

tfaip - a Generic and Powerful Research Framework for Deep Learning based on Tensorflow

Christoph Wick¹, Benjamin Kühn¹, Gundram Leifert¹, Konrad Sperfeld², Tobias Strauß¹, Jochen Zöllner^{1,2}, and Tobias Grüning¹

¹ Planet AI GmbH, Warnowufer 60, 18059 Rostock, Germany ² Institute of Mathematics, University of Rostock, 18051 Rostock, Germany

DOI: [DOIunavailable](#)

Software

- [Review](#) ↗
- [Repository](#) ↗
- [Archive](#) ↗

Editor: [Pending Editor](#) ↗

Reviewers:

- [@Pending Reviewers](#)

Submitted: N/A

Published: N/A

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

Summary

tfaip is a Python-based research framework for developing, structuring, and deploying Deep Learning projects powered by Tensorflow (Abadi et al., 2015) and is intended for scientists of universities or organizations who research, develop, and optionally deploy Deep Learning models. *tfaip* enables to implement both simple and complex scenarios, such as image classification, object detection, text recognition, natural language processing, or speech recognition. Each scenario is highly configurable by parameters that can directly be modified by the command line or the API.

Statement of Need

An application of *tfaip* resolves recurrent obstacles of research and development in an elegant and robust way.

A complete scenario including the graph (e.g., the network architecture), the training (e.g., the optimizer or learning rate schedule), and the data pipeline (e.g., the data sources) are fully parameterizable by the command line or the API. This simplifies hyper-parameter-optimization but also testing of novel ideas. Each scenario is created by implementing predefined interfaces (e.g., loss-function or the graph construction). This leads to a clean, structured, modularized, and readable code preventing bad code style and facilitates maintenance.

tfaip provides a simple API to deploy a scenario. The corresponding module will automatically apply pre-processing, infer the trained model, and optionally transform the output by a post-processing pipeline. The information about the pipeline-construction is embedded within the model which enables to store and load models with a different data pipeline even for the same scenario. This is handy if, for example, certain pre-processing steps are not required for one specific model or other inputs are expected.

During research, a tedious step is data preparation which often comprises the conversion of data into the format required by the framework. Tensorflow allows integrating Python code in the data pipeline which is however not run (truly) in parallel by multiple processes and results quite often in a bottleneck. To speed-up Tensorflow, a user has to transform Python into Tensorflow operations which is laborious, partly even impossible, and complicates debugging. *tfaip* tackles this issue by providing a sophisticated pipeline setup based on so-called data processors which apply simple transformation operations in pure Python code and are automatically executed in parallel.

State of the Field

Efficient research in the area of Deep Learning requires the integration of highly sophisticated Open-Source frameworks such as Tensorflow (Abadi et al., 2015), PyTorch (Paszke et al.,

2019), Caffe (Jia et al., 2014), CNTK (Seide & Agarwal, 2016), or Trax (Google Inc., 2021). These frameworks provide efficient tools to freely design Deep Learning scenario of any size and complexity. However, as the number of code lines rises, a project has to be structured into meaningful components to be maintainable. These components are almost identical for each Deep Learning scenario: there are modules for the graph, the model, the data, the training, the validation, and the prediction (i.e., the application of a trained model). Furthermore, support for dynamic parameter adaption for instance via the command line is desirable for efficient research. Therefore, to obtain a clean code base, it is highly desirable to only implement abstract templates that already set up the interaction among the modules by providing basic functionality that is required in any use-case. *tfaip* which is an extension to Tensorflow solves this and thus helps developers to efficiently handle and maintain small but also large-scale projects in research environments.

Recently, several AutoML approaches emerged, e.g., by Google Cloud or Microsoft Azure. AutoML targets developers with limited machine learning expertise and enables them to train their own models by automating processes like network construction, feature engineering, or hyperparameter tuning. In contrast, *tfaip* targets researchers with expertise in deep learning who actually design new network architectures, data processing pipelines, and setup training, but with only limited experience in or capacity for software engineering. *tfaip* helps to structure and maintain the code bases, and hereby also solves some recurrent problems that will likely occur during development (see next Section).

***tfaip* Functionality**

In the following, we highlight the main functionality of *tfaip*.

A basic concept of *tfaip* is to split parameters and their actual object whereby the parameters are used to instantiate the corresponding object. This allows to build a hierarchical parameter tree where each node can be replaced with other parameters. Each parameter and also the replacements can be defined via the command line which enables to dynamically adapt, for example, even complete graphs of a model. In a research environment this simplifies hyperparameter search and the setup of experiments.

Class-templates define the logical structure of a real-world scenario. Each scenario requires the definition of a model and a data class whereby basic functionality such as training, exporting, or loading of a model using the data is already provided.

To set up the data pipeline, a data generator and a list of data processors have to be implemented that define how raw data flows. The prepared data is then automatically fed into the neural network. Since each data processor is written in pure Python, it is simple to set up and debug the data processing pipeline. To speed up the data sample generation the pipeline can automatically be run in parallel.

The model comprises information about the loss, metrics, and the graph of the scenario. Its template hereby requires the user to implement methods, the superordinate modules are then connected automatically.

tfaip tracks the training and validation process by utilizing the Tensorboard provided by Tensorflow. The Tensorboard can be extended by custom data, for example by plotting Precision-Recall (PR) curves or rendering images.

The prediction module loads a trained scenario so that it can easily be applied on new data during deployment. Since a model stores its pre- and post-processing pipeline no additional data handling has to be performed.

Because each scenario follows a predefined setup, shared research code is clearer and consequently can be easier reviewed, extended, or applied. For example, this modularity simplifies the process if several users are working together on the same scenario.

An important feature of *tfaip* is the consistent use of Python's typing module including type checks. This leads to clean understandable code and fewer errors during development. Furthermore, this enables IDEs such as PyCharm (JetBrains, 2021) to perform autocompletion.

In some rare cases, the highly generic API of *tfaip* might not be sufficient. To tackle this, each scenario can optionally customize any functionality by implementing the base classes, for example the trainer or the data pipeline.

***tfaip* Documentation and Tutorials**

To help new users to become familiar with *tfaip*, a comprehensive documentation, several tutorials, and example scenarios with real-world use-cases are available. First, *tfaip* provides two tutorials that solve the classification of MNIST: the *minimal* scenario shows the minimal implementation that is required to implement the common MNIST-tutorial in *tfaip*, the *full* scenario implements the same use-case and highlights different advanced features of *tfaip*.

Some more examples are provided by transferring official Tensorflow tutorials in the *tfaip* framework. These scenarios show the power of the framework as complex scenarios are logically split into meaningful parts and components. The examples comprise Automatic Text Recognition (ATR) of single text line images, Image Classification, and Fine Tuning of a BERT.

Finally, templates for two basic scenarios allow setting up a new scenario by copying basic code and modifying it afterwards. All required files and classes are already set up, solely the abstract methods have to be implemented and classes should be renamed.

Usage of *tfaip* in Research

Diverse active research projects are already based on *tfaip*. Zöllner et al. (2021) integrated *tfaip* to solve Natural Language Processing (NLP) problems. Since its 2.0 release, the open-source ATR engine Calamari by Wick et al. (2020) is based on *tfaip*. Our research, for example a recent publication on ATR using Transformers, uses *tfaip* (Wick et al., 2021).

Acknowledgments

The authors would like to thank the open-source community, especially the developers and maintainers of Python, Tensorflow, and Numpy, since these packages empower *tfaip*.

This work was partially funded by the European Social Fund (ESF) and the Ministry of Education, Science and Culture of Mecklenburg-Western Pomerania (Germany) within the project NEISS under grant no ESF/14-BM-A55-0006/19.

References

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., ... Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems*. <https://www.tensorflow.org/>
- Google Inc. (2021). Trax. In *GitHub repository*. <https://github.com/google/trax>; GitHub.
- JetBrains. (2021). *PyCharm*. <https://jetbrains.com/pycharm>.
- Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., & Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv Preprint arXiv:1408.5093*.

- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems* 32 (pp. 8024–8035). Curran Associates, Inc. <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- Seide, F., & Agarwal, A. (2016). *CNTK: Microsoft's open-source deep-learning toolkit*. 2135–2135. <https://doi.org/10.1145/2939672.2945397>
- Wick, C., Reul, C., & Puppe, F. (2020). Calamari - A High-Performance Tensorflow-based Deep Learning Package for Optical Character Recognition. *Digital Humanities Quarterly*, 14(1).
- Wick, C., Zöllner, J., & Grüning, T. (2021). Bidirectional Transformer for Handwritten Text Recognition. *16th International Conference on Document Analysis and Recognition (ICDAR)*, accepted for.
- Zöllner, J., Sperfeld, K., Wick, C., & Labahn, R. (2021). Optimizing small BERTs trained for german NER. *Computational Linguistics*, submitted to.