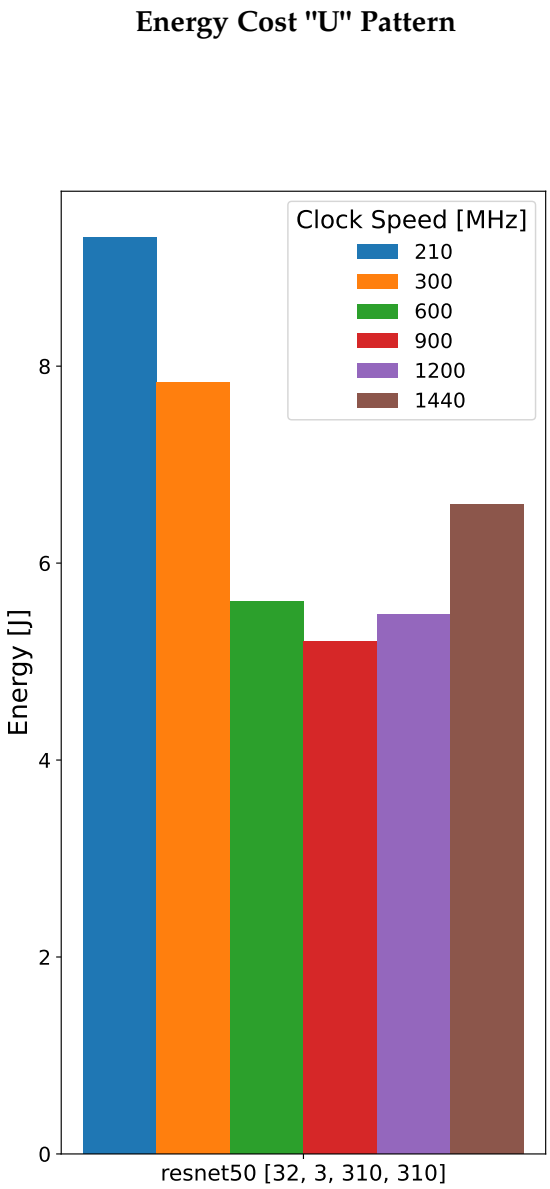

Figure 6.17: Looking at the energy measurements across different clock speeds for a single model, the "U" pattern which is present for all models becomes even more apparent. In terms of energy cost it is most efficient to run both inference and training tasks at a clock speed of 900 MHz for all our tested models.

6.4 PREDICTOR LATENCY

Other than the predictor performance in terms of accuracy, its performance in terms of latency is also a concern. The loading of the dataset turned out to be a very time intensive operation. Even after improving the performance, we arrived at a loading time of around 30 seconds for the full set of models used to validate the predictor. These 30 seconds are in a completely different order of magnitude in terms of execution time compared to preprocessing, prediction of the models or even execution of the models. Realistically, this time is more comparable to having to download a model before being able to benchmark it. The next largest contributor in terms of execution time is preprocessing the dataset in order to format is in a way that can be used as an input for our predictor model. In its current form this step takes somewhere around 500 ms. For the full validation set of 18 model-input sets it takes around 600 ms. For an individual model it takes less than 500 ms.

The most important evaluation in terms of predictor execution time is the execution time of the actual predictor model. Predicting time and energy at a specific clock speed for all model-input sets in the validation set shows very good performance. Across multiple clock speeds and for both for the training predictor and the inference predictor the resulting execution time hovers around 30 ms. Even for the scenario with the smallest execution time, inference at maximum clock speed, this is still faster than the collective execution time of around 300 ms of the validation set by one order of magnitude. This comparison shifts only further in favor of the predictor model when we look at a still very reasonable scenario of training at a clock speed of 930 MHz. With this the collective execution time is close to 2000 ms, while the predictor time is not affected. Here we have a predictor latency over 60 times lower than performing the measurement instead.

7 DISCUSSION AND OUTLOOK

7.1 DISCUSSION

The approach we took in this work was of an exploratory nature. Rather than diving deep into one specific configuration, we chose to go wide and look at multiple hardware and many neural network scenarios. That approach turned out to be both a blessing and a curse. A blessing, because the results can be used in a wide field of applications and our findings are understandable at a more abstract level. A curse, because we have to hamper our scientific curiosity not to follow every rabbit hole we encounter along our journey. Given the wide approach, we also discover a wide range is troubleshooting issues which vary in their importance to the study itself. This balancing act of weighing the cost and benefit of following up on interesting trends and deciding whether or not to invest the time to overcome the troubleshooting obstacles was the most difficult part of this work. In order to understand the choices we made concerning the scope, it is important to see that even in our quest to go wide, it lies in the nature of exploratory research, that no state of completeness can ever be achieved.

7.1.1 Limitations

In order to go wide, one would like to include a large number of hardware scenarios and study all of them with the same set of accurate tools. Ideally, we would like to include GPUs from Nvidia, AMD and Intel alongside FPGAs and other ML accelerators. But the platforms and tools are too varied, such that even if we found tooling which was capable of measuring power for all of the above, we would likely give up both time and power resolution in exchange for that improved compatibility. Another simple but very important limitation is the hardware available to us. Both because of these considerations as well as in order to keep the scope of this thesis in check, we decided to limit ourselves to Nvidia GPUs.
Another dimension to go wide in, is the plurality of neural networks. Here our decision was determined by our platform of choice. Due to prior experience with the platform we chose PyTorch. In order to use well known networks and implementations, we decided to work with neural networks from the torchvision library. This also improves the easy of reproducing our study. Our GPU with the smallest amount

of global memory added a limit to which model-input sets we were able to include in our suite profiled for the training set, since each model-input set had to be able to complete all benchmarks on all hardware configurations.
Our choice to study both time and energy for inference and training was shaped by the desire to study a novel section of the field and to add value to our research.
After initial attempts to include both GPUs in the clock speed study, the 2080TI was dropped from this part of the overall study after running into issues.

7.1.2 Application

The predictor models for the A30 and the 2080TI only differ in the A30 predictor including clock speed specific predictions. Apart from that they are built identically. In our evaluation of which type of predictor model we want to use, we are therefore looking for the model type that is the most stable across all our scenarios. This will enable it to serve as a recommendation suitable for various settings of neural networks and hardware.
With that in mind, let us recap our predictor results. For the 2080TI, the evaluation of the R^2 score showed similar or better performance using the random forest model over the XGBoost model for both the training and inference predictors. The evaluation of the A30 predictor revealed slightly better scores using the XGBoost predictor. However, the improvements over the random forest model were much smaller than the drop-off observed for the 2080TI predictor going from random forest to XGBoost. This behavior leads us to believe that the random forest approach serves as a more stable and consistent path across the plurality of hardware configurations this methodology might encounter in the future.
Earlier in this discussion it was mentioned that this is an exploratory work. In the same way it is also a foundational work. Each contribution aids to explore which paths can be taken and helps building a foundation of methodology and due diligence. Our work provides the tools for the collection of a required dataset and demonstrates the resilience of the resulting dataset through a direct validation. Since the predominant interest in the research community lies in finding insights regarding complete neural networks, the nature of our validation ensures our findings are applicable to full neural network executions. For the same reasons, the A30 predictor is evaluated on a neural network level. Operations level results might be interesting for a specific academic niche, but they can only fulfill their potential when they are also applicable to full neural networks.
While the primary reason for these evaluations and validations is to establish the capabilities and limitations of the specific predictors

trained in this work, they also serve to establish the soundness of the approach and methodology on a broader level, demonstrating their suitability as a base for future work.

7.2 OUTLOOK

The nature of this work opens up a number of avenues for future work. In light of our goal to guide towards the use of the best fitting hardware for specific tasks and requirements, the most promising avenue is to expand the study to a larger number of hardware platforms. The lowest friction way of doing that would be to collect datasets for further Nvidia GPUs at their default clocks and adding the GPU model as a parameter to the prediction model, in the same way the clock speed was added for the A30 predictor.
Another direction could be an attempt to improve the prediction performance. A good starting point would be the expansion of the set of model-input sets which were used to build the training set for the predictors in this work. Including operations from more model-input sets, especially types of models not included in this work, could go a long way towards improving the predictor's accuracy and generalizability beyond its current state.
Even though we limited our GPU clock study to the A30 in this work, there is no conceptual reason why these kinds of clock speed studies could not be expanded to a more GPUs, as long as they support setting the clock speed manually. This kind of study always carries the chance of discovering interesting patterns and behaviors.
Another avenue for expansion are the metrics included in the study. The metrics used in our work were fixed from a very early point onward. It covers time, power and energy. A prime candidate for the next metric to add would be the memory usage.
The complexity in the addition of further metrics lies in the tooling. The more tools are used, the harder expansion towards additional hardware platforms becomes.
The last and widest avenue for expansion is the inclusion of more hardware platforms. Our current tooling for power readout is Nvidia specific, but provided equivalent tools, an expansion towards GPUs from different manufacturers like AMD and Intel would be very interesting. Combined with real-time pricing this would allow determining the most cost effective GPU for a specific task including power costs across the entire GPU market.
The last and most difficult step is moving from GPU studies towards the inclusion of CPUs and more exotic accelerators like FPGAs and IPUs. This step will be the most limiting to our selection of suitable metrics, since they all need to be meaningfully applicable to all included hardware platforms. Expanding the number and types of

hardware platforms is the most fascinating avenue for future work, but it is also the one moving the furthest away from the foundation presented in this work and the least predictable in its development.
In summary, there are many opportunities for future expansion upon the basis presented here. Some of them will require a lot of additional work, while others are direct continuations, which should present minimal friction.

7.3 CONCLUSION

We identified the research gap as the lack of works covering performance predictions for runtime and energy for both training and inference workloads. In order to address this, we designed our profiling and prediction frameworks to cover time, power and energy for both inference and training operations.
Our first contribution covers the profiling part of this work. Here we presented our method for collecting a dataset of individual operations which serves as training data for our predictor. The second contribution presents our choice of the random forest model for our predictor, as well as the preprocessing of the dataset. Our third and largest contribution is focused on validating and evaluating the work product of the first two contributions. It contains a validation of the dataset quality through comparisons between direct measurements of full neural network runs and summed up operations-level profiling results. The random forest predictor models are evaluated via R^2 score with both cross-validation and test set performance on the operations level. On the neural network level, a set of unknown model-input sets is used to validate the predictor performance by comparing direct measurements of the full neural network runs to the summed up operations-level predictions. Profiling over a number of clock speeds also yielded the insight, that 900 MHz is the most energy efficient setting on the A30 across all model-input sets used for the validation. With R^2 scores of 0.7 to 0.9 for runtime predictions and 0.90 to 0.98 for power predictions on the operations level, we already contribute a tool which can be of considerable help in deciding which GPU and which clock speed to run a specific model at. Additionally the raw profiling data itself bears the potential of providing insight into useful patterns like the most energy efficient clock speed or the optimal clock setting for a minimal energy delay product.
Our work serves as an exploratory and foundational step into the profiling and prediction of an ever broadening zoo of parameter combinations between models, inputs, hardware platforms, clock settings, metrics and workload types. And while we contribute valuable findings and tools towards that goal, the available research potential in this field is far from exhausted and we hope our findings and contributions

can help to pave the way and guide future researchers in search of that same goal.

BIBLIOGRAPHY

[1] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. "NeuralPower : Predict and Deploy Energy-Efficient Convolutional Neural Networks." en. In: ().

[2] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." en. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Francisco California USA: ACM, Aug. 2016, pp. 785–794. DOI: 10.1145/2939672.2939785. URL: https://dl.acm.org/doi/10.1145/2939672.2939785 (visited on 07/17/2025).

[3] Eugenio Gianniti, Politecnico di Milano, and Li Zhang. "Performance Prediction of GPU-based Deep Learning Applications." en. In: ().

[4] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. eng. Second edition. Springer series in statistics. New York, NY: Springer, 2009. ISBN: 978-0-387-84858-7. URL: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=277008 (visited on 07/17/2025).

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th. Burlington, MA, USA: Morgan Kaufmann Publishers Inc., 2017.

[6] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. "Predicting the Computational Cost of Deep Learning Models." In: *2018 IEEE International Conference on Big Data (Big Data)*. Dec. 2018, pp. 3873–3882. DOI: 10.1109/BigData.2018.8622396. URL: https://ieeexplore.ieee.org/document/8622396/?arnumber=8622396 (visited on 11/27/2024).

[7] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. *A Learned Performance Model for Tensor Processing Units*. en. arXiv:2008.01040 [cs]. Mar. 2021. DOI: 10.48550/arXiv.2008.01040. URL: http://arxiv.org/abs/2008.01040 (visited on 02/05/2025).

[8] Hang Qi, Evan R Sparks, and Ameet Talwalkar. "PALEO: A PERFORMANCE MODEL FOR DEEP NEURAL NETWORKS." en. In: (2017).

[9] Max Sponner, Bernd Waschneck, and Akash Kumar. "AI-Driven Performance Modeling for AI Inference Workloads." en. In: *Electronics* 11.15 (Jan. 2022). Number: 15 Publisher: Multidisciplinary Digital Publishing Institute, p. 2316. ISSN: 2079-9292. DOI: 10.3390/electronics11152316. URL: https://www.mdpi.com/2079-9292/11/15/2316 (visited on 11/27/2024).

[10] Delia Velasco-Montero, Jorge Fernández-Berni, Ricardo Carmona-Galán, and Ángel Rodríguez-Vázquez. "PreVIous: A Methodology for Prediction of Visual Inference Performance on IoT Devices." In: *IEEE Internet of Things Journal* 7.10 (Oct. 2020). Conference Name: IEEE Internet of Things Journal, pp. 9227–9240. ISSN: 2327-4662. DOI: 10.1109/JIOT.2020.2981684. URL: https://ieeexplore.ieee.org/document/9040398/?arnumber=9040398 (visited on 02/06/2025).

[11] Chuan-Chi Wang, Ying-Chiao Liao, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. "PerfNet: Platform-Aware Performance Modeling for Deep Neural Networks." en. In: *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. Gwangju Republic of Korea: ACM, Oct. 2020, pp. 90–95. ISBN: 978-1-4503-8025-6. DOI: 10.1145/3400286.3418245. URL: https://dl.acm.org/doi/10.1145/3400286.3418245 (visited on 02/05/2025).

ERKLÄRUNG

Ich versichere, dass ich diese Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den Day/Month/Year Here

_____

Constantin Nicolai