Computationally Efficient Deep Learning Training: Investigating the Impact of Hyperparameters

Christian Burmester [a], Maximilan Kalcher [b], Marlena Reil [c]
University of Osnabrück, Germany
[a]jburmester@uos.de
[b]mkalcher@uos.de
[b]mareil@uos.de

April 2, 2023

Abstract - Optimizing Deep Learning training is accomplished by tuning and adapting the hyperparameters of a given model. Experimenting with the hyperparameters and consecutively training the model often is time intensive and draws a lot of GPU power. In this study, we built a pipeline to evaluate the effect of certain hyperparameters on the GPU power draw while not disregarding the model's performance. While the scope of the training did not allow us to generate statistically relevant results, we were able to trace first patterns that would favor some parameters over others and enable a more computationally efficient hyperparameter selection. A final training run with the top parameters in terms of energy efficiency yielded the highest energy efficiency score, demonstrating the potential of the suggested framework. Our automated pipeline allows support of future research and further investigations, even with other Deep Learning models.

Keywords: Energy Efficiency; Hyperparameter Optimization; Deep Learning

1 Introduction

Research in the field of Deep Learning (DL) has mainly focused on improving model performance toward various tasks and problems. As computing power continues to improve, it becomes possible to build and train larger and more complex DL networks. However, training these networks can be computationally intensive, requiring large amounts of processing power and memory. According to a study by Struwell *et al.*, training a single language model approximately generates the carbon dioxide emissions produced by a trans-American flight (Strubell, Ganesh, and McCallum (2019)). This is just one example of the significant energy consumption associated with DL models, which has been documented in various studies (Li, Chen, Becchi, and Zong (2016); Sevilla et al. (2022); Sharir, Peleg, and Shoham (2020); C.-J. Wu et al. (2022)).

Attempting to push towards more energy efficient DL training, some work has investigated methods to estimate energy consumption of DL models (García-Martín, Rodrigues, Riley, and Grahn (2019); Yang, Chen, Emer, and Sze (2017)). Wolff Anthony *et al.* developed a tool for tracking the carbon footprint of models (Anthony, Kanding, and Selvan (2020)). Other work achieved improvements through model compression (Cheng, Wang, Zhou, and Zhang (2017)) or conditional DL (Panda, Sengupta, and Roy (2017)).

Our work is directed at hyperparameter optimization, the task of finding the optimal hyperparameter setting of a model. Mostly performed automatically, hyperparameter optimization effects energy consumption in two ways: (1) the requirement for retraining in the searching process multiplies the amount of computational cost; and (2), the final choice of hyperparameters influences energy usage of subsequent model training and testing.

Research has focused on improving algorithms for the search process (Chowdhury, Hossen, Azam, and Rahman (2022); Parsa, Ankit, Ziabari, and Roy (2019)). Previous work has shown that computationally expensive methods like grid search, which exhaustively tests combinations (Larochelle, Erhan, Courville, Bergstra, and Bengio (2007); LeCun, Bottou, Bengio, and Haffner (1998)), can be replaced by random search, that is more efficient while performing sufficiently well (Bergstra and Bengio (2012)). In random search, combinations are selected at random until a predefined stopping criterion is reached.

Regardless of the search process, optimization results that have been optimized for performance vary widely in terms of energy consumption (Young et al. (2019)). However, to the best of the authors' knowledge, no approach so far has considered frameworks that account for selecting more efficient hyperparameter settings using a random approach.

To extend research in this area, we created an automated framework for evaluating hyperparameter configurations in terms of energy usage and accuracy, and explicitly incorporated an energy efficiency quotient. We looked at standard hyperparameters such

as pre-processing and augmentation methods, batch sizes or different optimizers. Our pipeline structure is inspired by a sweeps approach from Weights & Biases WeightsBiases (n.d.), which we extended by tracking GPU power consumption during training. We used random search as efficient, easy-to-implement search strategy. Beyond our experimental research, we aim to provide the research community with a method to help identify critical parameter settings and drive optimization research in a more resource-conscious direction.

A. Context of this paper.
This work was produced as part of the study seminar "*Implementing ANNs with TensorFlow*" at Osnabrück University, supervised by Leon Schmid.

B. Contribution of this paper.
We developed an automated, easy-to-run approach for hyperparameter selection, that involves random selection of parameter samples, while including all stages of model development (pre-processing, training, post-processing). It evaluates not only performance but combines accuracy and GPU power in a coefficient to measure energy consumption of the training procedure. The approach can be used for training any standard model in the keras library.

C. Organization of this paper.
In section 2, we describe the experimental setup and provide details about the hyperparameters and evaluation methods. We present our results in section 3. Section 4 gives an outlook and discusses potential challenges and shortcomings. We summarize our findings in section 5. In the Appendix, section 6, we provide an overview of our implementation.

2 Methodology

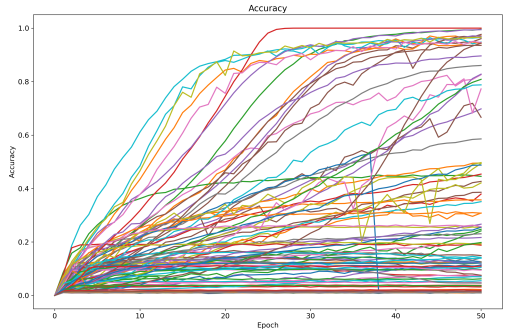
In this section we will present the methods used in our project. Figure 2 shows the pipeline we designed for efficiently experimenting with the training setup. The pipeline automatically logs GPU power draw, loss and accuracy values for each run that were later analyzed. We chose a standard model and dataset for our training pipeline to provide comparability for future work.

2.1 Training configurations

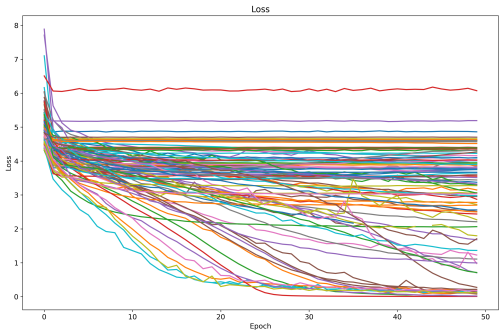
Training with random configuration started by sampling a random set parameter values, one parameter per section, from the options outlined in section 2.5. Every training run contained a combination of 10 different parameters. Some parameters configured the pre-processing of the data, others concerned the model building. We trained in total 100 runs with random configuration. All models were trained in one session, using an NVIDIA RTX 3090 on cloud provider `vast.ai`. The training concluded after 7 days and 16 hours.

2.2 Model

ResNet-50 is a convolutional neural network that was introduced in 2015 (He, Zhang, Ren, and Sun (2015)).



(a) The accuracy plots of all 100 runs.



(b) The loss plots of all 100 runs.

Figure 1: An overview of all training runs.

With 50 layers of depth, it is categorized as very deep network compared to many other popular models. According to a study by He et al. (2015), it is commonly used for image classification tasks in research and industry and has achieved state-of-the-art performance on many benchmarks. Therefore, it provided a good point of comparison for the task at hand.

2.3 Dataset

The CIFAR-100 dataset is a commonly used benchmark dataset in the field of machine learning. It consists of 50,000 training images, each of size 32×32 pixels, which are divided into 100 classes, with 500 images per class. The 100 classes in the CIFAR-100 dataset are divided into 20 superclasses, each containing 5 classes.

The CIFAR-100 dataset is often used as a benchmark dataset for image classification tasks because it is challenging and reflects the complexity of real-world image recognition problems. It is particularly useful for evaluating the performance of DL models.

2.4 Fixed parameters

In the following we list three parameters that were kept the same across all training runs in order to ensure comparability.

Epochs The number of epochs used for training was kept constant at 50. This was chosen as it was found to be just enough for the model to converge and perform well on the validation set, without overfitting signifi-

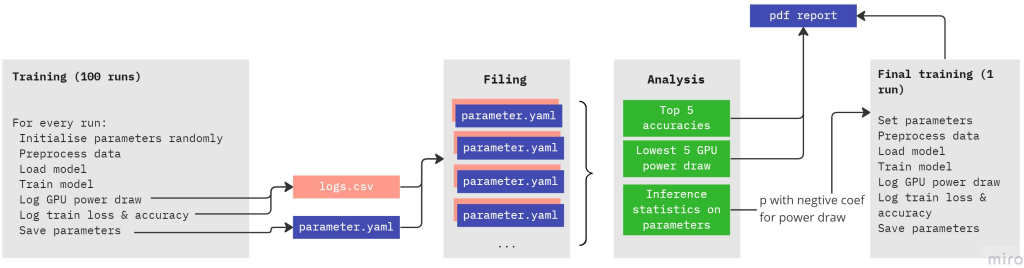


Figure 2: Training pipeline to randomly initialize parameter configurations, pre-process the data, building and training the model, and compiling the results in a PDF report after analysis.

cantly. This was also small enough to keep training times reasonable.

Loss The categorical cross-entropy loss was used for all models in the experiments. This is the standard choice for multi-class classification problems, and has been found to work well for neural network models.

Dataset size The size of the dataset before splitting was kept constant at 50,000 images. This was chosen as it was large enough to enable the model to learn good generalizations.

2.5 Variable parameters

In the following, we present the parameters that changed randomly for each training run, with different combinations.

Pre-processing The different pre-processing steps used were: no pre-processing, standardization, robust scaling, and MinMax scaling. These pre-processing steps are common techniques to scale the input data to a specific range. Several studies including the work of Bengio, Simard, and Frasconi (1994) have shown that this reduces the impact of outliers leading to better model performance.

Augmentation The different augmentation techniques used were: no augmentation, random augmentation, MixUp, and CutMix. The general goal of these augmentation techniques is to artificially expand the dataset by creating new samples or variations of existing samples. Cubuk, Zoph, Mane, Vasudevan, and Le (2019) have found that using augmentation techniques can lead to significant improvements in model performance, particularly when the training dataset is small or the model is prone to overfitting.

Precision Precision in DL refers to the number of bits used to represent a floating-point number. The precision of these floating-point numbers can have a significant impact on the accuracy and efficiency of the model as shown by Gupta, Agrawal, Gopalakrishnan, and Narayanan (2015). We casted the dataset to the following data types: float16, float32, float64, and global policy float16.

Batch size The different batch sizes used were: 4, 32, 64, and 128. Batch size is the number of training samples used for each training iteration. A larger batch size will generally result in faster training, but too large a batch size can lead to the model overfitting and not generalizing to the test set. Keskar, Mudigere, Nocedal, Smelyanskiy, and Tang (2016) investigated the impact of using different batch sizes on the performance of DL models. They show that larger batch sizes indeed can result in faster training times, but can also lead to overfitting and decreased generalization performance. They also found that larger batch sizes can result in training in "sharp minima," which can mitigate the performance of the model on the test set. The authors suggest that using smaller batch sizes can lead to better generalization and avoid sharp minima.

Data partitioning The different data partitioning used were: 60-20-20, 70-15-15, 80-10-10, and 90-5-5. These are the ratios of the dataset which are allocated to training, validation and test sets respectively. Tuning the partitioning ratios of the dataset can improve the performance of the model by accounting for the dataset size, complexity of the model, and the computational resources available. This was addressed and shown by Pereira, de Sá, and Alexandre (2017) who found that using a larger validation set can lead to better generalization and improved performance on the test set.

Learning rate The different learning rates used were: 0.01, 1.5e-4, 8.0e-4, and 6.25e-3. The learning rate is the rate at which the model's parameters are updated during training, and is an important hyperparameter to tune. Hinton, Srivastava, Krizhevsky, Sutskever, and Salakhutdinov (2012) suggest that setting a high learning rate can lead to overfitting and reduced generalization, while using a lower learning rate can result in slower training, but improved generalization. 0.01 is the baseline learning rate for the SDG (Stochastic Gradient Descent) optimizer, as for the Tensorflow documentation. The other values were taken from the work of (Woo et al. (2023)).

Learning rate schedule The different learning rate schedules used were: constant, exponential, polynomial, and cosine. The learning rate schedule is the process of scheduling the learning rate during training, in order to have better performance and stability. All learning rate schedules, except for the constant one, are initialized with 1000 decay steps, which was also used in Woo et al. (2023).

Optimizer The optimizer is the algorithm used to update the weights of the network. The optimizers used were RMSProp, SGD, Adam, and AdamW. The work of Ruder (2016) suggests that the choice of optimizer can have a significant impact on the convergence rate and final performance of the model. Du and Lee (2018) explored the impact of different optimization algorithms, including SGD and Adam, on the performance of DL networks. The authors found that different optimizers can result in significantly different convergence rates and final test accuracies.

Optimizer momentum The optimizer momentum is a hyperparameter that helps to accelerate the learning process and improve the convergence of the algorithm. It is used to control the contribution of the previous gradients to the current update, which can help to smooth out the updates and prevent oscillations during training. The momentum parameter is typically set between 0.0 and 1.0, with higher values indicating a stronger contribution of the previous gradients. In our given options, the momentum values are 0.0, 0.5, 0.9, and 0.99. Ruder (2016) explain that momentum can help "speed up convergence and reduce oscillations" in the optimization process. Similarly, (Smith (2018)) discusses the importance of momentum in optimization algorithms and provides empirical evidence to support its effectiveness in improving the convergence of neural network models.

Internal optimizations Internal optimizations are techniques used to optimize the performance and efficiency of DL models during training and inference. In this list, we have some common, but also increasingly important internal optimizations used in DL: None, Pre quantization, Post quantization, and JIT-compilation. Pre quantization is an optimization technique that involves quantizing the weights and activations of the model to a lower precision before training or inference. This can help to reduce the memory and computation requirements of the model, making it more efficient and faster to execute. Post quantization is similar to pre quantization, but involves quantizing after it has been trained. This can help to reduce the size of the model and improve its performance on devices with limited computational resources (Gholami et al. (2021)). JIT-compilation, or Just-In-Time compilation, is an optimization technique that involves compiling the model code during runtime, rather than ahead of time, leading to faster training and improved performance (Izawa, Masuhara, and Bolz-Tereick (2022)).

2.6 Measuring GPU usage

In order to measure the performance of our model on a GPU, we called a custom written callback function in the training process to be executed at specific intervals (e.g. after each epoch of training). The callback function is designed to capture various performance metrics related to the GPU, as well as the accuracy and losses of the model during training. To capture GPU performance metrics, the callback uses *nvidia-smi*, which is a command-line tool provided by NVIDIA for monitoring and managing GPU devices. This tool can be used to obtain information about the power draw, temperature, and utilization of the GPU, as well as other performance metrics.

After each epoch of training, the callback function saves the model's accuracy and losses, as well as the GPU information (e.g. power draw in Watt), and the time per epoch. This data is saved per run, allowing for easy comparison and analysis of the performance metrics across all runs.

2.7 Efficiency Quotient

In order to measure the efficiency of the training process, we chose to consider three key resulting variables: power draw, epoch length in seconds, and accuracy. Generally speaking, if a model consumes lower power per shorter times, while also achieving a high accuracy, it is likely to be an efficient model. These variables enable us to evaluate the overall efficiency of a trained model, taking into account both the computational resources required and the performance achieved.

The Efficiency Quotient is derived by first calculating the product of the average power draw per epoch[1], and the average time taken per epoch in seconds. Subsequently, we compute the mean of this product across all epochs, yielding the average power-time consumption per epoch. Finally, we divide the training accuracy of the model by this average power-time consumption value to obtain the Efficiency Quotient:

$$EQ = \frac{Model\ Accuracy}{\frac{1}{N}\sum_{i=1}^{N} PD_i \cdot T_i} \tag{1}$$

where N is the total number of epochs, PD_i is the power draw at epoch i, and T_i is the time taken for epoch i in seconds.

3 Results

3.1 Model Performance

The model's performance varied heavily across training runs. The main reason for this could be the incompatibility of certain combinations of parameters. For future research we would therefore propose a more fine tuned coordination of parameters. We elaborate further on this in 4.

As depicted in Figure 4 and 1, most of the runs resulted in a testing accuracy between 0% and 20%,

[1]DISCLAIMER: in our case, it is a value obtained as a snapshot of around a single second during training. We suggest a different approach in 4.

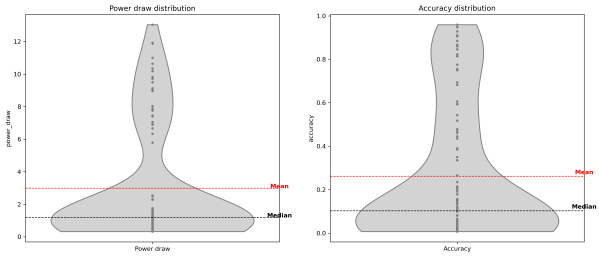


Figure 3: Power draw and accuracy distributions over all runs.

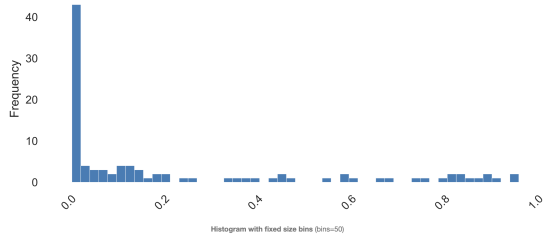


Figure 4: Test accuracy frequency.

while only a few runs managed values higher than 80%. In Figure 5, we can see that even though most runs take 100 seconds or less per epoch, it is primarily runs below 20% testing accuracy training for more seconds per epoch (e.g. 400 seconds per epoch).

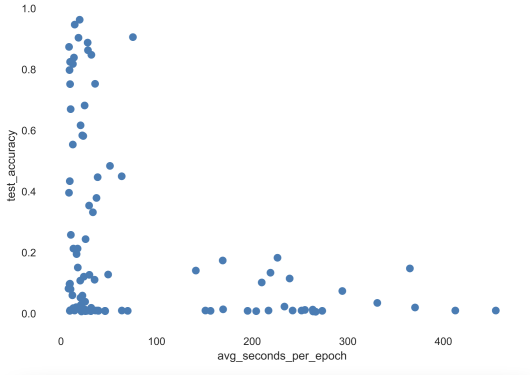


Figure 5: Test accuracy plotted against average time per epochs (in seconds).

In 13 of the 100 runs, the model achieved an accuracy above 80%. Figure 3 shows the distribution of all validation accuracies after the final epoch. We can observe that the distribution of accuracy values is scattered and skewed towards lower values, and the majority of the values are less than 20% revealing that the model is not very accurate overall. In certain cases however, the model showed high accuracy (e.g. 91.1% or 92.8%).

Figure 7 shows matrix visualisations of all runs, indicating which parameters were switched on during training. While the small number of total runs and the inter-dependencies between parameters makes it difficult to testify any hypothesis, we can observe some patterns in the parameter configurations. We can note, that a small batch size of 4 was not present in runs above 35% validation accuracy. On the contrary, a large batch size of 128 is scattered amongst the runs with high accuracy. This would confirm the assumption that larger batch sizes may help training progress faster. We can also note, that a cosine learning rate schedule was not present in runs above 35%. Another observation is that runs where Adam was set as optimizer are distributed closer to the top of the matrix, while runs where AdamW was set as optimizer are distributed closer to the bottom. All other values seem uniformly distributed.

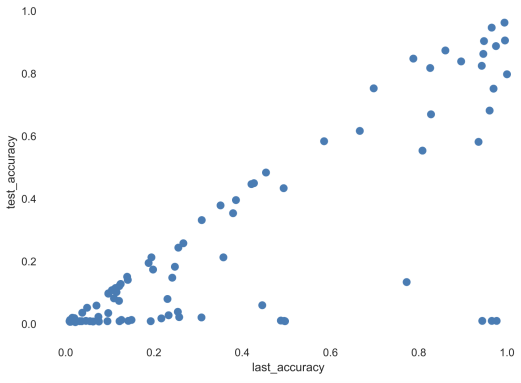


Figure 6: Test accuracy plotted against training accuracy (Overfitting).

Some amount of overfitting was also noticed during training. Figure 6 plots the testing accuracy against the training accuracy. The runs in the lower right triangle are the ones that show some degree of overfitting (e.g. having a high training accuracy, but a lower testing accuracy).

3.2 GPU

The GPU power draw values in Figure 3 show a wide distribution, with values ranging from 0.324 to 13.038 Watts per hour. The median value is around 1.5 Watt per hour, with a standard deviation of 2.86. Similar to the performance values, it appears that the power draw of the GPU can vary greatly depending on the specific model configuration used in the training process.

Figure 8 shows that most of the GPU Watt consumption as a single snapshot per epoch, varies greatly across all runs. We see that some Watt consumption

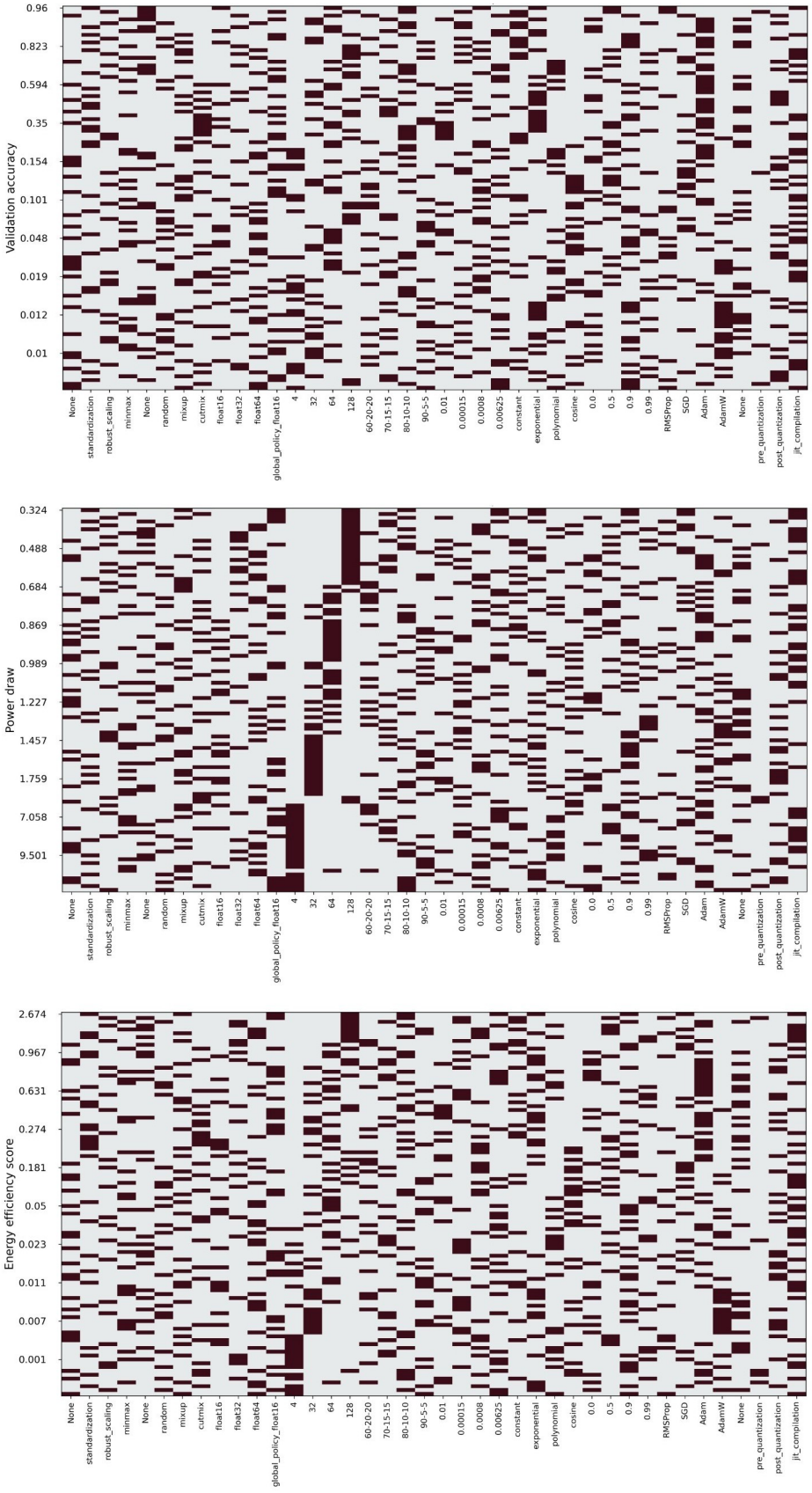


Figure 7: Matrix visualizations indicating the parameter settings for each run (a dark field indicates that the respective parameter was switched on). The first matrix shows all parameter configurations, sorted in ascending order in terms of validation accuracy. The second matrix shows all parameter configurations, sorted in descending order in terms of power draw. The third matrix shows all parameter configurations, sorted in ascending order in terms of the energy efficiency quotient.

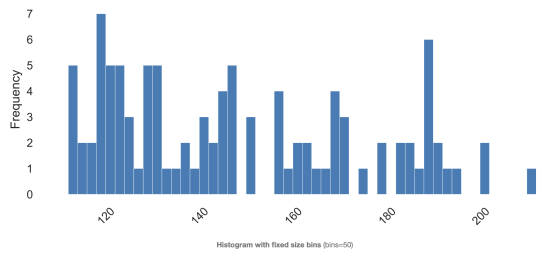


Figure 8: Average GPU Watt consumption per epoch, averaged across all epochs per training run, plotted against frequency.

is concentrated at around 120 Watt, while also having some individual outliers that consumed around an average of 200 Watt per epoch. By plotting the validation accuracy against the average GPU Watt consumption per epoch, as seen in Figure 9, we can gather that many 0% to 20% accuracies trained with an evenly distributed low consumption. In this plot, the efficient frontier of training are the runs that achieved both a high accuracy and low average GPU Watt consumption, so the ones in the upper left corner of the plot. In contrast to these, are the values in the lower right corner of the plot, which exhibit very high consumption but also low testing accuracy.

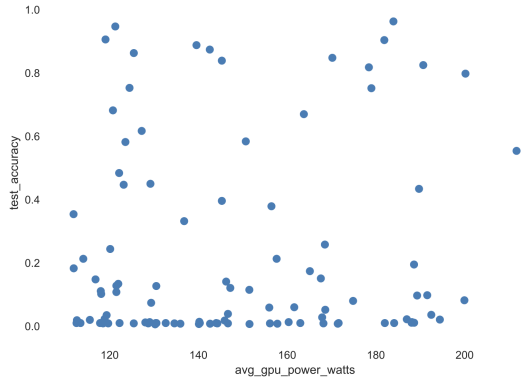


Figure 9: Test accuracy plotted against average GPU Watt consumption per epoch.

Looking at the matrix visualisations in Figure 7, we can observe a few patterns. Similarly to the accuracy patterns, the parameter batch size shows the most apparent results. The parameter values are almost linearly distributed between runs with low and high GPU power draws. A batch size of 4 is, with only one exception, present in all runs with a power draw above 7 Watts per hour. On the contrary, a batch size of 128 is present in all runs below 0.684. We can conclude that, when the objection of training is to keep GPU power draw low, it is advisable to choose a larger batch size. We can not make any other observations, since all other values are uniformly distributed and present in runs with high GPU power draws between 5-10 Watts per hour as well as with a GPU power draw below 1 Watt per hour.

3.3 Energy Efficiency

As suspected, there is a high variety throughout all metrics including training times. For a somewhat objective comparison between runs, we decided to use the energy quotient outlined in 2.7. Although this is an oversimplified quotient, it hints at some underlying patterns which we propose to look into in future research. The last matrix in Figure 7 shows parameter configurations, sorted in ascending order by their energy efficiency. Most of the parameters seem to follow a uniform distribution but we can conclude with a few tendencies that confirm our previous observations 3.1 and 3.2.

The parameter batch size shows clear tendencies for smaller batch sizes to invoke a less efficient training. On the other hand, a large batch size of 128 results in more efficient training. Another pattern we can observe is the tendency of runs with Adam as optimizer showing efficient training results. Runs with AdamW as optimizer represent the lower range of runs in terms of efficiency. As last observations, the final matrix confirms our observations from 3.1 about cosine as learning rate schedule. In the efficiency matrix it is not represented in any runs with high energy efficiency.

parameter_value	coef	std err	t	P>|t|	[0.025	0.975]
preprocessing_None	0.3407	0.246	1.386	0.172	-0.153	0.835
preprocessing_minmax	0.6148	0.241	2.549	0.014	0.13	1.1
preprocessing_robust_scaling	0.7701	0.339	2.268	0.028	0.088	1.452
preprocessing_standardization	-0.0934	0.301	-0.311	0.757	-0.698	0.511
augmentation_None	0.1098	0.284	0.386	0.701	-0.462	0.681
augmentation_cutmix	0.2389	0.257	0.93	0.357	-0.277	0.755
augmentation_mixup	0.5115	0.263	1.943	0.058	-0.017	1.04
augmentation_random	0.7721	0.281	2.751	0.008	0.208	1.336
batch_size_4	6.5199	0.263	24.814	0	5.992	7.048
batch_size_32	-1.239	0.312	-3.973	0	-1.866	-0.612
batch_size_64	-1.3272	0.25	-5.311	0	-1.829	-0.825
batch_size_128	-2.3215	0.323	-7.183	0	-2.971	-1.672
lr_0.00015	-0.2992	0.288	-1.037	0.305	-0.879	0.281
lr_0.0008	0.7753	0.257	3.021	0.004	0.259	1.291
lr_0.00625	0.3539	0.245	1.444	0.155	-0.139	0.847
lr_0.01	0.8021	0.276	2.906	0.005	0.247	1.357
lr_schedule_constant	0.3678	0.262	1.404	0.167	-0.159	0.894
lr_schedule_cosine	0.0641	0.286	0.224	0.824	-0.511	0.639
lr_schedule_exponential	1.0098	0.27	3.742	0	0.468	1.552
lr_schedule_polynomial	0.1905	0.352	0.541	0.591	-0.517	0.898
partitioning_60-20-20	0.1457	0.294	0.495	0.623	-0.446	0.737
partitioning_70-15-15	0.1663	0.265	0.628	0.533	-0.366	0.698
partitioning_80-10-10	0.6519	0.271	2.408	0.02	0.108	1.196
partitioning_90-5-5	0.6683	0.269	2.482	0.017	0.127	1.209
optimizer_Adam	0.5615	0.226	2.486	0.016	0.108	1.015
optimizer_AdamW	0.2131	0.286	0.744	0.46	-0.362	0.789
optimizer_RMSProp	0.3785	0.286	1.321	0.192	-0.197	0.954
optimizer_SGD	0.4791	0.28	1.712	0.093	-0.083	1.041
optimizer_momentum_0.0	0.7925	0.249	3.188	0.002	0.293	1.292
optimizer_momentum_0.5	-0.1966	0.274	-0.716	0.477	-0.748	0.355
optimizer_momentum_0.9	0.5699	0.236	2.42	0.019	0.097	1.043
optimizer_momentum_0.99	0.4664	0.343	1.359	0.18	-0.223	1.156
internal_None	-0.7924	0.277	-2.858	0.006	-1.349	-0.235
internal_jit_compilation	-1.0865	0.276	-3.934	0	-1.641	-0.532
internal_post_quantization	-0.5801	0.289	-2.005	0.051	-1.162	0.001
internal_pre_quantization	4.0911	0.463	8.831	0	3.16	5.022
precision_float16	0.2758	0.302	0.913	0.366	-0.331	0.883
precision_float32	0.6651	0.322	2.067	0.044	0.018	1.312
precision_float64	0.8149	0.247	3.296	0.002	0.318	1.312
precision_global_policy_float	-0.1236	0.242	-0.511	0.611	-0.61	0.362

Figure 10: OLS regression results.

3.4 Statistical Inference

Power draw and accuracy indicate a moderate negative correlation (e.g. -0.39). This suggests that as the power draw increases, the accuracy decreases to some extent. However, the correlation is not very strong,

meaning that the relationship between the two variables is not very consistent or predictable. However, it is important to note that correlation does not necessarily imply causation, and other factors may be influencing the relationship between the variables.

Regression Analysis We deployed Ordinary Least Squares (OLS) regression to study the relevance of each parameter as well as individual parameter values for the GPU power draw. OLS regression is a statistical method used to analyze the relationship between one or more independent variables and a dependent variable. The goal of OLS regression is to find the line (or plane in higher dimensions) that best fits the data by minimizing the sum of the squared differences between the predicted values and the actual values of the dependent variable. The performance of the regression model is measured using the R-squared value, which indicates the proportion of the variance in the dependent variable that is explained by the independent variables. The adjusted R-squared value takes into account the number of predictors and adjusts the R-squared value accordingly to prevent overfitting.

Regression Results The model returns an R-squared value of 0.942, indicating that the model is able to explain 94.2% of the variance in the dependent variable. The F-statistic of 26.49 and corresponding p-value of 2.84e-21 indicate that the model as a whole is statistically significant in predicting GPU power draw. However, it's important to note that the model includes 30 independent variables and only 80 observations, which could potentially lead to overfitting and difficulty in generalizing to new data. The adjusted R-squared value of 0.906 suggests that the model may have some degree of overfitting, as it is lower than the R-squared value. Therefore, it may be useful to perform additional analyses, such as cross-validation, to assess the model's performance on new data.

Figure 3.3 displays the coefficients, standard errors, t-values, and p-values for each of the independent variables. The coefficients represent the expected change in the dependent variable associated with a one-unit increase in the corresponding independent variable, holding all other independent variables constant. The t-values and p-values are used to determine whether the coefficients are significantly different from zero. A p-value less than (alpha=) 0.05 indicates that the coefficient is statistically significant. For our experiment we used (alpha=) 0.1 as threshold.

To test our observations from 3.2, we examined the coefficients for batch size, learning rate schedule, and optimizer. We also examined the coefficients for internal optimization, since the OLS regression indicated minor influence of certain parameter values on the power draw.

Batch size The coefficient values indicate the direction and magnitude of the relationship between batch size and power draw. A negative coefficient suggests an inverse relationship, meaning that as batch size increases, power draw tends to decrease. We can see a clear discrepancy between a small batch size of 4 and a large batch size of 128, confirming that this batch size has a negative or an increasing effect on the power draw. The p-values of 0 for all parameter values in batch size indicate that the coefficient values are statistically significant.

Learning rate schedule The coefficient for a cosine learning rate schedule do not differ from the coefficients of the other parameter values. Also, its p-value is above the defined alpha threshold of 0.1, indicating that it is not robust for statistical inference.

Optimizer The same can be said about the coefficients for the optimizers. There are no major discrepancies between coefficients of the different parameter values. Apart from AdamW, the p-values indicate that the coefficients are not statistically significant.

Internal optimizations The positive coefficient for pre-quantization suggests an increasing or positive effect of this value on the power draw. On the other hand, negative coefficients for all other parameter values in this parameter group suggest a decreasing or negative effect on the power draw.

4 Discussion

During our analysis, we recognized that additional epochs and runs would have strengthened the statistical support for the hypotheses and results presented. One possibility to improve the statistical confidence of subsequent analyses and inferences would have been to consider only those runs that exceeded a certain accuracy threshold. However, due to the limitations of the project, the number of training runs was restricted to a maximum of 100. We anticipate that by allowing for more runs, the number of usable training runs for subsequent analysis would increase.

It is essential to note that due to the limitations, parameters were not tested in isolation, which leaves uncertainty about the interdependence of parameters and their values. While this approach is practical, non-linear parameter interactions could lead to biased or incorrect analysis results.

Nevertheless, to confirm our results, we ran a final training of a (self-implemented) version of the ResNet-50 model. We initialized our model with the specific hyperparameter configuration of the winning run in terms of energy efficiency and trained this model for 100 epochs. This model achieved the best results, compared with all previously tested models. Its energy efficiency score of 6.16 constituted of a GPU power draw of 0.14 Watt per hour and a validation accuracy of 89%. This would suggest the capacity of the framework to having successfully examined the most efficient hyperparameter configuration.

To improve future research, we suggest pursuing two possible directions. One way would be exploring alternative search strategies, such as Bayesian optimization (J. Wu et al. (2019), Victoria and Maragatham (2021)) or Genetic algorithms (Aszemi and Dominic (2019); Xiao, Yan, Basodi, Ji, and Pan (2020)), which have

been disregarded for the scope of this work due to their complexity of implementation. This would testify our chosen search strategy on the given research question.

Secondly, further work should be directed at studying both the incompatibility of certain parameters and the parameters that showed influence on the computational efficiency. Also, increasing the number of runs, and employing different statistical inference models beyond ordinary least squares (OLS) regression would achieve greater statistical relevance

Finally, we advice to track GPU power draw continuously during training and extend the energy efficiency quotient by incorporating additional parameters to obtain statistical significance.

These improvements will help to increase the validity and reliability of the results and provide more comprehensive insights into the phenomena under investigation. We recognize that this method will require more computational resources and initially works against an energy-conscious approach. However, it will enable more robust and accurate analyses and help improving the energy footprint of models in the long term.

5 Conclusion

This work explored the impact of different hyperparameters with respect to model performance and energy consumption during training.

We built an automated framework that tests different combinations of hyperparameters in random configurations. While we could observe first patterns with regard to parameters such as batch size, it was not possible to generate statistically significant results due to time and computational limitations in the context of this work.

Future work should focus on training for more runs with more epochs and extending the pool of hyperparameters to test. More advanced search strategies could be deployed to speed up the optimization process. Additionally, different models should be used to allow for more general assertions.

References

Anthony, L. F. W., Kanding, B., & Selvan, R. (2020). Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *CoRR*, *abs/2007.03051*. Retrieved from https://arxiv.org/abs/2007.03051

Aszemi, N. M., & Dominic, P. (2019). Hyperparameter optimization in convolutional neural network using genetic algorithms. *International Journal of Advanced Computer Science and Applications*, *10*(6).

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization [misc]. *Journal of Machine Learning Research*, *13*(Feb), 281-305. Retrieved from http://www.jmlr.org/papers/v13/bergstra12a.html

Cheng, Y., Wang, D., Zhou, P., & Zhang, T. (2017). A survey of model compression and acceleration for deep neural networks. *CoRR*, *abs/1710.09282*. Retrieved from http://arxiv.org/abs/1710.09282

Chowdhury, A. A., Hossen, M. A., Azam, M. A., & Rahman, M. H. (2022). Deepqgho: Quantized greedy hyperparameter optimization in deep neural networks for on-the-fly learning. *IEEE Access*, *10*, 6407-6416. DOI: 10.1109/ACCESS.2022.3141781

Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., & Le, Q. V. (2019). Autoaugment: Learning augmentation policies from data. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 113–123).

Du, S. S., & Lee, J. D. (2018). Implicit regularization in deep matrix factorization. In *Advances in neural information processing systems* (pp. 109–118).

García-Martín, E., Rodrigues, C. F., Riley, G., & Grahn, H. (2019). Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, *134*, 75-88. Retrieved from https://www.sciencedirect.com/science/article/pii/S0743731518308773 DOI: https://doi.org/10.1016/j.jpdc.2019.07.007

Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., & Keutzer, K. (2021). *A survey of quantization methods for efficient neural network inference.*

Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015). Deep learning with limited numerical precision. In *International conference on machine learning* (pp. 1737–1746).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, *abs/1512.03385*. Retrieved from http://arxiv.org/abs/1512.03385

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Izawa, Y., Masuhara, H., & Bolz-Tereick, C. F. (2022). *Two-level just-in-time compilation with one interpreter and one engine.*

Keskar, N. S., Mudigere, D. V., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2016).

On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

Larochelle, H., Erhan, D., Courville, A. C., Bergstra, J., & Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *International conference on machine learning*.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the Institute of Radio Engineers*, *86*(11), 2278–2323. DOI: 10.1109/5.726791

Li, D., Chen, X., Becchi, M., & Zong, Z. (2016). Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus. In *2016 ieee international conferences on big data and cloud computing (bdcloud), social computing and networking (socialcom), sustainable computing and communications (sustaincom) (bdcloud-socialcom-sustaincom)* (p. 477-484). DOI: 10.1109/BDCloud-SocialCom-SustainCom.2016.76

Panda, P., Sengupta, A., & Roy, K. (2017). Energy-efficient and improved image recognition with conditional deep learning. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, *13*(3), 1–21.

Parsa, M., Ankit, A., Ziabari, A., & Roy, K. (2019). Pabo: Pseudo agent-based multi-objective bayesian hyperparameter optimization for efficient neural accelerator design. In *2019 ieee/acm international conference on computer-aided design (iccad)* (pp. 1–8).

Pereira, J. P. F., de Sá, E., & Alexandre, L. A. (2017). A systematic study of the class imbalance problem in convolutional neural networks. *CoRR*, *abs/1710.05381*. Retrieved from http://arxiv.org/abs/1710.05381

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Sevilla, J., Heim, L., Ho, A., Besiroglu, T., Hobbhahn, M., & Villalobos, P. (2022). Compute trends across three eras of machine learning. In *2022 international joint conference on neural networks (ijcnn)* (pp. 1–8).

Sharir, O., Peleg, B., & Shoham, Y. (2020). The cost of training NLP models: A concise overview. *CoRR*, *abs/2004.08900*. Retrieved from https://arxiv.org/abs/2004.08900

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*.

Strubell, E., Ganesh, A., & McCallum, A. (2019). *Energy and policy considerations for deep learning in nlp.* arXiv. Retrieved from https://arxiv.org/abs/1906.02243 DOI: 10.48550/ARXIV.1906.02243

Victoria, A. H., & Maragatham, G. (2021). Automatic tuning of hyperparameters using bayesian optimization. *Evolving Systems*, *12*, 217–223.

WeightsBiases. (n.d.). *Sweeps: Scalable, customizable hyperparameter search and optimization.* Retrieved 2023-03-30, from https://wandb.ai/site/sweeps

Woo, S., Debnath, S., Hu, R., Chen, X., Liu, Z., Kweon, I. S., & Xie, S. (2023). *Convnext v2: Co-designing and scaling convnets with masked autoencoders.*

Wu, C.-J., Raghavendra, R., Gupta, U., Acun, B., Ardalani, N., Maeng, K., ... Hazelwood, K. (2022). Sustainable ai: Environmental implications, challenges and opportunities. In D. Marculescu, Y. Chi, & C. Wu (Eds.), *Proceedings of machine learning and systems* (Vol. 4, pp. 795–813). Retrieved from https://proceedings.mlsys.org/paper_files/paper/2022/file/ed3d2c21991e3bef5e069713af9fa6ca-Paper.pdf

Wu, J., Chen, X.-Y., Zhang, H., Xiong, L.-D., Lei, H., & Deng, S.-H. (2019). Hyperparameter optimization for machine learning models based on bayesian optimizationb. *Journal of Electronic Science and Technology*, *17*(1), 26-40. Retrieved from https://www.sciencedirect.com/science/article/pii/S1674862X19300047 DOI: https://doi.org/10.11989/JEST.1674-862X.80904120

Xiao, X., Yan, M., Basodi, S., Ji, C., & Pan, Y. (2020). Efficient hyperparameter optimization in deep learning using a variable length genetic algorithm. *CoRR*, *abs/2006.12703*. Retrieved from https://arxiv.org/abs/2006.12703

Yang, T.-J., Chen, Y.-H., Emer, J., & Sze, V. (2017). A method to estimate the energy consumption of deep neural networks. In *2017 51st asilomar conference on signals, systems, and computers* (p. 1916-1920). DOI: 10.1109/ACSSC.2017.8335698

Young, S. R., Devineni, P., Parsa, M., Johnston, J. T., Kay, B., Patton, R. M., ... Potok, T. E. (2019). Evolving energy efficient convolutional neural networks. In *2019 ieee international conference on big data (big data)* (p. 4479-4485). DOI: 10.1109/BigData47090.2019.9006239

6 Appendix

6.1 Implementation

In the following section, we walk the reader through the implementation steps of our project.

Our repository consists of five subfolders, called `utils`, `runs`, `models`, `tests`, and `report` (the latter two were used for testing and storing output for this report). The root or parent folder of our repository contains a bash file `greenscreen.sh` to call and control the project pipeline and a main training file `main.py` to run and control individual training runs.

`greenscreen.sh` starts the pipeline by calling `main.py`. For each run, `main.py` calls `config_creator.py` in `utils` to generate a random combination of parameters for one run. Based on the selected parameters, `main.py` prepares preprocessing on the data, builds and compiles the model (imported from the folder `models`), and initiates model training. The hyperparameters were pre-selected and stored in `config.yaml` in `utils`.

For every training run, a new subfolder in `runs/` was created, where the respective dictionary of parameters was stored along with metrics of the training run. The metrics were tracked using using a custom callback function, like so:

```
combined_model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=args.epochs,
    callbacks=[SMICallback()],
)
```

This callback function starts GPU tracking upon the first epoch of training and logs GPU power draw in Watt for one second at the start of each epoch until the last epoch with the following command:

```
subprocess.check_output(['nvidia-smi',
'--query-gpu=power.draw,
temperature.gpu,utilization.gpu',
'--format=csv,noheader'])
```

As explained in Section 2.6, in addition to GPU power draw, the function tracks losses and accuracy and logged those together in a `logs.csv` in the respective training run folder.

After training and logging, this process was repeated for 100 runs until the pipeline had logged all their respective training metrics in the `runs/` folder. These were used for analysis afterwards.

Next, `analysis.py` is called to compare all runs and extract the top runs in terms of accuracies, lowest GPU power draw, and overall efficiency (accuracy / GPU power draw). It also runs inference statistics to check for the influence / relevance of each parameter for the GPU power draw. Our initial idea was that `analysis.py` hands over to `main.py` the parameter value of each parameter that is most relevant for a low GPU power draw. With this, `greenscreen.sh` would call `main.py` for a final training of 50 epochs. However, due to instability and statistically weak results, this remains an option rather than the default.

Nevertheless, we initialized final training with one specific hyperparameter configuration (e.g. that of the winning run in terms of energy efficiency) for 100 epochs. This model achieved the best results among all previously tested models.

Finally, `pdf_creation.py` may be called to produce a pdf report with a summary of the training results. The report includes tables with the top runs in terms of accuracies, lowest GPU power draw, and overall efficiency (accuracy / GPU power draw) as well as plots visualizing loss, accuracy, and GPU power draw during training of the most efficient run as well as the respective parameters that were switched on.