Journal of Machine Learning Research 26 (2025) 1-18 Submitted 3/24; Revised 1/25; Published 1/25

depyf: Open the Opaque Box of PyTorch Compiler for Machine Learning Researchers

Kaichao You[†*] YOUKAICHAO@GMAIL.COM
Runsheng Bai[†] BRS21@MAILS.TSINGHUA.EDU.CN
Meng Cao[§] MENGCAO@APPLE.COM
Jianmin Wang[†] JIMWANG@TSINGHUA.EDU.CN
Ion Stoica[‡] ISTOICA@CS.BERKELEY.EDU
Mingsheng Long[††] MINGSHENG@TSINGHUA.EDU.CN
[†] School of Software, BNRist, Tsinghua University, Beijing 100084, China
[§] AIML, Apple [‡] Division of Computer Science, UC Berkeley, CA 94720-1776, USA

Editor: Sebastian Schelter

Abstract

PyTorch `2.x` introduces a compiler designed to accelerate deep learning programs. However, for machine learning researchers, fully leveraging the PyTorch compiler can be challenging due to its operation at the Python bytecode level, making it appear as an opaque box. To address this, we introduce `depyf`, a tool designed to demystify the inner workings of the PyTorch compiler. `depyf` decompiles the bytecode generated by PyTorch back into equivalent source code and establishes connections between the code objects in the memory and their counterparts in source code format on the disk. This feature enables users to step through the source code line by line using debuggers, thus enhancing their understanding of the underlying processes. Notably, `depyf` is non-intrusive and user-friendly, primarily relying on two convenient context managers for its core functionality. The project is openly available and is recognized as a PyTorch ecosystem project.

Keywords: PyTorch, Deep Learning Compiler, Decompilation

1. Introduction

Deep learning has profoundly impacted our daily lives, especially with the recent advancements in Large Language Models (LLMs) like ChatGPT (Schulman et al., 2022). These models demand considerable computational resources, prompting the swift development of specialized hardware (LeCun, 2019), such as GPUs (Markidis et al., 2018) and TPUs (Jouppi et al., 2020). However, fully leveraging the capabilities of this advanced hardware is challenging. It requires in-depth knowledge of hardware-specific programming, exemplified by technologies like FlashAttention (Dao et al., 2022). Such expertise often extends beyond the focus of machine learning researchers who concentrate on algorithm development. To bridge this gap, domain-specific deep learning compilers have been introduced (Li et al., 2020). These compilers are crafted to optimize deep learning programs for efficient operation on modern hardware. While these compilers simplify the optimization process, adapting

∗. This work is conducted during Kaichao You's internship at Apple.
†. Mingsheng Long is the corresponding author.

them to maximize benefits remains a complex endeavor. This complexity highlights the ongoing tension between hardware advancements and software optimization in the rapidly evolving field of deep learning.

PyTorch (Paszke et al., 2019), a widely-used deep learning framework among machine learning researchers, was traditionally imperative and user-friendly. To keep pace with recent hardware advancements and to enable better optimization for large-scale distributed training (Rasley et al., 2020; Shoeybi et al., 2020), PyTorch recently underwent a significant update, transitioning from PyTorch `1.x` to PyTorch `2.x`. This update included the `torch.compile` interface, integrating a deep learning compiler to better utilize modern hardware. While this bytecode-based approach is more robust than tracing-based compilers like JAX (Frostig et al., 2018), it is also more difficult to understand.

This paper first describes the challenges machine learning researchers face in understanding the PyTorch compiler, illustrated through a concrete example. It then discusses how the proposed tool addresses these challenges, concluding with practical usage examples and experimental results.

2. Challenges in Understanding the PyTorch Compiler

2.1 Dynamo: The Frontend of the PyTorch Compiler

The most complex component of the PyTorch compiler is its frontend named Dynamo. Dynamo's key functionality is to separate user code into distinct segments: pure Python code and pure PyTorch code, which forms the computation graph. Figure 1 (left) provides a detailed example of Dynamo's operation. This process involves three primary steps:

- Identifying the first operation that cannot be represented in the computation graph but requires the value of a previously computed tensor in the graph. Examples include operations like `print`ing a tensor's value or using a tensor's value to determine the control flow in Python `if` statements.
- Dividing preceding operations into two segments: a computation graph focused solely on tensor computations and Python code dedicated to manipulating Python objects.
- Handling the subsequent operations as one or more new functions (referred to as `resume functions`) and recursively reinitiating the analysis described above.

Dynamo functions at the Python bytecode level (see `LOAD`, `JUMP`, `CALL` instructions in Figure 1), which is a lower level than Python source code. It's important to note that **very few machine learning researchers are proficient in interpreting this bytecode**.

2.2 The Backend of the PyTorch Compiler

After the frontend extracts a computation graph, the backend optimizes this graph and ultimately generates binary executables suitable for CPU, GPU, and TPU hardware. A computation graph in Python is a *dynamically generated* function with tensor variables as nodes and tensor operations as edges, meaning it must be *executed in its entirety*. Consequently, users are **unable to employ debuggers for a step-by-step analysis of the function**. This becomes particularly challenging when the computation leads to a `NaN` (Not a Number) error, as it precludes the possibility of tracing through the code line by line to identify the operation responsible for the numeric issue.

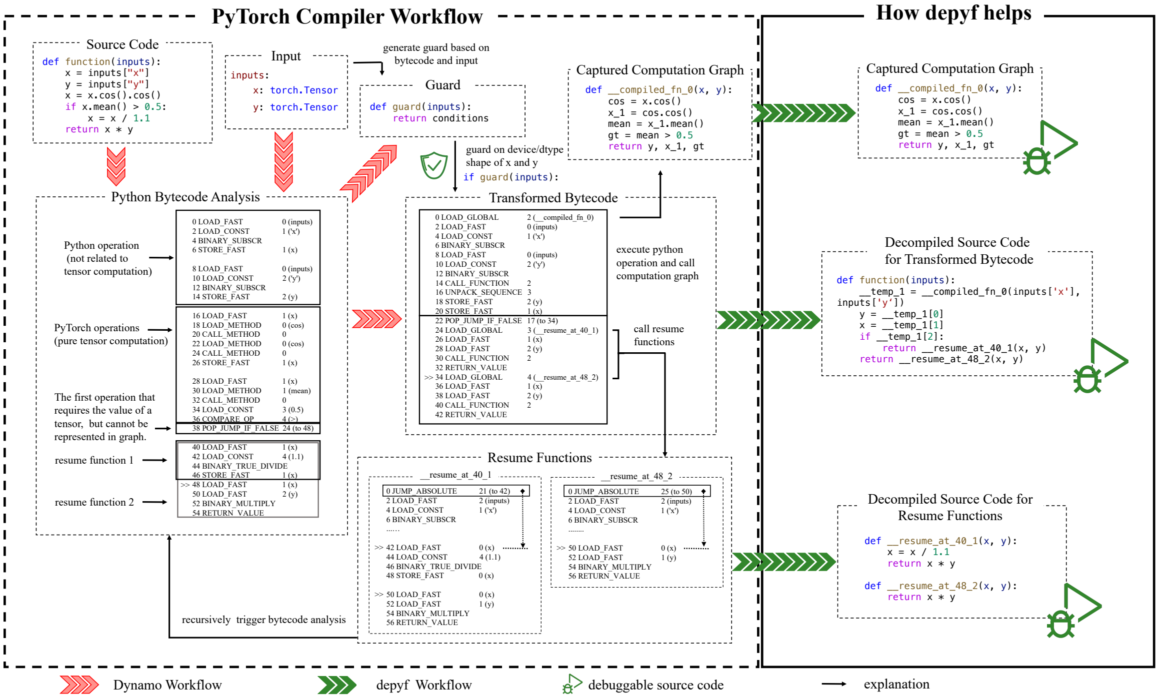

Figure 1: The workflow of the PyTorch compiler (left), and how `depyf` helps (right).

3. Solution

Bytecode Decompilation: The primary goal is to free machine learning researchers from the complexities of bytecode. The process of converting bytecode back into source code is called "decompilation". Before `depyf`, existing Python decompilers could transform Python bytecode into source code, but they have significant limitations:

- Designed for decompiling bytecode compiled from source code, they struggle with program-generated bytecode like that from PyTorch.
- They typically support only old versions of Python with limited compatibility.

To overcome these issues, we created a new Python bytecode decompiler[1] through symbolic execution of the bytecode. This approach requires handling only about two hundred types of Python bytecode, ensuring *compatibility with all Python versions supported by PyTorch*.

Moreover, the core component of the PyTorch compiler, written in C, is replicated in Python within `depyf` to elaborate on the underlying mechanisms for users.

Function Execution Hijacking: To facilitate line-by-line code execution with debuggers, the bytecode executed by Python must originate from an on-disk source code file. We utilize advanced Python features to intercept and replace critical function calls in PyTorch. This replacement involves dynamically generated functions with counterparts that include debugging information.

1. The name `depyf` stands for: decompile Python functions. We focus on function bytecodes, which is also the main focus of the PyTorch compiler.

Decompiler	Python 3.8	Python 3.9	Python 3.10	Python 3.11	PyTorch
decompyle3	90.6%(77/85)	✗	✗	✗	✗
uncompyle6	91.8%(78/85)	✗	✗	✗	✗
pycdc	74.1%(63/85)	74.1%(63/85)	74.1%(63/85)	67.1%(57/85)	19.3%(27/140)
depyf	100%(85/85)	100%(85/85)	100%(85/85)	100%(85/85)	100%(140/140)

Table 1: Correctness of decompilers in Python and PyTorch tests.

Usage: Using `depyf` is straightforward and non-intrusive. Users simply need to enclose their code within the context manager `with depyf.prepare_debug()`. This action enables `depyf` to capture all internal PyTorch details in that context, including decompiled source code and the computation graph. For those wishing to step through decompiled code with debuggers, an additional context manager, `with depyf.debug()`, is available. Appendix B provides more details about the usage.

Overview: Figure 1 provides an overview of `depyf`. More comprehensive details can be found in Appendix A. The advantages of `depyf` are threefold:

- It offers a Python implementation analogous to PyTorch's C implementation, aiding users in grasping the PyTorch compiler's logic. (See `full_code_xxx.py` in Figure 3)
- It includes a Python bytecode decompiler that transforms bytecode into equivalent source code, helping users understand the transformed bytecode from PyTorch. (See `__transformed_xxx.py` in Figure 3)
- It hijacks critical functions in PyTorch, enabling users to step through computation graph functions line by line using debuggers. (See `__compiled_xxx.py` in Figure 3)

4. Experiments

Table 1 presents the compatibility status of various existing decompilers with Python and PyTorch. Detailed descriptions of these tests can be found in Appendices C and D. Notably, `depyf` is *the only decompiler to successfully pass all the tests.* See Appendix E for an in-depth explanation of why they fail. Our testing approach is conducted in a continuous integration manner, whereby every new commit undergoes testing against the nightly version of PyTorch across all supported Python versions. This proactive strategy allows us to identify and resolve any compatibility issues before the release of new PyTorch versions. Furthermore, we engage in discussions with the PyTorch team to propose solutions that maintain this compatibility.

5. Conclusion

In this paper, we introduced `depyf`, a novel tool designed to open the opaque box of the PyTorch compiler, facilitating machine learning researchers' understanding and adaptation to `torch.compile`.

`depyf` is deployed in real-world projects like vLLM (Kwon et al., 2023) to help their `torch.compile` integration and received the PyTorch Innovator Award during the PyTorch Conference 2024. Its practical significance becomes more pronounced as the PyTorch compiler becomes more widely adopted.

Acknowledgments

We thank the PyTorch team for helpful discussions and guidance on many internal details. Special thanks for Jason Ansel, Horace He, and Edward Yang.

We thank many colleagues from the Machine Learning Group in Tsinghua University (THUML) for providing helpful feedback on the draft and documentation.

Kaichao You is partly supported by the Apple Scholar in AI/ML.

This work is supported by the National Natural Science Foundation of China (62021002 and U2342217) and the BNRist Innovation Fund (BNR2024RC01010).

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021.

Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers, 2022.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

Andrew Brock, Soham De, Samuel L. Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization, 2021.

Chun-Fu Chen, Quanfu Fan, and Rameswar Panda. Crossvit: Cross-attention multi-scale vision transformer for image classification, 2021a.

Yunpeng Chen, Jianan Li, Huaxin Xiao, Xiaojie Jin, Shuicheng Yan, and Jiashi Feng. Dual path networks, 2017.

Zhengsu Chen, Lingxi Xie, Jianwei Niu, Xuefeng Liu, Longhui Wei, and Qi Tian. Visformer: The vision-friendly transformer, 2021b.

Xiangxiang Chu, Zhi Tian, Yuqing Wang, Bo Zhang, Haibing Ren, Xiaolin Wei, Huaxia Xia, and Chunhua Shen. Twins: Revisiting the design of spatial attention in vision transformers, 2021.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020.

Will Constable, Xu Zhao, Victor Bittorf, Eric Christoffersen, Taylor Robie, Eric Han, Peng Wu, Nick Korovaiko, Jason Ansel, Orion Reblitz-Richardson, and Soumith Chintala. TorchBench: A collection of open source benchmarks for PyTorch performance and usability evaluation, September 2020. URL `https://github.com/pytorch/benchmark`.

Cheng Cui, Tingquan Gao, Shengyu Wei, Yuning Du, Ruoyu Guo, Shuilong Dong, Bin Lu, Ying Zhou, Xueying Lv, Qiwen Liu, Xiaoguang Hu, Dianhai Yu, and Yanjun Ma. Pp-lcnet: A lightweight cpu convolutional neural network, 2021.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS*, 2022.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*, 2019.

Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. Repvgg: Making vgg-style convnets great again, 2021.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR*, 2021a.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021b.

Stéphane d'Ascoli, Hugo Touvron, Matthew L Leavitt, Ari S Morcos, Giulio Biroli, and Levent Sagun. Convit: improving vision transformers with soft convolutional inductive biases*. *Journal of Statistical Mechanics: Theory and Experiment*, 2022(11):114005, November 2022. ISSN 1742-5468. doi: 10.1088/1742-5468/ac9830. URL `http://dx.doi.org/10.1088/1742-5468/ac9830`.

Alaaeldin El-Nouby, Hugo Touvron, Mathilde Caron, Piotr Bojanowski, Matthijs Douze, Armand Joulin, Ivan Laptev, Natalia Neverova, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jegou. Xcit: Cross-covariance image transformers, 2021.

Angela Fan, Shruti Bhosale, Holger Schwenk, Zhiyi Ma, Ahmed El-Kishky, Siddharth Goyal, Mandeep Baines, Onur Celebi, Guillaume Wenzek, Vishrav Chaudhary, Naman Goyal, Tom Birch, Vitaliy Liptchinsky, Sergey Edunov, Edouard Grave, Michael Auli, and Armand Joulin. Beyond english-centric multilingual machine translation, 2020.

Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *Systems for Machine Learning*, 2018.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

Shang-Hua Gao, Ming-Ming Cheng, Kai Zhao, Xin-Yu Zhang, Ming-Hsuan Yang, and Philip Torr. Res2net: A new multi-scale backbone architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(2):652–662, February 2021. ISSN 1939-3539. doi: 10.1109/tpami.2019.2938758. URL `http://dx.doi.org/10.1109/TPAMI.2019.2938758`.

Ben Graham, Alaaeldin El-Nouby, Hugo Touvron, Pierre Stock, Armand Joulin, Hervé Jégou, and Matthijs Douze. Levit: a vision transformer in convnet's clothing for faster inference, 2021.

Dongyoon Han, Sangdoo Yun, Byeongho Heo, and YoungJoon Yoo. Rethinking channel dimensions for efficient model design, 2021a.

Kai Han, Yunhe Wang, Qi Tian, Jianyuan Guo, Chunjing Xu, and Chang Xu. Ghostnet: More features from cheap operations, 2020a.

Kai Han, Yunhe Wang, Qiulin Zhang, Wei Zhang, Chunjing Xu, and Tong Zhang. Model rubik's cube: Twisting resolution, depth and width for tinynets, 2020b.

Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. Transformer in transformer, 2021b.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention, 2021.

Byeongho Heo, Sangdoo Yun, Dongyoon Han, Sanghyuk Chun, Junsuk Choe, and Seong Joon Oh. Rethinking spatial dimensions of vision transformers, 2021.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

Zhewei Huang, Wen Heng, and Shuchang Zhou. Learning to paint with model-based deep reinforcement learning, 2019.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

Huaizu Jiang, Deqing Sun, Varun Jampani, Ming-Hsuan Yang, Erik Learned-Miller, and Jan Kautz. Super slomo: High quality estimation of multiple intermediate frames for video interpolation. In *CVPR*, 2018.

Zihang Jiang, Weihao Yu, Daquan Zhou, Yunpeng Chen, Jiashi Feng, and Shuicheng Yan. Convbert: Improving bert with span-based dynamic convolution, 2021.

Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 2020.

Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollar, and Ross Girshick. Segment Anything. 2023.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 2017.

Oleksii Kuchaiev and Boris Ginsburg. Training deep autoencoders for collaborative filtering, 2017.

Alexey Kurakin, Ian Goodfellow, Samy Bengio, Yinpeng Dong, Fangzhou Liao, Ming Liang, Tianyu Pang, Jun Zhu, Xiaolin Hu, Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, Alan Yuille, Sangxia Huang, Yao Zhao, Yuzhe Zhao, Zhonglin Han, Junjiajia Long, Yerkebulan Berdibekov, Takuya Akiba, Seiya Tokui, and Motoki Abe. Adversarial attacks and defences competition, 2018.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations, 2020.

Yann LeCun. Deep Learning Hardware: Past, Present, and Future. In *ISSCC*, 2019.

Youngwan Lee and Jongyoul Park. Centermask : Real-time anchor-free instance segmentation, 2020.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019.

Minghao Li, Tengchao Lv, Jingye Chen, Lei Cui, Yijuan Lu, Dinei Florencio, Cha Zhang, Zhoujun Li, and Furu Wei. Trocr: Transformer-based optical character recognition with pre-trained models, 2022.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

Ming Lin, Hesen Chen, Xiuyu Sun, Qi Qian, Hao Li, and Rong Jin. Neural architecture design for gpu-efficient networks, 2020a.

Qian Lin and Hwee Tou Ng. A semi-supervised learning approach with two teachers to improve breakdown identification in dialogues, 2022.

Shanchuan Lin, Andrey Ryabtsev, Soumyadip Sengupta, Brian Curless, Steve Seitz, and Ira Kemelmacher-Shlizerman. Real-time high-resolution background matting, 2020b.

Xi Victoria Lin, Todor Mihaylov, Mikel Artetxe, Tianlu Wang, Shuohui Chen, Daniel Simig, Myle Ott, Naman Goyal, Shruti Bhosale, Jingfei Du, Ramakanth Pasunuru, Sam Shleifer, Punit Singh Koura, Vishrav Chaudhary, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Zornitsa Kozareva, Mona Diab, Veselin Stoyanov, and Xian Li. Few-shot learning with multilingual language models, 2022.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search, 2018.

Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. Pay attention to mlps, 2021a.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual denoising pre-training for neural machine translation, 2020.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, 2021b.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s, 2022.

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, 2018.

Louis Martin, Benjamin Muller, Pedro Javier Ortiz Suárez, Yoann Dupont, Laurent Romary, Éric de la Clergerie, Djamé Seddah, and Benoît Sagot. Camembert: a tasty french language model. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.645. URL `http://dx.doi.org/10.18653/v1/2020.acl-main.645`.

Dushyant Mehta, Oleksandr Sotnychenko, Franziska Mueller, Weipeng Xu, Mohamed Elgharib, Pascal Fua, Hans-Peter Seidel, Helge Rhodin, Gerard Pons-Moll, and Christian Theobalt. Xnect: real-time multi-person 3d motion capture with a single rgb camera. *ACM Transactions on Graphics*, 39(4), August 2020. ISSN 1557-7368. doi: 10.1145/3386569.3392410. URL `http://dx.doi.org/10.1145/3386569.3392410`.

Sachin Mehta and Mohammad Rastegari. Mobilevit: Light-weight, general-purpose, and mobile-friendly vision transformer, 2022.

Dianwen Ng, Yunqi Chen, Biao Tian, Qiang Fu, and Eng Siong Chng. Convmixer: Feature interactive convolution with curriculum learning for small footprint and noisy far-field keyword spotting. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, May 2022. doi: 10.1109/icassp43922.2022.9747025. URL `http://dx.doi.org/10.1109/ICASSP43922.2022.9747025`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022.

Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces, 2020.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *KDD*, 2020.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, and Michael Pokorny. ChatGPT: Optimizing language models for dialogue. *OpenAI blog*, 2022.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2020.

Kurt Shuster, Jing Xu, Mojtaba Komeili, Da Ju, Eric Michael Smith, Stephen Roller, Megan Ung, Moya Chen, Kushal Arora, Joshua Lane, Morteza Behrooz, William Ngan, Spencer Poff, Naman Goyal, Arthur Szlam, Y-Lan Boureau, Melanie Kambadur, and Jason Weston. Blenderbot 3: a deployed conversational agent that continually learns to responsibly engage, 2022.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

Aravind Srinivas, Tsung-Yi Lin, Niki Parmar, Jonathon Shlens, Pieter Abbeel, and Ashish Vaswani. Bottleneck transformers for visual recognition, 2021.

Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Designing hardware-efficient convnets in less than 4 hours, 2019.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices, 2020.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

Mingxing Tan and Quoc V. Le. Mixconv: Mixed depthwise convolutional kernels, 2019.

Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile, 2019.

Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. Mlp-mixer: An all-mlp architecture for vision, 2021.

Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Gautier Izacard, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. Resmlp: Feedforward networks for image classification with data-efficient training, 2021a.

Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention, 2021b.

Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers, 2021c.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023.

Ashish Vaswani, Prajit Ramachandran, Aravind Srinivas, Niki Parmar, Blake Hechtman, and Jonathon Shlens. Scaling local self-attention for parameter efficient visual backbones, 2021.

Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, and Jun-Wei Hsieh. Cspnet: A new backbone that can enhance learning capability of cnn, 2019.

Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. Deep high-resolution representation learning for visual recognition, 2020.

Ross Wightman. PyTorch Image Models, 2023.

Ross Wightman, Hugo Touvron, and Hervé Jégou. Resnet strikes back: An improved training procedure in timm, 2021.

Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, and Sam Shleifer. Transformers: State-of-the-art natural language processing. In *EMNLP*, 2020.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search, 2019.

Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V. Le. Self-training with noisy student improves imagenet classification, 2020.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks, 2017.

Weijian Xu, Yifan Xu, Tyler Chang, and Zhuowen Tu. Co-scale conv-attentional image transformers, 2021.

Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. Layoutlm: Pre-training of text and layout for document image understanding. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20. ACM, August 2020. doi: 10.1145/3394486.3403172. URL http://dx.doi.org/10.1145/3394486.3403172.

Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mt5: A massively multilingual pre-trained text-to-text transformer, 2021.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding, 2020.

Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. Deep layer aggregation, 2019.

Weihao Yu, Mi Luo, Pan Zhou, Chenyang Si, Yichen Zhou, Xinchao Wang, Jiashi Feng, and Shuicheng Yan. Metaformer is actually what you need for vision, 2022.

Li Yuan, Qibin Hou, Zihang Jiang, Jiashi Feng, and Shuicheng Yan. Volo: Vision outlooker for visual recognition, 2021.

Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Haibin Lin, Zhi Zhang, Yue Sun, Tong He, Jonas Mueller, R. Manmatha, Mu Li, and Alexander Smola. Resnest: Split-attention networks, 2020a.

Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization, 2020b.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices, 2017.

Zizhao Zhang, Han Zhang, Long Zhao, Ting Chen, Sercan O. Arik, and Tomas Pfister. Nested hierarchical transformer: Towards accurate, data-efficient and interpretable visual understanding, 2021.

Appendix A. System Architecture Overview

Figure 2 shows the overall architecture when we use `torch.compile` with `depyf`.

- Normally, when we execute Python code, the code is compiled into Python bytecode, which is then executed by the Python interpreter.
- When `torch.compile` is used, PyTorch will compile the function into a new bytecode object to execute. It achieves this via registering a frame-evaluation function to the Python interpreter. The frame-evaluation function will be called whenever the function is executed. PyTorch wraps the frame-evaluation callback registration via the `torch._C._dynamo.eval_frame.set_eval_frame` function. Because PyTorch directly generates the bytecode, it does not have the source code information. The bytecode is directly executed by the Python interpreter.
- When `depyf` is used together with PyTorch, it will register a bytecode hook to PyTorch via `torch._dynamo.convert_frame.register_bytecode_hook` (we work together with the PyTorch team to design this bytecode hook mechanism). The hook will be called whenever PyTorch compiles a function. The hook will decompile the bytecode into source code and dump the source code to disk. The source code is then compiled into a new bytecode object, which is functionally equivalent to the bytecode generated by PyTorch, but with source code information. PyTorch will use the new bytecode object to execute the function. The part related with `depyf` is marked as green.

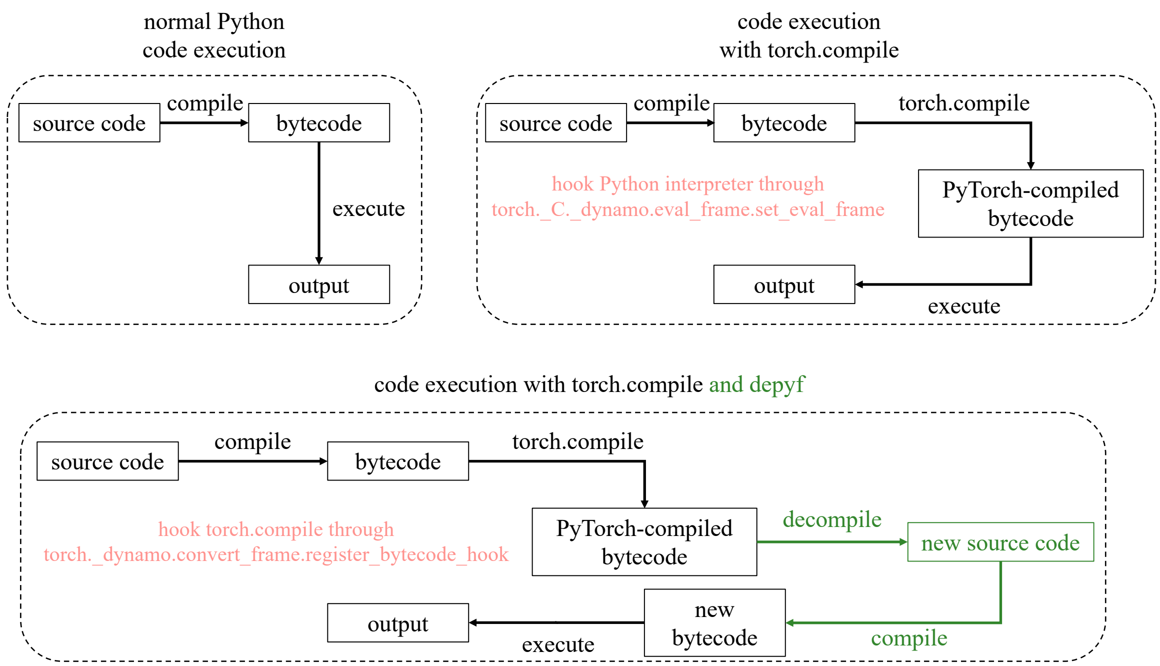

Figure 2: System architecture overview when using `torch.compile` with `depyf`.

For more details, please refer to the documentation page [2], which also includes the source code structure and more explanations about the decompiler.

Appendix B. Usage

We provide two convenient context managers for users: `with depyf.prepare_debug()` and `with depyf.debug()`. The first one will capture all the calls to functions using `torch.compile`, and dump many internal details in a directory specified by users (*i.e.*, the argument of `with depyf.prepare_debug()`). The second one will pause the program for users to set breakpoints in the dumped source code, and any call to functions related with `torch.compile` can be stepped through line by line using standard Python debuggers.

There are three types of source code dumped by `depyf`: computation graphs (prefixed by `__compiled`), decompiled source code (prefixed by `__transformed`), and descriptive source code (prefixed by `full_code`).

There is also a detailed use case in the documentation, to show how to use `depyf` to understand and optimize the performance of PyTorch code after compilation. The optimization is hard to achieve without understanding the PyTorch compiler, but it becomes a lot easier after `depyf` decompiles the bytecode to source code and reveals internal details of the PyTorch compiler. Note that this is a real-world example and goes into production with the latest vLLM (Kwon et al., 2023) project.

2. `https://depyf.readthedocs.io/en/latest/dev_doc.html`

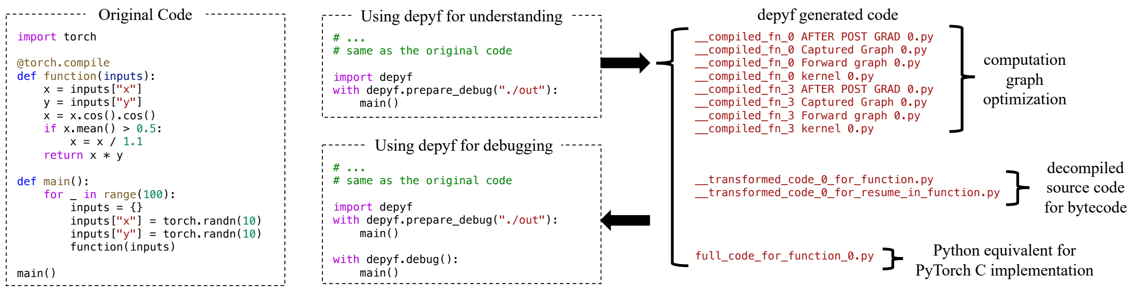

Figure 3: Two usage of `depyf`.

Appendix C. Tested PyTorch Models

This section lists all of the PyTorch models we test in Table 1. These models come from three suites of deep learning models: TorchBench (Constable et al., 2020) collects models from famous (highly cited projects as ranked by `https://paperswithcode.com/`) machine learning repositories like Segment Anything (Kirillov et al., 2023) and SuperSloMo (Jiang et al., 2018); Huggingface Transformers (Wolf et al., 2020) is the most popular library for transformers models including LLaMA (Touvron et al., 2023) and BERT (Devlin et al., 2019); TIMM (Wightman, 2023) is the most popular library for computer vision models including ResNet (He et al., 2016) and ViT (Dosovitskiy et al., 2021a).

To be specific, the models include:

- `BertForMaskedLM`, `BertForQuestionAnswering`, `BERT_pytorch`, `hf_Bert` (Devlin et al., 2019)
- `AlbertForMaskedLM`, `AlbertForQuestionAnswering` (Lan et al., 2020)
- `AllenaiLongformerBase` (Beltagy et al., 2020)
- `BartForCausalLM`, `BartForConditionalGeneration`, `hf_Bart` (Lewis et al., 2019)
- `BlenderbotForCausalLM`, `BlenderbotForConditionalGeneration`, `BlenderbotSmallForCausalLM`, `BlenderbotSmallForConditionalGeneration` (Shuster et al., 2022)
- `CamemBart` (Martin et al., 2020)
- `DebertaForMaskedLM`, `DebertaForQuestionAnswering`, `DebertaV2ForMaskedLM`, `DebertaV2ForQuestionAnswering` (He et al., 2021)
- `DistilBertForMaskedLM`, `DistilBertForQuestionAnswering` (Sanh et al., 2020)
- `DistilGPT` (Radford et al., 2019; Sanh et al., 2020)
- `ElectraForCausalLM`, `ElectraForQuestionAnswering` (Clark et al., 2020)
- `GPT2ForSequenceClassification`, `hf_GPT2` (Radford et al., 2019)
- `GPTJForCausalLM`, `GPTJForQuestionAnswering` (Radford et al., 2022)
- `GPTNeoForCausalLM`, `GPTNeoForSequenceClassification` (Gao et al., 2020)
- `LayoutLMForMaskedLM`, `LayoutLMForSequenceClassification` (Xu et al., 2020)
- `M2M100` (Fan et al., 2020)
- `MBartForCausalLM`, `MBartForSequenceClassification` (Liu et al., 2020)
- `MT5ForConditionalGeneration` (Xue et al., 2021)
- `MegatronBertForMaskedLM`, `MegatronBertForQuestionAnswering` (Fan et al., 2020)
- `MobileBertForMaskedLM`, `MobileBertForQuestionAnswering` (Sun et al., 2020)

- `OPTForCausalLM` (Zhang et al., 2022)
- `PLBartForCausalLM, PLBartForConditionalGeneration` (Ahmad et al., 2021)
- `PegasusForCausalLM, PegasusFOrConditionalGeneration` (Zhang et al., 2020b)
- `RoBERTaForCausalLM, RoBERTaForQuestionAnswering` (Liu et al., 2019)
- S2T2 (Lin and Ng, 2022)
- `T5ForConditionalGeneration, T5Small, hf_T5` (Raffel et al., 2023)
- `TrOCRForCausalLM` (Li et al., 2022)
- `XGLMForCausalLM` (Lin et al., 2022)
- `XLNetLMHeadModel` (Yang et al., 2020)
- `YituTechConvBert` (Jiang et al., 2021)
- `gluon_inception_v3, inception_v3` (Szegedy et al., 2015)
- `adv_inception_v3` (Kurakin et al., 2018)
- `beit_base_patch16_224` (Bao et al., 2022)
- `botnet26t_256` (Srinivas et al., 2021)
- `eca_botnext26ts_256, sebotnet33ts_256` (Srinivas et al., 2021; Wightman et al., 2021)
- `cait_m36_384` (Touvron et al., 2021c)
- `coat_lite_mini` (Xu et al., 2021)
- `convit_base` (d'Ascoli et al., 2022)
- `convmixer_768_32` (Ng et al., 2022)
- `convnext_base` (Liu et al., 2022)
- `crossvit_9_240` (Chen et al., 2021a)
- `cspdarknet53` (Wang et al., 2019; Bochkovskiy et al., 2020)
- `deit_base_distilled_patch16_224` (Touvron et al., 2021b)
- `dla102` (Yu et al., 2019)
- `dm_nfnet_f0, nfnet_l0, timm_nfnet` (Brock et al., 2021):
- `dpn107` (Chen et al., 2017)
- `eca_halonext26ts` (Vaswani et al., 2021; Wightman et al., 2021)
- `ese_vovnet19b_dw, timm_vovnet` (Lee and Park, 2020)
- `fbnetc_100, fbnetv3_b` (Wu et al., 2019)
- `gernet_l` (Lin et al., 2020a)
- `ghostnet_100` (Han et al., 2020a)
- `mixer_b16_224, gmixer_24_224` (Tolstikhin et al., 2021)
- `gmlp_s16_224` (Liu et al., 2021a)
- `hrnet_w18` (Wang et al., 2020)
- `jx_nest_base` (Zhang et al., 2021)
- `lcnet_050` (Cui et al., 2021)
- `levit_128` (Graham et al., 2021)
- `mixnet_l, tf_mixnet_l` (Tan and Le, 2019)
- `mnasnet_100, mnasnet1_0` (Tan et al., 2019)
- `mobilenetv2_100, mobilenet_v2` (Sandler et al., 2019; Wightman et al., 2021)
- `mobilenetv3_large_100, mobilenet_v3_large` (Howard et al., 2019)
- `mobilevit_s` (Mehta and Rastegari, 2022)
- `pit_b_224` (Heo et al., 2021)
- `pnasnet5large` (Liu et al., 2018)
- `poolformer_m36` (Yu et al., 2022)
- `regnety_002, timm_regnet` (Radosavovic et al., 2020)

- `repvgg_a2` (Ding et al., 2021)
- `res2net101_26w_4s, res2net50_14w_8s, res2next50, resnet18, resnet50` (Gao et al., 2021)
- `resmlp_12_224` (Touvron et al., 2021a)
- `resnest101e, timm_resnest` (Zhang et al., 2020a)
- `rexnet_100` (Han et al., 2021a)
- `selecsls42b` (Mehta et al., 2020)
- `spnasnet_100` (Stamoulis et al., 2019)
- `swin_base_patch4_window7_224` (Liu et al., 2021b)
- `swsl_resnext101_32x16d, resnext50_32x4d` (Xie et al., 2017)
- `tf_efficientnet_b0, timm_efficientnet` (Tan and Le, 2020; Xie et al., 2020)
- `tinynet_a` (Han et al., 2020b)
- `tnt_s_patch16_224` (Han et al., 2021b)
- `twins_pcpvt_base` (Chu et al., 2021)
- `visformer_small` (Chen et al., 2021b)
- `vit_base_patch16_224, timm_vision_transformer` (Dosovitskiy et al., 2021b)
- `volo_d1_224` (Yuan et al., 2021)
- `xcit_large_24_p8_224` (El-Nouby et al., 2021)
- `Background_Matting` (Lin et al., 2020b)
- `LearningToPaint` (Huang et al., 2019)
- `alexnet` (Krizhevsky et al., 2017)
- `dcgan` (Radford et al., 2016)
- `densenet121` (Huang et al., 2018)
- `nvidia_deeprecommender` (Kuchaiev and Ginsburg, 2017)
- `pytorch_unet` (Ronneberger et al., 2015)
- `shufflenet_v2_x1_0` (Zhang et al., 2017)
- `squeezenet1_1` (Iandola et al., 2016)
- `vgg16` (Simonyan and Zisserman, 2015)

Appendix D. Tested Python Syntax

We also collect commonly used Python features in the above models, and store them in a simple Python test with over 80 testcases in `https://github.com/thuml/depyf/blob/master/tests/test.py`.

Appendix E. Why Existing Decompilers Fail

The three existing decompilers are designed to work with bytecode compiled from source code, while `depyf` is designed to work with bytecode generated by the PyTorch compiler. Bytecode compiled from source code has clear patterns and structures, which they take advantage of to decompile. For example, `decompyle3` uses the common bytecode patterns for generator comprehensions to decompile them. However, bytecode generated by the PyTorch compiler is more complex and harder to decompile. It can deviate from the common patterns and structures in an arbitrary way, which makes it hard for existing decompilers to work with. For example, PyTorch can pack non-constant objects inside the bytecode's `co_consts` field, which will fail all three existing decompilers. `depyf` is not only a general-purpose decompiler but is also equipped with extended features to handle irregularities in PyTorch bytecode, which makes it the only tool that can decompile PyTorch bytecode.

Appendix F. Collected Output

We collect all the output from PyTorch in `https://github.com/thuml/learn_torch.compile`. It includes many commonly used models, how PyTorch converts them, and what is the shape of tensors across training and inference. All details are in self-contained scripts.