FastEstimator: A Deep Learning Library for Fast Prototyping and Productization

Xiaomeng Dong, Junpyo Hong, Hsi-Ming Chang, Michael Potter, Aritra Chowdhury*,
Purujit Bahl, Vivek Soni, Yun-Chan Tsai, Rajesh Tamada, Gaurav Kumar,
Caroline Favart, V. Ratna Saripalli, Gopal Avinash
GE Healthcare
*GE Research

Abstract

As the complexity of state-of-the-art deep learning models increases by the month, implementation, interpretation, and traceability become ever-more-burdensome challenges for AI practitioners around the world. Several AI frameworks have risen in an effort to stem this tide, but the steady advance of the field has begun to test the bounds of their flexibility, expressiveness, and ease of use. To address these concerns, we introduce a radically flexible high-level open source deep learning framework for both research and industry. We introduce FastEstimator.

1 Introduction

Over the last two decades, interest in Artificial Intelligence (AI) has grown significantly [1]. This rise in interest has greatly expanded the AI community and created an ever increasing demand for AI systems. Simultaneously, the requirements of these systems are becoming diversified, which poses a challenge in designing a general framework for the AI community.

Researchers and experts require ultimate flexibility from AI systems since their goal is to explore new ideas and discover uncharted territory in AI. Frameworks like TensorFlow [2], Caffe [3], MXNet [4], CNTK [5], and PyTorch [6] gained the favor of expert users because they make few assumptions regarding user behaviors and allow users to control fine-grained details by building experiments from the ground up. However, building AI from scratch may result in unnecessary verbosity and redundant efforts. Therefore, a framework which preserves flexibility while removing these redundancies will be preferable for experts and researchers—improving their productivity.

Entrepreneurs, beginners, and enterprise users tend to favor AI systems that have lower learning curves and faster time to deployment. High-level frameworks like Keras [7], Gluon [4], fastai [8], and Ludwig [9] are examples of such systems. The benefit of a higher-level framework is obviously ease of use; however, simplicity comes at the expense of flexibility. Furthermore, as more and more new ideas in AI have proven useful in real-world applications, high-level frameworks are not evolving fast enough to serve the industry's interests in these ideas. For example, there are few high-level frameworks that provide flexible support for generative adversarial network (GAN) applications and progressive training schemes. As a result, gaps are forming between state-of-the-art (SOTA) and ease of use.

We therefore introduce FastEstimator, an open source high-level deep learning library made for both research and industry. Our intent is to bridge the gap by providing a simple yet flexible interface for implementing SOTA ideas, which we hope will help different groups within the AI community to move forward together. For researchers, FastEstimator will continuously monitor the latest advancements in AI to provide an easy and flexible interface. For industry, FastEstimator can enable more SOTA AI products and shorten the product development cycle.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

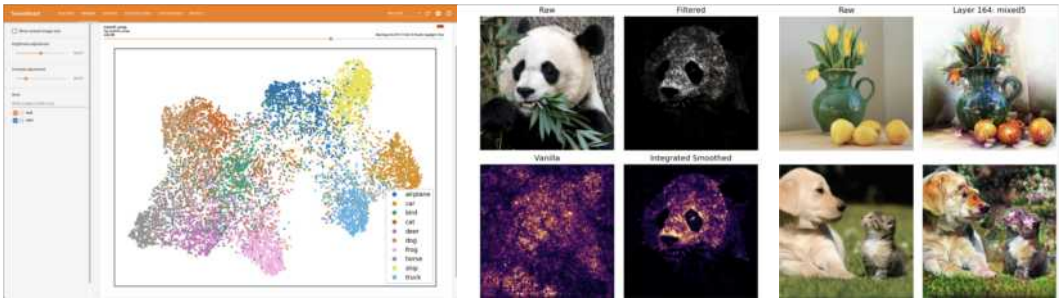
2 Highlights of FastEstimator

Multi-Framework. FastEstimator is built on TensorFlow2 [2] and PyTorch [6] with a unique multi-framework design. Libraries like Keras [7] handle multiple frameworks by unifying different backends with consistent APIs. In contrast, FastEstimator uses the best components from different backends. For example, our data preprocessing module uses a combination of `torch.DataLoader` and `tf.data`. As a result, FastEstimator can leverage the advantages of both frameworks and provide a good trade-off between speed and flexibility.

Simple yet Flexible. FastEstimator users only need to interact with three main APIs (see Sec. 3) for any deep learning task. Besides these, the `Operator` module (see Sec. 4.1) allows users to define a complex computational graph in a concise manner. The `Trace` module (see Sec. 4.2) provides users with further control over the training loop. We will show in Sec. 5 that FastEstimator can significantly reduce the effort required to implement several deep learning tasks.

Easy to Scale. Distributed training in many deep learning frameworks requires non-trivial effort on the user's side. For example, users are expected to understand device communication patterns and rewrite their workflows to be distribution-aware. In FastEstimator, all modules are designed to be distribution-friendly such that users can scale their training and evaluation across multiple devices without any change of code.

Useful Utility AI Tools. Besides the training APIs, FastEstimator offers useful AI utility functions to facilitate the prototyping and production process. For example, the model interpretation module contains visualization tools such as feature UMAP [10], saliency maps [11, 12] and caricature maps [13] to help users build more robust models (Fig. 1). All utility modules have full compatibility with TensorBoard. We also provide utility tools for automatic report generation for documentation purposes.



(a) Activation UMAP (b) Saliency Map (c) Caricatures

Figure 1: Visualization utility tools

Application Hub. Some frameworks [2, 4, 6, 7] provide a model zoo, which allows users to import pre-built model architectures and weights. However, it is often the case that the true complexity of implementing a new idea lies more on the data pipeline and training loop than on the model architecture itself. For this reason, FastEstimator provides Application Hub—a place to showcase different end-to-end AI applications. Every template in Application Hub has step-by-step instructions to ensure users can easily build new AI applications with their own data.

3 Architecture Overview

All deep learning training workflows involve three essential components: data pipeline, network, and optimization strategy. Data pipeline extracts data from disk to RAM, performs transformations, and then loads the data onto the device. Network stores trainable and differentiable graphs. Optimization strategy combines data pipeline and network in an iterative process. Each of these components represents a critical API in FastEstimator: `Pipeline`, `Network`, and `Estimator` (Fig. 2). Users will interact with these three APIs for any deep learning task.

`Pipeline` can be summarized as an Extraction-Transformation-Load (ETL) process. The extractor can take data either from disk or RAM, with features being either paired or unpaired (e.g., domain

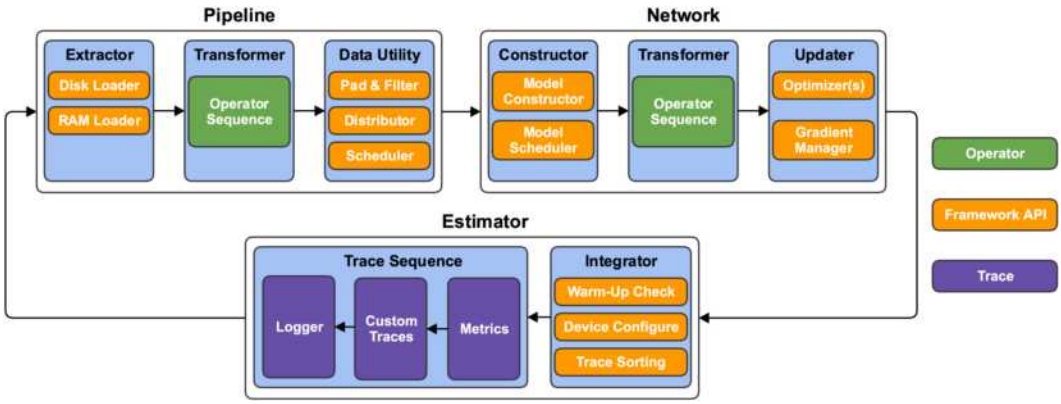


Figure 2: FastEstimator architecture overview

adaptation). The transformer builds graphs for preprocessing. The data utility provides support for scenarios like imbalanced training, feature padding, distributed training, and progressive training.

`Network` manages trainable models. First, the constructor builds model graphs and creates timestamps on these graphs in the case of progressive training. The transformer then connects different pieces of model graphs and non-trainable graphs together. The updater tracks and applies gradients to each trainable model.

`Estimator` is responsible for the training loop. Before training starts, a smoke test is performed on all graphs to detect potential run-time errors as well as to warm up the graph for faster execution. It then proceeds with training, generating any user-specified output along the way.

The central component of both `Pipeline` and `Network` is a sequence of `Operators`. Similarly for the `Estimator`, there is a sequence of `Traces`. `Operator` and `Trace` are both core concepts which differentiate FastEstimator from other frameworks.

4 Core Concepts

4.1 Operator

The common goal of all high-level deep learning APIs is to enable complex graph building with less code. For that, most frameworks like Keras [7], MXNet [4], and PyTorch [6] introduce the concept of layers (aka blocks and modules) to simplify network definition. However, as model complexity increases, even layer representations may become undesirably verbose—for example when expressing multiple time-dependent model connections. Therefore, we propose the concept of `Operator`, a higher level abstraction for layers, to achieve code reduction without losing flexibility.

An `Operator` represents a task-level computation module for data (in the form of key:value pairs), which can be either trainable or non-trainable. Every `Operator` has three components: input key(s), transformation function, and output key(s). The execution flow of a single `Operator` involves: 1) take the value of the input key(s) from the batch data, 2) apply transformation functions to the input value, 3) write the output value back to the batch data with output key(s). Fig. 3 shows how a deep learning application can be expressed as a sequence of `Operators`.

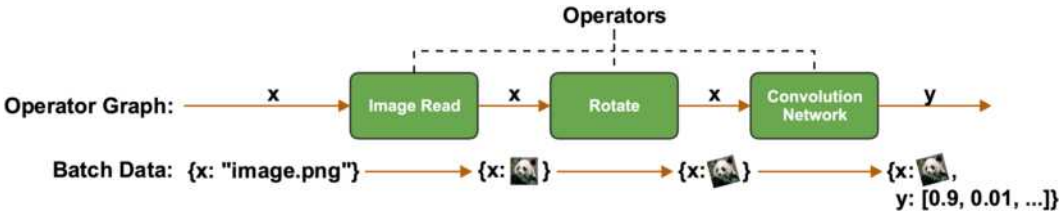


Figure 3: A sequence of `Operators` reading and modifying batch data

FastEstimator offers concise `Operator` expressions that allow users to construct various graph topologies in an efficient way (Fig. 4). With the help of `Operators`, complex computational graphs can be built using only a few lines of code. We will further demonstrate the convenience of `Operators` in Sec. 5 on different deep learning applications.

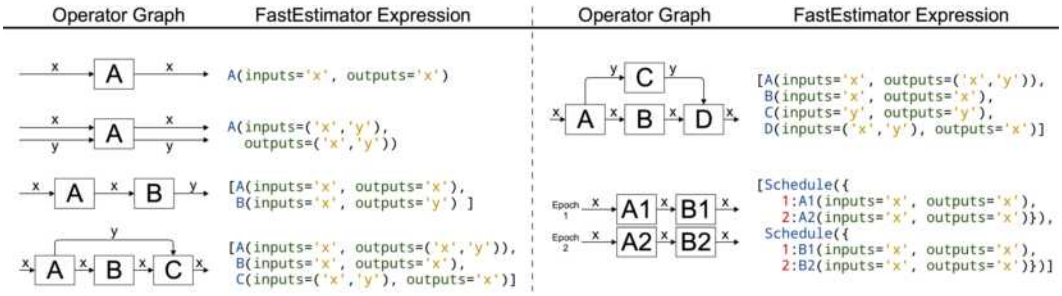


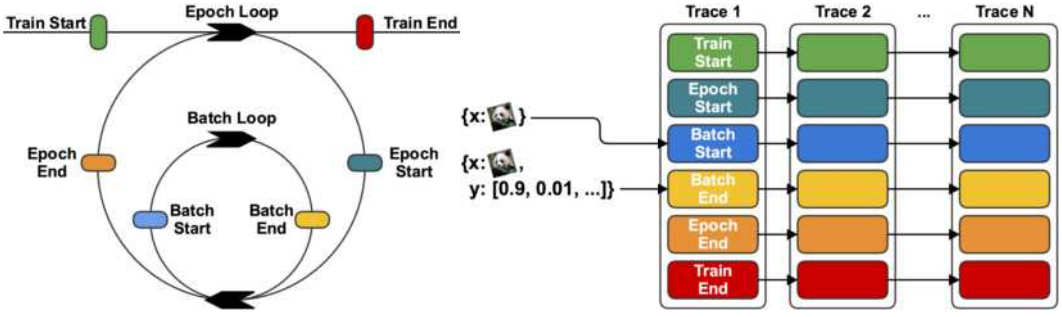
Figure 4: `Operator` graphs and FastEstimator expressions

4.2 Trace

Some APIs [2, 6, 7] offer two separate modules: metrics and callbacks (aka hooks and handlers). In FastEstimator, both metrics and callbacks are unified into `Traces`. As a result, we can effectively overcome several limitations introduced by separating metrics and callbacks.

Metrics are quantitative measures of model performance and are computed during the training or validation loop. From the implementation perspective, APIs tend to implement metrics as a built-in computational graph with two parts: a value and an update rule. While this pre-compiled graph enables faster computation, it also limits the choice of metrics. For example, some domain-specific metrics are not easily expressed as a graph or require running external post-processing libraries. Furthermore, the benefit offered by pre-compiling metric graphs is not significant, because these calculations only account for a small portion of the system's total computation.

Callbacks are modules that contain event functions like `on_epoch_begin` and `on_batch_begin`, which allow users to insert custom functions to be executed at different locations within the training loop (Fig. 5a). Implementation-wise, since metrics and callbacks are separate, callbacks in most frameworks are not designed to have easy access to batch data. As a result, researchers may have to use less efficient workarounds to access intermediate results produced within the training loop. Moreover, callbacks are not designed to communicate with each other, which adds further inconvenience if a later callback needs the outputs from a previous callback.



(a) Events within the training loop

(b) `Trace` communications on events

Figure 5: `Traces` throughout training

`Trace` in FastEstimator is a unification of metrics and callbacks; it preserves the event functions in callbacks and overcomes the aforementioned limitations through the following improvements: 1) `Traces` have easy access to batch data directly from the batch loop, 2) every `Trace` can pass data to later `Traces` to increase re-usability of results as shown in Fig. 5b, 3) metric computation can leverage batch data directly without a graph, 4) metrics can be accumulated through `Trace` member

variables without update rules. These improvements brought by `Trace` have enabled many new functionalities that are not easily achieved with conventional callbacks. For example, our model interpretation module is made possible by easy batch data access. Furthermore, `Trace` has access to all API components such that changing model architecture or data pipeline within the training loop is straightforward. This can unlock support for Meta-Learning, Reinforcement Learning (RL), and AutoML algorithms.

5 Examples and Applications

In this section, we show FastEstimator `Operator` expressions for various deep learning tasks. All source code is available on `github.com/fastestimator`.

Image Classification. We start with an image classification example (Fig. 6). `MinMax` applies normalization to input images before the model forward pass, the system calculates the gradients from the provided loss function, and performs back propagation.

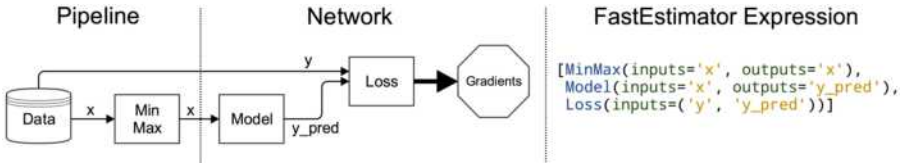


Figure 6: An image classification example, where the input image has key `x` and label key `y`

Image Classification with Progressive Resizing. Progressive resizing techniques have been applied to super-resolution [14] and GAN training [15]. They start with low resolution images and gradually increase the image resolution as training progresses. Fig. 7 shows an example where we increase the image resolution by a factor of 2 on epoch 2 and 4.

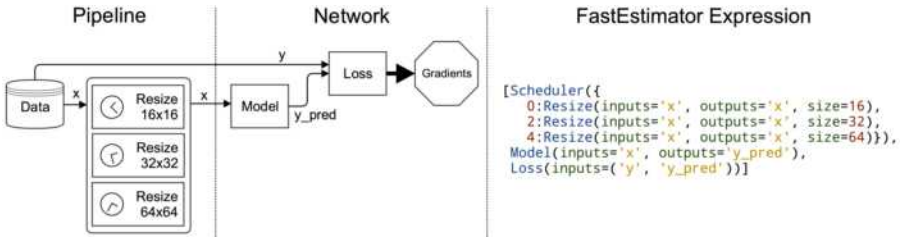


Figure 7: Image classification with progressive resizing

Image Classification with Adversarial Training. It is also easy to apply modifications to the training process during the network forward pass. For example, training a neural network using input images perturbed by an adversarial attack—which has been shown to make models more robust to future adversarial attacks [16]—can be accomplished with only four extra `Operators`. This makes it easy to add adversarial training to any model (Fig. 8).

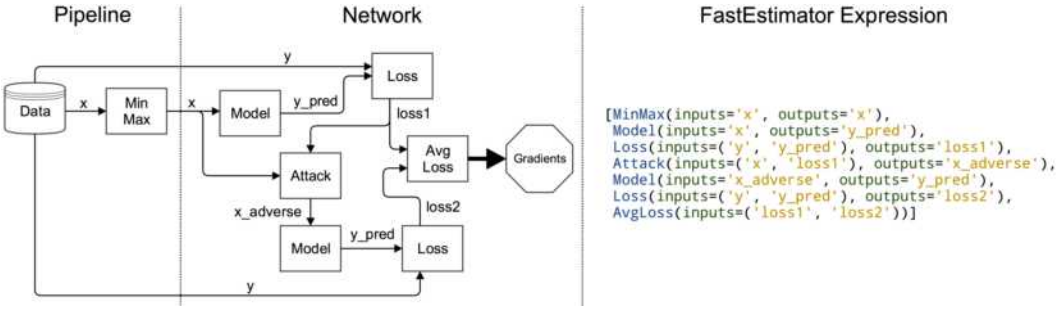


Figure 8: Image classification with adversarial training

Image Generation with Deep Convolutional GAN. For multi-model training such as DC-GAN [17], users can associate different losses to different models. The gradients are calculated with respect to each loss and the system will perform updates for each model (Fig. 9).

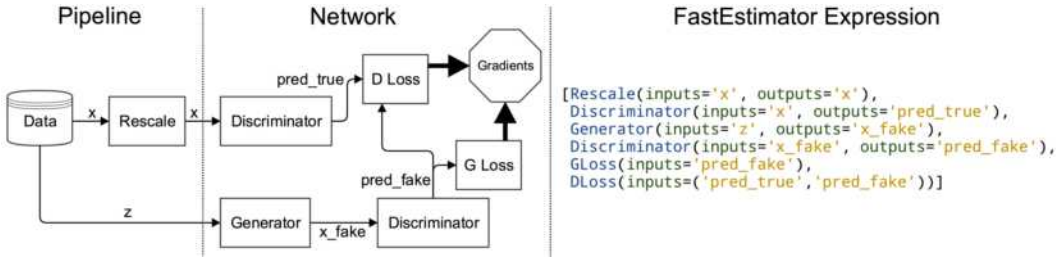


Figure 9: Image generation with DC-GAN, with real image as `x` and random noise as `z`

Image Generation with Cycle-GAN. For a more complicated problem such as unsupervised unpaired image translation using the Cycle-GAN [18], users often need to define different losses that involve outputs of multiple models. FastEstimator `Operators` make it easy to break down such complex interactions between different generators and discriminators (Fig. 10).

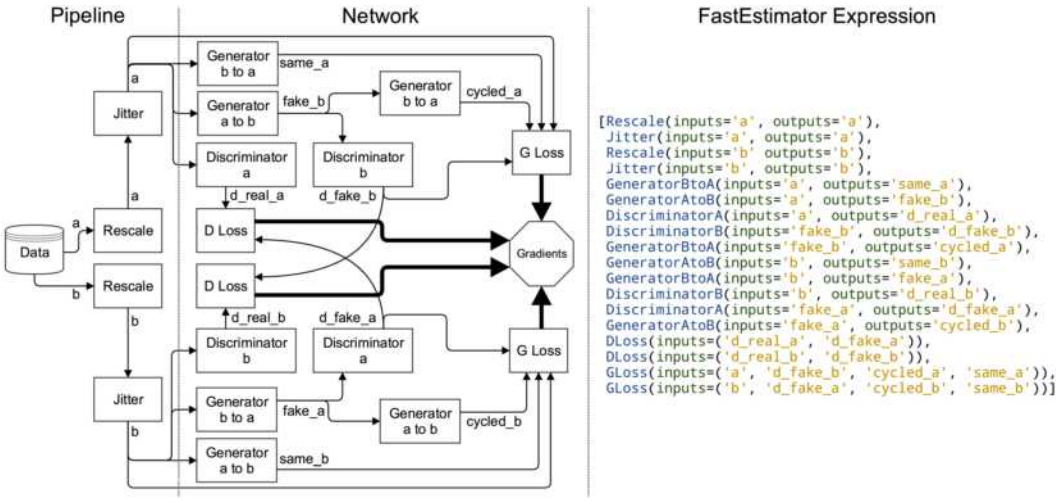


Figure 10: Image generation with Cycle-GAN, with `a` and `b` as unpaired images from two domains

6 Future

Going forward we intend to expand FastEstimator into the deep RL domain by integrating our API with existing RL libraries such as those provided by OpenAI [19] and TensorFlow [2]. We also intend to provide more support for AutoML and Meta-Learning. Finally, we will continue implementing and enabling more state-of-the-art ideas across different domains to provide users with ready-made AI solutions.

Acknowledgments

We would like to thank the following people for their feedback and guidance: Dibyajyoti Pati, Hans Krupakar, Min Zhang, Ravi Soni, Pál Tegzes, Levente Török, Dániel Szabó, Máté Fejes, Valentin Mikhaylenko, and Karley Yoder.

References

[1] Yoav Shoham, Raymond Perrault, Erik Brynjolfsson, Jack Clark, James Manyika, Juan Carlos Niebles, Terah Lyons, John Etchemendy, Barbara Grosz, and Zoe Bauer. The AI Index 2018 Annual Report. In *AI Index Steering Committee Human-Centered AI Initiative, Stanford University, Stanford, CA, December 2018*.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. `https://www.tensorflow.org`, 2015.

[3] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In Neural Information Processing Systems, Workshop on Machine Learning Systems, 2016.

[5] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*. Association for Computing Machinery, 2016.

[6] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[7] François Chollet et al. Keras. `https://keras.io`, 2015.

[8] Jeremy Howard et al. fastai. `https://github.com/fastai/fastai`, 2018.

[9] Piero Molino. Ludwig: a type-based declarative deep learning toolbox. *To appear*, 2019.

[10] L. McInnes, J. Healy, and J. Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *ArXiv e-prints*, 2018.

[11] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*.

[12] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic Attribution for Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*.

[13] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature Visualization. `https://distill.pub/2017/feature-visualization`, 2017.

[14] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced Deep Residual Networks for Single Image Super-Resolution. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*.

[15] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[16] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[17] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[18] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*.

[19] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.