The Journal of Open Source Software

Tiny-TimeNAS: Time-Series Analysis on IoT Devices with Tiny Neural Architecture Search

Patara Trirat [1]

1 School of Computing, KAIST, South Korea

DOI: N/A

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: 01 January 1970

Summary

Tiny-TimeNAS is a Python package that enables deep learning-based time-series analysis on IoT devices with a few lines of code. It provides a fast and efficient neural architecture search for deep learning-based time-series analysis on IoT devices, e.g., human activity recognition and air quality prediction. Tiny-TimeNAS aims to reduce the time consumption in developing deep neural networks for microcontrollers or IoT devices by freeing users from numerous manual trial-and-error of various datasets, model combinations, and hardware constraints. Specifically, Tiny-TimeNAS bridges the gap between hardware-aware neural architecture search (Benmeziane et al., 2021), on-device deep learning (Lin et al., 2020), and time-series analysis (Esling & Agon, 2012) by proposing a first-ever hardware-aware efficient architecture search space combined with zero-cost proxies for general time-series analysis. The package also provides a framework for developing additional easy-to-use search spaces and zero-cost proxies for different time-series analysis tasks.

Statement of need

Due to the proliferation of sensor devices, time-series analysis is increasingly essential and at the core of various real-world applications, such as classification for human activity recognition, regression for air quality prediction, and anomaly/outlier detection for industrial system monitoring (Blázquez-García et al., 2021; Foumani et al., 2023). However, making efficient deep models for various downstream analysis tasks is extremely challenging because it laboriously relies on tedious manual trial-and-error to design the network architectures and select corresponding hyperparameters, especially under highly restricted environments like IoT devices and microcontrollers having diverse deployment constraints, which requires more steps than the traditional workflow.

To address this problem, researchers employ hardware-aware neural architecture search (NAS) to automatically discover an optimal architecture for a specific platform and downstream task. Despite the rapid advancement in NAS (White et al., 2023) in the past years, no work focuses on general time-series analysis. However, recent studies on *tiny* deep learning for IoT devices (Benmeziane et al., 2021; Lin et al., 2020, 2021) only use existing search spaces designed for computer vision tasks that could not be directly adopted for time-series analysis. In addition, existing time series-relevant search spaces focusing only on an analysis (e.g., forecasting (Deng et al., 2022; Lai et al., 2023) or classification (Rakhshani et al., 2020; Ren et al., 2022)) cannot generalize well in different tasks or even cannot be run on IoT devices. Therefore, neural architecture search and search space for on-device time-series analysis are crucial to facilitate deep learning-based analysis and significantly lower the time consumption in neural architecture design and evaluation given the large amounts of various time-series data nowadays.

The Journal of Open Source Software

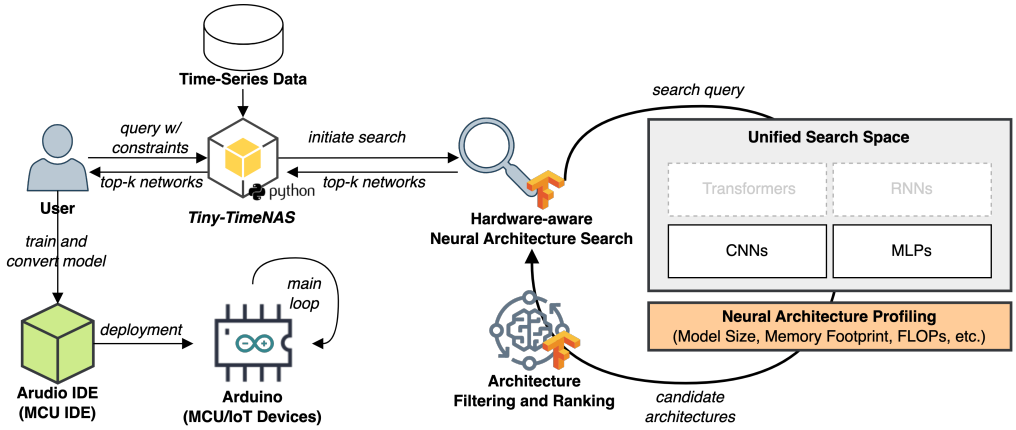

Figure 1: Overall procedure of `Tiny-TimeNAS`.

Features

`Tiny-TimeNAS` aims to provide a TinyML-based hardware-aware neural architecture search for general-purpose time-series analysis, e.g., human activity recognition and humidity forecasting, that can be run on IoT devices given various constraints. As in Figure 1, `Tiny-TimeNAS` mainly consists of the following ***three*** parts.

- **Tiny-TimeNAS API**: This component is the core application programming interface (API) that enables users to communicate with the underlying processes, such as architecture search, performance evaluation, and model conversion.
- **Hardware-aware NAS**: This part is responsible for the entire search process after receiving a specific query or command from the user. Here, one baseline (FLOPs) (Krishnakumar et al., 2022) and one state-of-the-art (ZiCo) (Li et al., 2023) zero-cost proxies are implemented to accelerate the search time while having high-performing architectures.
- **Unified Search Space**: This element unifies different types of neural networks designed based on preliminary experimental results for time-series analysis. These results are further normalized and used as the selection probability for a given task to enhance the search efficiency.

Table 1: Summary of available search space.

Network Architecture	Option	Selection Prob.	
		Classification	**Regression**
MLP	Units: [8, 16, 32, 64, 128, 256]	0.4516	*0.5055*
CNN	Filters: [8, 16, 32, 64, 128, 256], Kernel Size: [2, 3, 5, 7, 9], Strides: [1, 2, 3]	*0.5484*	0.4945

Search Space

In the current version, the package only contains multi-layer perceptrons (or fully-connected layer) and convolutional neural networks due to the unsupported operations in Tensorflow Lite Micro. Table 1 summarizes the currently available search space with selection probabilities.

Core Functions

Assuming that we have a set of training (`x_train`) time series and its labels (`y_train`), we can easily run the search using the following two lines of code.

```
searcher = Searcher(max_ram=128, max_flash=256, "classification", x_train, y_train)
cand_models = searcher.run_search(n_archs=100, top_k=1, proxy="zico")
```

Here, we set the constraints to have a set of 100 candidate models that use less than 128KB of memory and 256KB of flash storage for the classification task. Then, only the best model based on the ZiCo score will be selected. After we get the `cand_models`, we can access the `Model` object class via `model = cand_models[0]["model"]` as in the original Keras API. For example, we can run `model.compile(...)` or `model.summary()` directly to the model object. Thus, any Keras API is compatible with the found architecture(s).

Subsequently, if the model is successfully trained and evaluated with the given time-series datasets, we can further convert or quantize the model via `quantize_model(...)`. Then, further convert it to the C++ bytearrays through the command line interface.

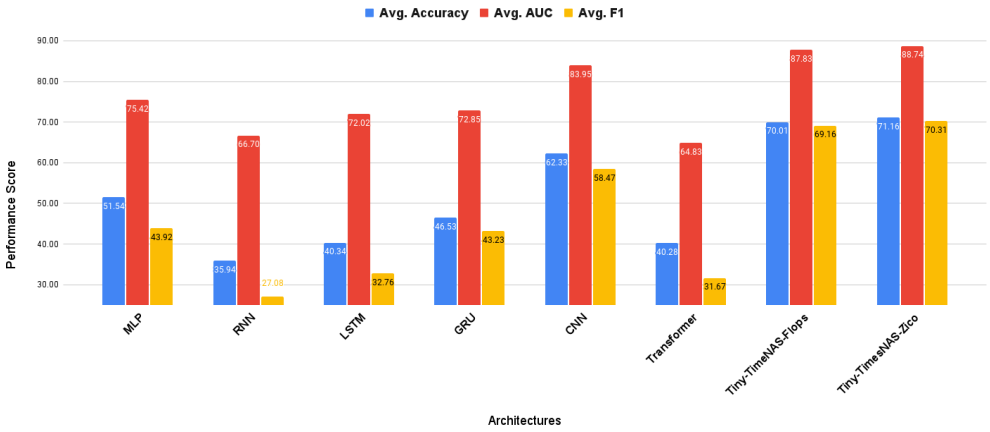

Figure 2: Performance comparison with manual designs.

Performance Evaluation

Finally, to verify whether the discovered architecture(s) is really useful for the given analysis task, we empirically compare the classification performance with the manually designed architectures. Averaging from ***five*** datasets — AsphaltObstacles, BasicMotions, Handwriting, MotionSenseHAR, and UWaveGestureLibrary from https://timeseriesclassification.com — we can observe that models found by `Tiny-TimeNAS` (either FLOPs or ZiCo) significantly outperform the manually designed models under the similar constraints as shown in Figure 2.

Application: Human Activity Recognition for Arduino Nano

This section provides a step-by-step walkthrough for an application use case on the actual IoT device. First, we start by importing the necessary packages and libraries.

```
import os, sys, gc, warnings, logging, json, time, glob, math, random
import tensorflow as tf
import pandas as pd
import numpy as np

from tensorflow import keras
```

```
from tiny_tnas.searcher import Searcher
from tiny_tnas.converters import quantize_model
from tiny_tnas.profiler import get_model_size
from tiny_tnas.data_loader import load_dataset, AVAILABEL_DATASETS
from tiny_tnas.evaluator import evaluate_classification
```

Then, define the target task and dataset. This example dataset has 3D accelerometer and a 3D gyroscope (i.e., 6 features) collected from a smart watch. It consists of four classes, which are walking, resting, running and badminton.

```
task = 'classification' # define the target analysis task
x_train, y_train, x_test, y_test = load_dataset(task, data_name='BasicMotions')
```

After we successfully load the dataset, we can run the search similarly to the process described earlier to get the final top-k model and train it. Here, `MAX_RAM`, `MAX_FLASH`, and `N_CANDIDATES` indicate the search hyperparameters for maximum memory usage, maximum storage usage, and the number of sampling candidates, respectively.

```
# run the search
searcher = Searcher(MAX_RAM, MAX_FLASH, task, x_train, y_train)
cand_models = searcher.run_search(N_CANDIDATES, top_k=1, proxy='zico')

# compile the found model
model = cand_models[0]['model']
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['sparse
model.fit(x_train, y_train)
```

After model training, we can evaluate the model with unified metrics for a specific task, i.e., classification as follows.

```
y_pred = model.predict(x_test)
perf = evaluate_classification(y_test, y_pred)
```

We can access the performance through a dictionary key, e.g., `perf["accuracy"]` or `perf["f1"]`.

If we are satisfied with the model performance, we can convert and/or quantize the trained model with the following commands.

```
quantized_model = quantize_model(model, x_train, mode='int')
open(MODEL_TFLITE, "wb").write(quantized_model)
```

Here, `MODEL_TFLITE` indicates the Tensorflow Lite model's filename, where `MODEL_TFLITE_MICRO` indicates the Tensorflow Lite Micro filename to be deployed on the device.

```
!xxd -i {MODEL_TFLITE} > {MODEL_TFLITE_MICRO}
REPLACE_TEXT = MODEL_TFLITE.replace('/', '_').replace('.', '_')
!sed -i 's/'{REPLACE_TEXT}'/g_model/g' {MODEL_TFLITE_MICRO}
```

Finally, we can now deploy this ***micro*** model to the device. Note that we can also apply the exact same procedure to other tasks (e.g., regression and anomaly detection) by simply changing the `task`, given dataset, and evaluation metrics.

Limitations and Future Work

Even though we have found that different neural network types work differently on a given task; for example, CNN works best for classification, Transformer works best for regression, and gated recurrent unit (GRU) works best for anomaly detection; we still cannot include them all due to a very limited number of operations in Tensorflow Lite Micro, especially the recurrent

neural network (RNN) family and multi-head attention layer. The anomaly detection task is also currently excluded because of the different learning schemes (specifically, unsupervised learning with autoencoders) and the requirement of post-training processes that need more low-level configurations for model profiling, threshold selection, anomaly scoring function, and so on. In future work, `Tiny-TimeNAS` will include more time series-specific networks and model profiling, such as RNN- and Transformer-based architectures, by low-level custom operations. To further increase usability, the graphical user interfaces will be provided, along with a fully "end-to-end" source file generation using MicroPython.

References

Benmeziane, H., El Maghraoui, K., Ouarnoughi, H., Niar, S., Wistuba, M., & Wang, N. (2021). Hardware-aware neural architecture search: Survey and taxonomy. In Z.-H. Zhou (Ed.), *Proceedings of the thirtieth international joint conference on artificial intelligence, IJCAI-21* (pp. 4322–4329). International Joint Conferences on Artificial Intelligence Organization. https://doi.org/10.24963/ijcai.2021/592

Blázquez-García, A., Conde, A., Mori, U., & Lozano, J. A. (2021). A review on outlier/anomaly detection in time series data. *ACM Computing Surveys (CSUR)*, *54*(3), 1–33.

Deng, D., Karl, F., Hutter, F., Bischl, B., & Lindauer, M. (2022). Efficient automated deep learning for time series forecasting. In M.-R. Amini, S. Canu, A. Fischer, T. Guns, P. Kralj Novak, & G. Tsoumakas (Eds.), *Machine learning and knowledge discovery in databases* (pp. 664–680). Springer Nature Switzerland. ISBN: 978-3-031-26409-2

Esling, P., & Agon, C. (2012). Time-series data mining. *ACM Computing Surveys (CSUR)*, *45*(1), 1–34.

Foumani, N. M., Miller, L., Tan, C. W., Webb, G. I., Forestier, G., & Salehi, M. (2023). Deep learning for time series classification and extrinsic regression: A current survey. *arXiv Preprint arXiv:2302.02515*.

Krishnakumar, A., White, C., Zela, A., Tu, R., Safari, M., & Hutter, F. (2022). NAS-bench-suite-zero: Accelerating research on zero cost proxies. *Thirty-Sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=yWhuljljH8k

Lai, Z., Zhang, D., Li, H., Jensen, C. S., Lu, H., & Zhao, Y. (2023). LightCTS: A lightweight framework for correlated time series forecasting. *Proceedings of the 2023 International Conference on Management of Data*.

Li, G., Yang, Y., Bhardwaj, K., & Marculescu, R. (2023). ZiCo: Zero-shot NAS via inverse coefficient of variation on gradients. *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=rwo-ls5GqGn

Lin, J., Chen, W.-M., Cai, H., Gan, C., & Han, S. (2021). Memory-efficient patch-based inference for tiny deep learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Liang, & J. W. Vaughan (Eds.), *Advances in neural information processing systems* (Vol. 34, pp. 2346–2358). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2021/file/1371bccec2447b5aa6d96d2a540fb401-Paper.pdf

Lin, J., Chen, W.-M., Lin, Y., cohn, john, Gan, C., & Han, S. (2020). MCUNet: Tiny deep learning on IoT devices. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (Vol. 33, pp. 11711–11722). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2020/file/86c51678350f656dcc7f490a43946ee5-Paper.pdf

Rakhshani, H., Ismail Fawaz, H., Idoumghar, L., Forestier, G., Lepagnot, J., Weber, J., Brévilliers, M., & Muller, P.-A. (2020). Neural architecture search for time series

classification. *2020 International Joint Conference on Neural Networks (IJCNN)*, 1–8. https://doi.org/10.1109/IJCNN48605.2020.9206721

Ren, Y., Li, L., Yang, X., & Zhou, J. (2022). AutoTransformer: Automatic transformer architecture design for time series classification. In J. Gama, T. Li, Y. Yu, E. Chen, Y. Zheng, & F. Teng (Eds.), *Advances in knowledge discovery and data mining* (pp. 143–155). Springer International Publishing. ISBN: 978-3-031-05933-9

White, C., Safari, M., Sukthanker, R., Ru, B., Elsken, T., Zela, A., Dey, D., & Hutter, F. (2023). Neural architecture search: Insights from 1000 papers. *arXiv Preprint arXiv:2301.08727*.