Quick-Tune-Tool: A Practical Tool and its User Guide for Automatically Finetuning Pretrained Models

Ivo Rapant[1] **Lennart Purucker**[1] **Fabio Ferreira**[1] **Sebastian Pineda Arango**[1] **Arlind Kadra**[1] **Josif Grabocka**[2] **Frank Hutter**[3, 1]

[1]University of Freiburg
[2]University of Technology Nuremberg
[3]ELLIS Institute Tübingen

Abstract Pretrained models have become essential tools for machine learning practitioners across various domains, including image classification, segmentation, and natural language processing. However, the complexity of selecting the appropriate pretrained model and finetuning strategy remains a significant challenge. In this paper, we present `Quick-Tune-Tool`, an automated solution to guide practitioners in selecting and finetuning pretrained models. Leveraging the Quick-Tune algorithm, `Quick-Tune-Tool` abstracts intricate research-level code into a user-friendly tool. Our contributions include the release of `Quick-Tune-Tool`, a detailed architectural overview, a user guide for image classification, and empirical evaluations. In experiments on four vision datasets, our results underscore the effectiveness and practicality of `Quick-Tune-Tool` for automating model selection and finetuning.

1 Introduction

Pretrained models are well-performing solutions for many machine learning practitioners for an expanding amount of domains from image classification (Caron et al., 2021), image segmentation (Kirillov et al., 2023), natural language processing (Qiu et al., 2020), time series (Liang et al., 2024), to tabular data (Hollmann et al., 2023). As a result, pretrained models are constantly being published on model hubs such as Hugging Face[1] or in the *timm* library (Wightman, 2019).

A challenge with this practice is that the multitude of pretrained models poses a substantial complexity for machine learning practitioners; prompting hard-to-answer questions such as *which* pretrained model to use and *how*. For instance, a practitioner in the domain of image classification must choose between one of the more than 700 pretrained models in the *timm* library. Afterwards, the practitioner must optimize the model for their application by *also* selecting a finetuning strategy and its hyperparameters, such as the regularization technique or learning rate schedule.

Our overarching goal is to automate the process of *which* pretrained model to use and *how* for *practitioners in any domain*. Therefore, in this paper, we present `Quick-Tune-Tool`, the first step towards our goal. For this, we adopt the recently proposed Quick-Tune algorithm (Arango et al., 2023) and both abstract and elevate it from research-level code to a practitioner's tool with a strong focus on usability. We intentionally abstracted and optimized `Quick-Tune-Tool` for future adaptation to new domains. The first version, presented in this work, is already a user-friendly, broadly accessible, and easy-to-install tool for practitioners in the domain of image classification.

Our contribution. As part of this paper, we: (**A**) publish the first version of `Quick-Tune-Tool`[2], (**B**) present its architecture and design principles, (**C**) provide a user-guide to do image classification in 3 lines of code, and (**D**) perform experiments showing that `Quick-Tune-Tool` is easy to use and outperforms random search on 4 image classification datasets.

[1]`https://huggingface.co/posts`
[2]`https://github.com/automl/QTT`

2 Background and Related Work

Finetuning Tools. To leverage pretrained models, the research literature has proposed a large set of methods such as Co-Tuning (You et al., 2020) or SP Regularization (Li et al., 2018) for finetuning. Unfortunately, most of this work does not translate into maintained, easy-to-use code that can be readily applicable in the industry. The development of finetuning tools allows us to close the gap between researchers and practitioners, facilitating the application of the research findings in production use cases. The *Transfer Learning Library* (Jiang et al., 2020) builds a collection of finetuning strategies for computer vision, streamlining testing methods on new datasets. The *PEFT library* (Mangrulkar et al., 2022) encapsulates different state-of-the-art approaches for parameter efficient finetuning such as Adapters or LoRA (Hu et al., 2022), and integrates with Hugging Face Hub. Finally, *TorchTune* (Pytorch, 2024) provides a command line interface for finetuning models natively using the Torch backend while offering a tool that is easy to integrate. However, none of these tools natively offer a way to select the model and hyperparameters to use, thus leaving it to their users to determine which pretrained model to use and how.

AutoML Systems. The diversity of datasets and tasks requires the use of automated machine learning (AutoML) systems to automatically select the best model with the best hyperparameters for a given task without incurring manual trial-and-error. For deep learning, tools such as *AutoKeras* (Jin et al., 2019), *AutoPytorch* (Zimmer et al., 2021), or *NePS* (Stoll et al., 2023) allow users to automatically search for hyperparameter configurations of neural networks; typically tailored to optimizing the architecture or (pre-)training but not finetuning of neural networks. In contrast, *AutoGluon MultiModal* (Tang et al., 2024) offers support for finetuning foundation models for different multimodal tasks, such as Classification, Regression, or Image Segmentation, by evaluating a pre-defined portfolio of hyperparameters and models. Lastly, the AutoML system *ZAP* (Öztürk et al., 2022), addresses the finetuning and hyperparameter search problem by leveraging an algorithm selection perspective to find appropriate finetuning hyperparameters and pretrained models through the abstraction of deep learning pipelines.

QuickTune Method. *QuickTune* (Arango et al., 2023) uses a probabilistic performance predictor $\hat{\ell}_\theta$ and a cost predictor $\hat{c}_w$ to select the pipeline $x \in \mathcal{X}$ to finetune for Δt epochs after resuming from the checkpoint at epoch t (Equation 1). Thus, it balances the Multi-fidelity Expected Improvement (Wistuba et al., 2022) with the actual cost of finetuning. Information from auxiliary tasks is included by meta-learning the parameters θ and w. *QuickTune* selects the pipeline x from a search space $\mathcal{X}$ containing different combinations of hyperparameters and pretrained models. These combinations are fed into the predictors using MLP encoders (Pineda Arango and Grabocka, 2023). Specifically, the authors applied the method on Computer vision tasks from the *Meta-Album* collection (Ullah et al., 2022) using the model Hub contained in *Timm library* (Wightman, 2019). In the search space design, they included different finetuning strategies. However, the method is task-agnostic and can be easily adapted to other data modalities or search spaces.

$$x \in \text{argmax}_{x \in \mathcal{X}} \frac{\text{EI}_{\text{MF}}(x, t + \Delta t, \hat{\ell}_\theta)}{\hat{c}_w(x, t + \Delta t)} \tag{1}$$

Overview. We summarize the position of `Quick-Tune-Tool` in the area of ML in Appendix A.

3 Quick-Tune-Tool: A Practical Tool For Finetuning Pretrained Models

The architecture of our tool is composed of four main components: `QuickTuner`, `ConfigManager`, `Optimizer`, and the `Objective Function`. These components work together to provide a comprehensive solution for selecting a finetuning pipeline, i.e. a pretrained model from a Model hub and its finetuning hyperparameters from a search space. We provide an overview of the architecture in Figure 1 and describe each component in detail in the remainder of this section.

In summary, using `Quick-Tune-Tool` starts by defining the search space and having the `ConfigManager` generate initial configurations. `QuickTuner` then employs the `Optimizer` to suggest configurations based on previous evaluations, e.g. `History`. Next, the `Objective Function` evaluates these configurations and returns results of performance metrics. Finally, `QuickTuner` saves these results, updates the `Optimizer`, and repeats the cycle until the time budget is exhausted.

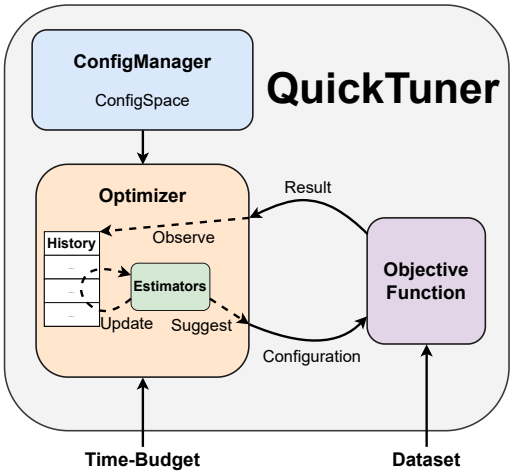

Figure 1: Quick-Tune-Tool Architecture.

QuickTuner. The tuner is the core component that organizes the optimization process by integrating all components, managing environment setup, interacting with the optimizer, invoking the objective function, saving intermediate results, and ensuring experiment continuity.

ConfigManager. The `ConfigManager`, as the name suggests, manages the configurations and processes them to be input to the optimizer. It takes a configuration space as input, i.e., the pipeline search space over which the optimization is performed. We use `ConfigSpace` (Lindauer et al., 2019) to define the search space.

```python
def objective_function(
config: dict,
task_info: dict | None,
) -> dict | list[dict]:
# 1. setup
# 2. run objective
# 3. evaluate model
# 4. collect results
return result
```

Figure 2: Objective Function.

Optimizer. The optimizer component can implement various optimization methods. Its main task is to interact with the tuner to suggest the next configuration to evaluate. Currently, we provide the *QuickTune* optimizer (Arango et al., 2023) and additionally, a basic random-search optimizer. To ensure flexibility and extensibility, any optimization method can be easily added, as long as it adheres to the interface requirements of the `QuickTuner`, e.g. a Bayesian optimization method like BOHB (Falkner et al., 2018), DPL (Kadra et al., 2024) or ifBO (Rakotoarison et al., 2024), or evolutionary based like DEHB (Awad et al., 2021).

Objective Function. The Objective Function is invoked by the tuner during optimization. It can be any function that accepts configuration, budget, and optional task-related information. It returns results as a dictionary or a list of dictionaries. For efficiency, it should manage interrupted training by saving and loading models. Existing training scripts can be adapted to work with `Quick-Tune-Tool` if they adhere to the interface in Figure 2 and return results as shown in Figure 3.

```python
{
"config_id": 42,
"score": 0.99,
"cost": 123.45,
"fidelity": 1,
"status": True,
}
```

Figure 3: Result Dict.

Quick-Tune-Tool Already Supports Image Classification OOTB. We designed `Quick-Tune-Tool` to be abstract and provide an interface for fine-tuning in any domain or application. To test this interface in its first version, we implemented support for using `Quick-Tune-Tool` for image classification. Moreover, we plan to extend it to other domains in the future. In detail, for image classification, `Quick-Tune-Tool` provides three pretrained optimizers, meta-learned on learning curves from different Meta-Album versions: *micro, mini*, and *extended*. Moreover, we

provide a search space (see Table 3 in Appendix) and a connection to a model hub (Table 4). Thus, `Quick-Tune-Tool` can be applied to solve new image classification problems out-of-the-box.

4 A User Guide for Quick-Tune-Tool

To showcase the proposed `Quick-Tune-Tool`, we now present its exemplary usage for image classification. Figure 4 shows a basic use of `Quick-Tune-Tool` with the pretrained optimizer in the *micro* benchmark. The user only has to adapt the path to the custom dataset where the images have to be in PyTorch's ImageFolder format (Figure 7 in Appendix).

```
from qtt import QuickTuner, get_pretrained_optimizer
from qtt.finetune.cv.classification import finetune_script

task_info, metafeat = extract_task_info_metafeat("path/to/dataset")
optimizer = get_pretrained_optimizer("mtlbm/micro")
optimizer.setup(128, metafeat)  # number of configs
qt = QuickTuner(optimizer, finetune_script)
qt.run(task_info, time_budget=3600)
```

Figure 4: A simple example of using the Quick-Tune-Tool.

1. **Get the Optimizer**. The *get_pretrained_optimizer* method is designed to retrieve an optimizer based on specified parameters. Currently, QuickTuneTool includes three pretrained optimizers, which can be accessed using `"mtlbm/micro"`, `"mtlbm/mini"` and `"mtlbm/extended"`. This command will load the optimizer along with its associated search space and *ConfigManager*, constructing it for use with QuickTuner.

2. **Create QuickTuner Object With the Finetuning Script**. The finetune script serves as the objective function for QuickTuner. We provide a script for image classification, which takes a configuration as input and manages all aspects of running evaluations, like downloading pretrained weights, saving and loading models, and returning results.

3. **Fit the QuickTuner** The tuner handles the setup and correct flow of the optimization process. We only have to pass the optimizer and an objective function. Optionally, one can specify an output path and the logger verbosity. We start the optimization with `QuickTuner.run()`, we can additionally pass a dictionary with parameters that deviate from default. Depending on the size of the dataset, the fitting can take a few hours. Optionally, we can speed up the training by specifying a time limit and / or fixing the number ofevaluations. For example, `qt.fit(..., time_limit=3600)` will stop training after 3600 seconds. Higher limits will generally result in better performance.

   We present additional information about the tuning and post-tuning steps in the Appendix.

5 Experiments and Results: Quick-Tune-Tool in Action

We evaluated the performance of our finetuning tool on four widely-used vision datasets: Oxford Flowers 102 (Nilsback and Zisserman, 2008), Stanford Cars (Krause et al., 2013), Imagenette (Howard, 2020), and FGVC-Aircraft (Maji et al., 2013). An in-depth evaluation of the underlying Quick-Tune algorithm can be found in the prior work by Arango et al. (2023).

`Quick-Tune-Tool` finetuned models on the training sets while top-1 accuracy was measured on the validation sets. We show results using 1 step ($\Delta t = 1$) and 2 step ($\Delta t = 2$), which is the number of epochs evaluated per finetuning step. We select time budgets following prior work (Arango et al., 2023), with 1 and 4 hours for the different optimizers to account for the number of samples in the datasets. Table 1 presents the final top-1 accuracy per dataset, and Figure 5 visualizes the performance over time for all the compared methods. We observe that `Quick-Tune-Tool` consistently outperforms the RandomOptimizer, with top-1 accuracy improvements ranging from 13% to 23%. In general, `Quick-Tune-Tool` converges faster to a better solution than random search.

Dataset	Quick-Tune	Random	Time Budget (Hours)	Optimizer
Imagenette	$99.6_{[99.3-99.9]}$	$82.1_{[60.4-99.9]}$	1	micro
Oxford Flowers	$89.1_{[84.0-94.2]}$	$74.7_{[56.0-93.1]}$	1	micro
Stanford Cars	$53.5_{[34.4-72.7]}$	$40.4_{[22.2-58.6]}$	4	mini
FGVC-Aircraft	$48.0_{[35.2-60.8]}$	$25.0_{[16.2-33.9]}$	4	mini

Table 1: Top-1 Accuracy Results. Mean over ten seeds with confidence intervals ($\Delta t = 1$).

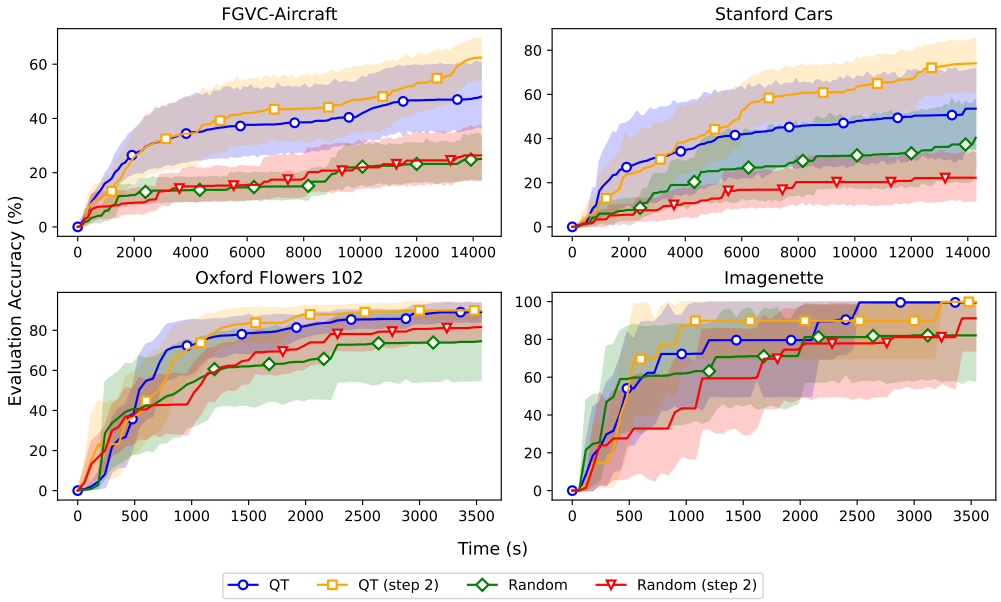

Figure 5: Evaluations on common vision datasets using `Quick-Tune-Tool` (QT) and Random Search as optimizers. We present results for $\Delta t \in \{1, 2\}$

6 Conclusion and Outlook

In this paper, we introduced `Quick-Tune-Tool`, a tool that simplifies the automated selection and finetuning of pretrained models. Our tool leverages the Quick-Tune algorithm to offer a user-friendly yet powerful solution for practitioners in image classification. Empirical evaluations on four datasets demonstrate that `Quick-Tune-Tool` offers a substantial performance improvement in finetuning pretrained models for image classification tasks.

`Quick-Tune-Tool` is designed with extensibility in mind, allowing for future adaptations to new domains and tasks. Our future work will focus on expanding the tool's capabilities to support additional data modalities and tasks, incorporating more advanced optimization techniques, and continuous integration of user feedback for further enhancement. By making advanced finetuning accessible and efficient, `Quick-Tune-Tool` stands to facilitate the wider adoption of pretrained models across diverse application areas.

Broader Impact Statement. We believe that our work does not present notable or new negative broader impacts. Yet, our work initially requires resource-intensive, environmentally costly computation. However, it has the potential for resource savings over time through optimized evaluations. In contrast, we believe that `Quick-Tune-Tool` provides a positive societal impact by enabling easier access to enhanced image classification tailored to, for example, medical diagnostics or disease detection applications. Furthermore, `Quick-Tune-Tool` democratizes using and finetuning pretrained models.

Acknowledgements. We acknowledge the financial support of the Hector Foundation. We also acknowledge funding by the European Union (via ERC Consolidator Grant DeepLearning 2.0, grant no. 101045765). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. Moreover, we acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant number 417962828, by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG. We also acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 499552394 – SFB 1597 and acknowledge Robert Bosch GmbH for financial support.

References

Arango, S. P., Ferreira, F., Kadra, A., Hutter, F., and Grabocka, J. (2023). Quick-tune: Quickly learning which pretrained model to finetune and how. In *The Twelfth International Conference on Learning Representations*.

Awad, N., Mallik, N., and Hutter, F. (2021). DEHB: Evolutionary hyberband for scalable, robust and efficient Hyperparameter Optimization. In *Proc. of IJCAI'21*, pages 2147–2153.

Caron, M., Touvron, H., Misra, I., Jégou, H., Mairal, J., Bojanowski, P., and Joulin, A. (2021). Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9650–9660.

Cui, Y., Song, Y., Sun, C., Howard, A., and Belongie, S. (2018). Large scale fine-grained categorization and domain-specific transfer learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4109–4118.

Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient Hyperparameter Optimization at scale. In *Proc. of ICML'18*, pages 1437–1446.

Gardner, J., Pleiss, G., Weinberger, Q., Bindel, D., and Wilson, A. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Proc. of NeurIPS'18*, pages 7576–7586.

Harris, C., Millman, K., van der Walt, S., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M., Brett, M., Haldane, A., del Río, J., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. (2020). Array programming with numpy. *Nature*, 585(7825):357–362.

Hollmann, N., Müller, S., Eggensperger, K., and Hutter, F. (2023). TabPFN: A transformer that solves small tabular classification problems in a second. In *Proc. of ICLR'23*.

Howard, J. (2020). imagenette.

Hu, E. J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2022). LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Jiang, J., Chen, B., Fu, B., and Long, M. (2020). Transfer-learning-library. https://github.com/thuml/Transfer-Learning-Library.

Jin, H., Song, Q., and Hu, X. (2019). Auto-keras: An efficient neural architecture search system. In Teredesai, A., Kumar, V., Li, Y., Rosales, R., Terzi, E., and Karypis, G., editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 1946–1956. ACM.

Kadra, A., Janowski, M., Wistuba, M., and Grabocka, J. (2024). Scaling laws for hyperparameter optimization. *Advances in Neural Information Processing Systems*, 36.

Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., et al. (2023). Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026.

Krause, J., Stark, M., Deng, J., and Fei-Fei, L. (2013). 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*.

Li, H., Chaudhari, P., Yang, H., Lam, M., Ravichandran, A., Bhotika, R., and Soatto, S. (2019). Rethinking the hyperparameters for fine-tuning. In *International Conference on Learning Representations*.

Li, X., Grandvalet, Y., and Davoine, F. (2018). Explicit inductive bias for transfer learning with convolutional networks. In Dy, J. G. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 2830–2839. PMLR.

Liang, Y., Wen, H., Nie, Y., Jiang, Y., Jin, M., Song, D., Pan, S., and Wen, Q. (2024). Foundation models for time series analysis: A tutorial and survey. *arXiv preprint arXiv:2403.14735*.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Marben, J., Müller, P., and Hutter, F. (2019). BOAH: A tool suite for Multi-fidelity Bayesian Optimization & analysis of hyperparameters. *arXiv:1908.06756 [cs.LG]*.

Maji, S., Kannala, J., Rahtu, E., Blaschko, M., and Vedaldi, A. (2013). Fine-grained visual classification of aircraft. Technical report.

Mangrulkar, S., Gugger, S., Debut, L., Belkada, Y., Paul, S., and Bossan, B. (2022). Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft.

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61.

Nilsback, M.-E. and Zisserman, A. (2008). Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*.

Öztürk, E., Ferreira, F., Jomaa, H. S., Scmidth-Thieme, L., Grabocka, J., and Hutter, F. (2022). Zero-shot automl with pretrained models. In *Proc. of ICML'22*, pages 1128–1135.

Paszke, A., Gross, S., Massa, F., Lerer, A., et al. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Proc. of NeurIPS'19*, pages 8024–8035.

Pineda Arango, S., Ferreira, F., A., K., F., H., and J., G. (2024). Quick-tune: Quickly learning which pretrained model to finetune and how. In *Proc. of ICLR'24*.

Pineda Arango, S. and Grabocka, J. (2023). Deep pipeline embeddings for automl. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1907–1919.

Pytorch (2024). Torchtune. https://github.com/pytorch/torchtune.

Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., and Huang, X. (2020). Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897.

Rakotoarison, H., Adriaensen, S., Mallik, N., Garibov, S., Bergman, E., and Hutter, F. (2024). In-context freeze-thaw bayesian optimization. In *Forty-first International Conference on Machine Learning*.

Stoll, D., Mallik, N., Schrodi, S., Janowski, M., Garibov, S., Abou Chakra, T., Rogalla, D., Bergman, E., Hvarfner, C., Binxin, R., Kober, N., Vallaeys, T., and Hutter, F. (2023). Neural Pipeline Search (NePS).

Tang, Z., Fang, H., Zhou, S., Yang, T., Zhong, Z., Hu, T., Kirchhoff, K., and Karypis, G. (2024). Autogluon-multimodal (automm): Supercharging multimodal automl with foundation models. *arXiv preprint arXiv:2404.16233*.

Ullah, I., Carrion, D., Escalera, S., Guyon, I. M., Huisman, M., Mohr, F., van Rijn, J. N., Sun, H., Vanschoren, J., and Vu, P. A. (2022). Meta-album: Multi-domain meta-dataset for few-shot image classification. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Wightman, R. (2019). Pytorch image models. https://github.com/rwightman/pytorch-image-models.

Wistuba, M., Kadra, A., and Grabocka, J. (2022). Supervising the multi-fidelity race of hyperparameter configurations. *Advances in Neural Information Processing Systems*, 35:13470–13484.

You, K., Kou, Z., Long, M., and Wang, J. (2020). Co-tuning for transfer learning. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Zimmer, L., Lindauer, M., and Hutter, F. (2021). Auto-Pytorch: Multi-fidelity metalearning for efficient and robust AutoDL. *TPAMI*, 43:3079–3090.

Submission Checklist

1. For all authors...

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We present the first version of `Quick-Tune-Tool` and its out-of-the-box support for image classification.

   (b) Did you describe the limitations of your work? [Yes] See Section 6.

   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 6.

   (d) Did you read the ethics review guidelines and ensure that your paper conforms to them? `https://2022.automl.cc/ethics-accessibility/` [Yes] We believe our work conforms to the guidelines.

2. If you ran experiments...

   (a) Did you use the same evaluation protocol for all methods being compared (e.g., same benchmarks, data (sub)sets, available resources)? [Yes] See Section 5.

   (b) Did you specify all the necessary details of your evaluation (e.g., data splits, pre-processing, search spaces, hyperparameter tuning)? [Yes] See Section 4 and Section 5

   (c) Did you repeat your experiments (e.g., across multiple random seeds or splits) to account for the impact of randomness in your methods or data? [Yes] We run our experiments across 10 seeds, see Section 5

   (d) Did you report the uncertainty of your results (e.g., the variance across random seeds or splits)? [Yes] See Figure 5.

   (e) Did you report the statistical significance of your results? [No] Due to the restricted number of samples (four) in our experiment's results, no meaningful statistical test is applicable. We point to the confidence intervals in Figure 5 as a proxy for significance.

   (f) Did you use tabular or surrogate benchmarks for in-depth evaluations? [N/A] We are not aware of any tabular or surrogate benchmarks for fine-tuning pretrained models.

   (g) Did you compare performance over time and describe how you selected the maximum duration? [Yes] See Figure 5. We picked the maximum duration appropriate to the dataset size and following prior work (Pineda Arango et al., 2024).

   (h) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Section A

   (i) Did you run ablation studies to assess the impact of different components of your approach? [N/A] `Quick-Tune-Tool` has not components that would make ablation studies necessary. Prior work ablates components of the QuickTune algorithm itself, see (Pineda Arango et al., 2024).

3. With respect to the code used to obtain your results...

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit versions), random seeds, an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] See `https://github.com/automl/QTT`

   (b) Did you include a minimal example to replicate results on a small subset of the experiments or on toy data? [Yes] See Section 4.

(c) Did you ensure sufficient code quality and documentation so that someone else can execute and understand your code? [Yes] See our repository.

(d) Did you include the raw results of running your experiments with the given code, data, and instructions? [Yes] We provide performance results over time in our repository.

(e) Did you include the code, additional data, and instructions needed to generate the figures and tables in your paper based on the raw results? [Yes] See our repository.

4. If you used existing assets (e.g., code, data, models)…

(a) Did you cite the creators of used assets? [Yes] See Section 5

(b) Did you discuss whether and how consent was obtained from people whose data you're using/curating if the license requires it? [N/A] Our experiments were conducted on publicly available datasets.

(c) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] We are using free, common, publicly available datasets.

5. If you created/released new assets (e.g., code, data, models)…

(a) Did you mention the license of the new assets (e.g., as part of your code submission)? [Yes] We provide a license in our code repository.

(b) Did you include the new assets either in the supplemental material or as a URL (to, e.g., GitHub or Hugging Face)? [Yes] See our code repository.

6. If you used crowdsourcing or conducted research with human subjects…

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

7. If you included theoretical results…

(a) Did you state the full set of assumptions of all theoretical results? [N/A]

(b) Did you include complete proofs of all theoretical results? [N/A]

A Supplementary Materials

Feature/Aspect	HPO Tools	Finetuning Tools	AutoML Systems
Primary Focus	Optimizing hyperparameters	Adapting pretrained models to new tasks	Automating the entire ML pipeline
User Input Required	High (defining model architecture, hyperparams)	Moderate (selecting pretrained model, data)	Low (minimal input required, fully automated)
Automation Level	Partial (automates hyperparameter search)	Partial (automates training on new data)	Full (data processing, model selection, tuning)
Use Case	Improve model performance by tuning hyperparameters	Adapt models to new tasks with small datasets	End-to-end automation for building ML models
Tools & Libraries	Optuna Hyperopt Ray Tune SMAC3 NePS	Hugging Face (Transformers, PEFT) torchtune TensorFlow Hub	AutoGluon Google AutoML H2O.ai AutoKeras Microsoft Azure AutoML
Customization	High	Moderate	Low
Expertise Required	High	Moderate	Low to Moderate
Outcome	Optimized hyperparameters for better performance	Fine-tuned model specific to the new task	Fully trained and optimized ML model
Target Audience	Data Scientists, ML Engineers, Researchers	Data Scientists, ML Engineers, Researchers	Non-experts, Business Analysts, Data Scientists

Table 2: Comparison of HPO Tools, Finetuning Tools, and AutoML Systems

Positioning Quick-Tune-Tool: Bridging Finetuning and AutoML. Hyperparameter optimization (HPO) tools, finetuning tools, and AutoML systems are crucial in machine learning, each with specific purposes but common objectives. They differ in focus, automation level, and user expertise required, but all aim to improve model performance. We provide our interpretation of this complex landscape in Table 2. Within this area, `Quick-Tune-Tool` represents a hybrid approach that integrates finetuning principles with HPO techniques and aims to offer accessible usability.

`Quick-Tune-Tool` differs from conventional HPO tools, e.g., Optuna or Ray Tune, as it supports and leverages meta-learning on top of traditional optimizes. Furthermore, `Quick-Tune-Tool` integrates libraries of pretrained models (such as the Timm library). Finally, `Quick-Tune-Tool` can integrate conventional HPO tools into its pipeline to utilize different optimization methods, which clearly differentiates `Quick-Tune-Tool` from HPO tools.

`Quick-Tune-Tool` is similar to AutoML systems as it offers a curated finetuning script and a predefined search space. Thus, `Quick-Tune-Tool` also enables an easy-to-use interface for finetuning which aims to be accessible for non-experts.

In summary, we believe that `Quick-Tune-Tool` is currently positioned on the transition of a finetuning tool towards an AutoML system.

Additional QuickTuner Notes. The tuner supports basic logging and monitoring to track the progress of the tuning process. After `Quick-Tune-Tool` terminated, the user will find the finetuned models and the intermediate results in an experiment folder.

Post-Tuning Notes. Once the fitting is done, we can access the evaluation results by calling the `statistics()` function. This provides an overview (see Figure 6) of the evaluated configurations, their scores, and additional information. Subsequently, we can either fully finetune the best configurations using the `finetune()` function or assess the performance of each trained model on new data, e.g. that was not used during training, with the `leaderboard(path_to_test_data)` function.

Resources used. Our experiments utilized an internal cluster equipped with NVIDIA GeForce RTX 2080 Ti GPUs, requiring one GPU per experiment. We conducted four experiments with four different settings and ten seeds each, totaling 400 GPU-hours.

```
+----------+---------+----------+-----------------------------------------+
| config   | score   | fidelity | model                                   | ...   |
|----------+---------+----------+-----------------------------------------|
|       10 |   0.987 |        2 | beit_large_patch16_512                  | ...   |
|        0 |   0.948 |        4 | swinv2_base_window12to24_192to384 | ...   |
|        9 |   0.803 |       36 | edgenext_x_small                        | ...   |
|      114 |   0.673 |       19 | edgenext_xx_small                       | ...   |
|        1 |   0.554 |       12 | mobilevit_xs                            | ...   |
...
+----------+---------+----------+-----------------------------------------+
```

Figure 6: Evaluation results.

Search Space. A subset of the hyperparameters (see Figure 3 and 4) defined in the search space, that was used for the image classification experiments. For a complete list, please refer to the original Quick-Tune paper (Pineda Arango et al., 2024).

Hyperparameter Group	Name	Options
Fine-Tuning Strategies	Percentage to freeze	0, 0.2, 0.4, 0.6, 0.8, 1
	Layer Decay	None, 0.65, 0.75
	Linear Probing	True, False
	Stochastic Norm	True, False
Regularization Techniques	MixUp	**0**, 0.2, 0.4, 1, 2, 4, 8
	MixUp Probability	0, 0.25, 0.5, 0.75, 1
	CutMix	0, 0.1, 0.25, 0.5, 1,2,4
	DropOut	0, 0.1, 0.2, 0.3, 0.4
Data Augmentation	Data Augmentation	None, Trivial-Augment, Random-Augment, Auto-Augment
	Auto Augment	None, v0, original
Optimizer Related	Type	SGD, SGD+Momentum, Adam, AdamW, Adamp
	Learning Rate	0.1,0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005, 0.00001
	Batch Size	2,4,8,16,32,64,128,256,512
Model	Model	See Table 4

Table 3: Search space for image classification.

Libraries Used. The `Quick-Tune-Tool` is built on top of the popular deep learning library PyTorch (Paszke et al., 2019), benefiting from its large and active open-source community. This foundation allows the use of GPyTorch (Gardner et al., 2018) for GPU-accelerated Gaussian Processes in the optimization process and easy integration with Hugging Face's model hub for pretrained models. The framework employs Pandas (McKinney, 2010) and NumPy (Harris et al., 2020) for data handling and loading, providing efficient data manipulation capabilities.

Datasets. In our experiments, we chose three datasets with a large dissimilarity concerning Imagenet, the dataset used for pretraining the models. Following previous work (Li et al., 2019), we select the datasets using Earth's Moving Distance as similarity metric (Cui et al., 2018): FGVC-Aircraft, Stanford Cars and Oxford 102 Flower. Additionally, we include Imagenette, a dataset more similar to Imagenet. We report a brief description of the datasets in Table 5.

Model Name	No. of Param. (M)	Top-1 Acc.
beit_large_patch16_512	305.67	90.691
volo_d5_512	296.09	90.610
volo_d5_448	295.91	90.584
volo_d4_448	193.41	90.507
swinv2_base_window12to24_192to384_22kft1k	87.92	90.401
beit_base_patch16_384	86.74	90.371
volo_d3_448	86.63	90.168
tf_efficientnet_b7_ns	66.35	90.093
convnext_small_384_in22ft1k	50.22	89.803
tf_efficientnet_b6_ns	43.04	89.784
volo_d1_384	26.78	89.698
xcit_small_12_p8_384_dist	26.21	89.515
deit3_small_patch16_384_in21ft1k	22.21	89.367
tf_efficientnet_b4_ns	19.34	89.303
xcit_tiny_24_p8_384_dist	12.11	88.778
xcit_tiny_12_p8_384_dist	6.71	88.101
edgenext_small	5.59	87.504
xcit_nano_12_p8_384_dist	3.05	85.025
mobilevitv2_075	2.87	82.806
edgenext_x_small	2.34	81.897
mobilevit_xs	2.32	81.574
edgenext_xx_small	1.33	78.698
mobilevit_xxs	1.27	76.602
dla46x_c	1.07	73.632

Table 4: Vision models in search space.

Dataset Name	# Samples	# Classes	Image Resolution
Oxford 102 Flower	2040	102	500 - 1168
FGVC-Aircraft	10,200	102	430 - 1188
Stanford Cars	16,185	196	57 - 3744
Imagenette	13,394	10	320

Table 5: Vision Datasets used for Evaluation.

Image-Folder. The images of the custom datasets have to be arranged in a certain way to be compatible with PyTorch ImageFolder-format, see Figure 7.

```
root/train/
|-- a
|   |-- xyz.jpg
|   |-- abc.jpg
|   |-- ...
|-- ...

root/val/
|-- a
|   |-- efg.jpg
|   |-- opq.jpg
|   |-- ...
|-- ...
```

Figure 7: ImageFolder format