Seminar Report

What's new with PyTorch?

Constantin Dalinghaus

MatrNr: 21969745

Supervisor: Chirag Mandal

Georg-August-Universität Göttingen
Institute of Computer Science

September 20, 2024

Abstract

The efficient training of deep learning models is critical, especially for researchers and practitioners with limited computational resources. Optimizing model training performance remains a significant area of focus in machine learning research. While PyTorch 2.0 introduced the torch.compile() function, promising substantial performance improvements through Just-In-Time (JIT) compilation, independent verification of these claims is limited. This study aims to assess whether torch.compile() delivers on its performance promises in academic and business contexts.

Existing compiler optimization tools like Glow, TensorFlow XLA, and Apache TVM offer performance enhancements but may not integrate seamlessly with dynamic computational graphs or require significant code modifications. The limitations of these tools highlight the need for solutions that provide performance gains without sacrificing usability. We conducted an independent evaluation of PyTorch 2.0's torch.compile() by benchmarking training performance on Convolutional Neural Networks and Transformer-based models, comparing compiled and non-compiled executions across different model sizes and hardware configurations.

Our results demonstrate that model compilation via torch.compile() generally leads to significant reductions in execution time and improved hardware utilization, particularly for larger models. However, for smaller models like ResNet18, the performance gains were less consistent, suggesting that the efficacy of compilation may vary based on model complexity. These findings confirm the performance claims of PyTorch 2.0 and underscore the importance of considering model characteristics when employing compilation strategies.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☑ As a tool in the toolbox of a modern developer, LLMs have been extensively used throughout the entire process of this work, from information retrieval to writing code as well as rephrasing and optimizing text for clarity. This makes it hard to explicitly declare specific usages. The intention of using these models is not to avoid doing actual work or undermine the scientific method, but rather to optimize the process to arrive at better results, faster.

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC	High-Performance Computing
CNN	Convolutional neural network
LLM	Large language model
timm	Pytorch Image Models
MFU	Mean Flop Utilization
DSL	Domain Specific Language
ONNX	Open Neural Network Exchange

1 Introduction

The efficient training of Deep Learning models remains a critical area of research, particularly for independent academic researchers who often conduct multiple training experiments to examine the impact of hyperparameters. Unlike large-scale foundational models, which are commonly pretrained on extensive GPU clusters, academic researchers typically operate under resource constraints and must optimize their workflows accordingly. As a result, improving the efficiency of model training on shared computational infrastructures, such as university high-performance computing (HPC) systems, is of significant importance. Despite recent advancements, inefficiencies in the utilization of computational resources during model training persist, underscoring the need for techniques that maximize performance, even on smaller-scale infrastructures.

In 2023, the PyTorch foundation introduced PyTorch 2.0, which includes a significant overhaul to the backend of PyTorch, most notably the Just In Time (JIT) compilation functionality enabled by the torch.compile() function. The PyTorch foundation published alongside benchmarks suggesting considerable performance improvements when compared to previous versions of the framework.

Independent verification of these claims, especially with respect to reproducibility and applicability in academic contexts, remains scarce. This study aims to address this gap in knowledge by investigating the following research questions:

- RQ1: Can the performance claims made by the PyTorch foundation regarding the torch.compile() functionality in PyTorch 2.0 be independently verified? Specifically, does it typically (1) work out of the box with no additional overhead, (2) yield significant performance gains, and (3) result in higher GPU utilization?
- RQ2: Are the performance improvements offered by PyTorch 2.0 significant enough to meet the requirements of a typical business use case?

2 Background

2.1 PyTorch

PyTorch is a deep learning framework developed by Meta AI, designed primarily for use with Python. Derived from the Torch framework, which was originally implemented in Lua, PyTorch was first released in September 2016. It provides a flexible and intuitive platform for constructing and training neural networks and has rapidly become a prominent tool in the machine learning community.

A key aspect of PyTorch's design philosophy is the prioritization of ease of use, which is reflected in its core design principles [Foub]:

1. Principle: Usability over Performance
2. Principle: Simple Over Easy
3. Principle: Python First with Best In Class Language Interoperability

The clear decision to prefer usability over performance is unusual for a deep learning framework. Many industry specialists have argued that it is this focus on user experience that makes PyTorch the dominant player in the deep learning space [Xco17].

2.2 PyTorch 2.0

PyTorch 2.0, released in March 2023, represents a significant upgrade to the PyTorch framework. This release includes substantial modifications to the underlying architecture, most notably the introduction of `torch.compile`, a mechanism primarilly designed to support ahead-of-time (AOT) compilation. The introduction of `torch.compile` aims to enhance computational efficiency by optimizing the execution of neural networks during runtime.

A primary objective of this release was to maintain backward compatibility, ensuring that the core PyTorch interface remains consistent with previous versions. This approach was taken to minimize disruptions for existing users, thereby facilitating a smooth transition to the new version while leveraging the performance improvements offered by the updated backend[Foua] 1.

2.2.1 Torch Dynamo

Torch Dynamo is a Python-level Just-In-Time (JIT) compiler introduced in PyTorch 2.0. Its primary function is to transform user-authored model code into an FX (Function Transformation) graph representation. This transformation enables subsequent stages of the compilation process to optimize and execute the model more efficiently. Torch Dynamo serves as the initial stage of the compilation pipeline, converting high-level model definitions into a format suitable for further processing by the PyTorch backend.

2.2.2 AOT Autograd

Ahead-of-Time (AOT) Autograd is a novel approach to the compilation of computational graphs within PyTorch. In contrast to previous methods that required the computational

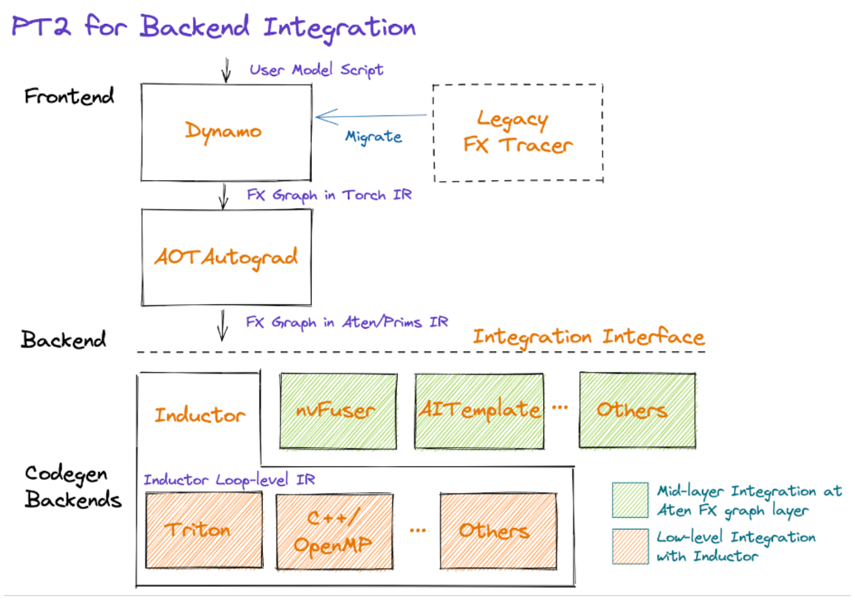

Figure 1: Overview over the PyTorch 2.0 backend architecture

graph to be compiled at each execution, AOT Autograd enables the graph to be compiled once and reused across multiple executions. This approach not only reduces the overhead associated with repeated graph compilation but also allows for advanced optimizations to be applied to the graph prior to execution, enhancing overall computational efficiency. Additionally, AOT Autograd also performs operator lowering.

2.2.3 Torch Inductor

Torch Inductor is the new compilation engine introduced in PyTorch 2.0. It operates by taking the lowered FX graph generated by AOT Autograd and further compiling it into hardware-specific kernel code. For GPUs, particularly those built by Nvidia and AMD, Torch Inductor generates code in the programming language Triton, which is a Domain Specific Language (DSL) for GPU programming. This component of the PyTorch 2.0 backend is crucial for achieving high-performance execution across diverse hardware architectures [TKC19].

2.2.4 PrimTorch

PrimTorch addresses the complexity of PyTorch's extensive operator set by providing a canonicalization mechanism. It reduces the large set of operators available in PyTorch to a smaller subset from which all operations can be constructed. The primary motivation behind PrimTorch is to simplify the development of custom backends, making the PyTorch ecosystem more modular and extensible. This approach also enhances the consistency and maintainability of the framework by standardizing the building blocks used across different components.

2.3 Neural Network Architectures

There exist a large variety of architectores for building Neural Networks. All of these have specific hardware requirements under which they perform most efficiently. We'll briefly review the architectures relevant for this project.

2.3.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) were initially proposed by Yann LeCun and colleagues at Bell Labs in 1989 [LeC+89]. These networks introduced a structured approach to integrating convolutional filters within neural networks, with the parameters of these filters being optimized through backpropagation.

CNNs inherently incorporate (among other inductive biases) the inductive bias of locality, which assumes that local features are more pertinent than global features. This bias is particularly advantageous when the available training data is limited. However, when larger datasets are accessible, models without such inductive biases tend to be more computationally efficient during training [Dos+21].

2.3.2 Transformer Models

Transformer Neural Networks are used for most modern large scale deep learning applications. Transformer Neural Networks offer a computational architecture that is very versatile in the functions it can learn, while being very efficient to train on the hardware that's currently available. Transformer models underly Large Language Models, current SOTA Image classification solutions etc. In this work, Transformer Networks are used only in their Large Language Model form, the rationale being that outside of input and output layers, most of the core compute graph is identical among various implementations of the transformer architectures.

3 Related Research

3.1 Compiler Optimization in Deep Learning

Compiler optimization plays a crucial role in enhancing the performance of deep learning models by efficiently translating high-level neural network definitions into hardware-specific instructions. Various compilers and frameworks have been developed to address the computational challenges in deep learning, with the goal of maximizing hardware utilization and minimizing execution time.

3.1.1 Glow Compiler

Glow, an earlier attempt at creating a deep learning compiler, focuses on optimizing neural network execution through operator lowering. Operator lowering reduces the complexity of backend implementation by minimizing the number of operators required. This makes it easier to optimize the performance across different hardware architectures. Glow's approach is conceptually similar to PyTorch 2.0's `torch.compile` feature, as both aim to streamline the execution pipeline and improve computational efficiency. However, Glow's emphasis on operator lowering specifically addresses the challenge of backend diversity, ensuring that a wide range of hardware can be effectively utilized with a simplified set of operators [Rot+18]. PyTorch 2.0 performs operator lowering during the AOT Autograd stage.

3.1.2 TensorFlow XLA (Accelerated Linear Algebra)

TensorFlow XLA (Accelerated Linear Algebra) is an independent compiler designed to optimize the execution of machine learning models. Unlike PyTorch 2.0, which was developed primarily for dynamic computational graphs under a "define-by-run" paradigm, XLA was originally built to optimize static computational graphs. XLA operates by transforming high-level operations into highly optimized low-level code, targeting specific hardware such as CPUs, GPUs, and TPUs. This approach significantly improves execution efficiency, particularly for models that do not require dynamic graph definition. With the introduction of OpenXLA, a more framework-agnostic fork of pure XLA, XLA now supports multiple deep learning frameworks, including TensorFlow, JAX, and PyTorch, enabling seamless integration and optimization across different platforms [teab].

3.1.3 Apache TVM

Apache TVM is an open-source deep learning compiler stack that provides end-to-end optimization for machine learning models across diverse hardware backends. Unlike PyTorch 2.0, TVM's primary focus is optimizing performance during inference. TVM facilitates the automatic generation of optimized code for various hardware targets, including CPUs, GPUs, and specialized accelerators. It employs techniques such as tensorization, operator fusion, and auto-tuning to maximize performance. TVM's modular design also allows developers to extend and customize the compiler for specific use cases, making it highly adaptable to new hardware and emerging technologies [].

3.1.4 ONNX (Open Neural Network Exchange)

ONNX (Open Neural Network Exchange) is an open-source format designed to represent deep learning models in a platform-independent manner, allowing them to be transferred between different frameworks such as PyTorch, TensorFlow, and MXNet. ONNX serves as a bridge between different deep learning environments, enabling models trained in one framework to be deployed in another without requiring extensive reconfiguration. Additionally, ONNX provides a standardized interface for applying various compiler optimizations, ensuring that models are executed efficiently regardless of the underlying hardware [teaa].

4 Methodology

To answer the research questions, we benchmark model training for Convolutional Neural Networks and Transformer Neural Networks. CNN benchmarking was performed by training an image classification model, Transformer benchmarking was done by training a Large Language Model. For both architectures, we train small and medium scale models to observe any possible impact of model size on training performance. We were unable to train a large variant of each models due to hardware constraints.

Please note that for benchmarking LLMs and CNNs, different hardware configurations were used, both of which are documented in the following paragraphs.

4.1 CNN Benchmarking

For CNN benchmarking, we perform one epoch of training a Resnet34 and Resnet18 model [He+16] on the full ILSVRC train dataset [Rus+15]. We evaluate three different szenarios:

1. No model compilation
2. Torch inductor model compilation
3. Torchscript model tracing

We use the train.py script in an unmodified version of Pytorch Image Models (timm) [Wig]. CNN benchmarking was run on a single Nvidia H100 GPU, 12 cores of Intel Xeon Platinum 8468 processors as well as 64GB of memory. For reproduction, see the full slurm config used in appendix 4.

4.2 LLM benchmarking

For LLM benchmarking we use a modified version of nanoGPT, an LLM training framework by Andrew Karpathy [Kar] which is a pure minimal PyTorch implementation of Large Language Models. To enable comparison of different compilation strategies, we made modifications to the code which are documented in appendix 3.

LLM Benchmarking was performed on a single compute node using one Nvidia A100 80GB GPU, 6 cores of AMD Zen3 EPYC 7513 and using 32GB of memory. For reproduction, see the full slurm config used in appendix 1.

Please note that slurm partition names etc. have to be adapted to fit the compute environment.

5 Results

5.1 Results for LLM benchmarking

Table 1: Execution time in ms, per iteration

	μ_{gpt2}	σ_{gpt2}	$\mu_{shakespeare}$	$\sigma_{shakespeare}$
Eager execution	3107.334	2.762	16.029	2.666
Compiled (Cudagraphs backend)	3098.297	0.956	16.828	3.935
Compiled (Inductor backend)	**2640.227**	5.773	**12.191**	1.556

Table 2: Mean Flop Utilization (mfu) in percent

	μ_{gpt2}	σ_{gpt2}	$\mu_{shakespeare}$	$\sigma_{shakespeare}$
Eager execution	43.337	0.000	22.951	0.853
Compiled (Cudagraphs backend)	43.463	0.000	21.916	0.653
Compiled (Inductor backend)	**51.023**	0.004	**29.920**	0.952

Our experimental data shows a drop in mean execution time for our gpt2 training, as well as for our shakespeare model. Notably, while the mean execution time decreases, the variance seems to increase on the gpt2 training, while it actually decreases on the shakespeare training (Table 1).

The mean flop utilization also showed an increase for the compiled versions of both gpt2 and shakespeare models. In both gpt2 and shakespeare models, the variance of the MFU increased when the model was compiled. What's further noteworthy is that the variance of the variance is multiple orders of magnitude lower for the gpt2 model than it is for the smaller shakespeare model (Table 2).

5.2 Results for CNN benchmarking

For CNN benchmarking, we only measure mean execution time per training step.

Table 3: Execution time in seconds, per iteration

	$\mu_{resnet34}$	$\sigma_{resnet34}$	$\mu_{resnet18}$	$\sigma_{resnet18}$
Eager execution	0.243	0.101	0.299	0.132
Compiled (Using torchscript)	0.229	0.097	**0.144**	0.049
Compiled (Inductor backend)	**0.205**	0.099	0.403	0.161

Our experimental data shows a negative correlation between level of compilation and execution time per seconds for training the resnet34. Notably, this compilation is not observed when training the smaller resnet18. The standard deviation of execution time per iteration appears steady for the resnet34 variant. The standard deviation of execution time per iteration for the resnet18 model is measured to be more variable.

6 Discussion

6.1 Impact of Compilation on Execution Time

Our experimental data supports the hypothesis that model compilation generally improves execution time. This was particularly evident for the GPT-2 model and the Shakespeare model, where compiled versions (using the Inductor backend) consistently showed reduced execution time compared to the eager execution mode. Notably, the Inductor backend achieved a 15% decrease in execution time for GPT-2 and a 24% decrease for the Shakespeare model (Table 1). This indicates that, at least for LLMs, compilation via the Inductor backend offers significant performance gains.

In contrast, the CNN benchmarking results reveal a more complex relationship between model compilation and execution time. For the ResNet34 model, execution time per iteration decreased with increasing levels of compilation, which aligns with the expected behavior. However, for the smaller ResNet18 model, the opposite trend was observed, with compiled execution times being slightly higher than those in the eager execution mode (Table 3). This discrepancy suggests that model size and complexity may influence the efficacy of compilation optimizations, especially for smaller models.

6.2 MFU Analysis

The analysis of MFU further supports the observed trends in execution time. Compiled versions of both the GPT-2 and Shakespeare models demonstrated improved MFU relative to eager execution, with a notable increase in utilization for the Inductor backend (51.0% for GPT-2 and 29.9% for the Shakespeare model, compared to 43.3% and 22.9% in eager mode, respectively) (Table 2). These results indicate that model compilation not only reduces execution time but also enhances GPU utilization, allowing for more efficient hardware usage.

Interestingly, while MFU improved for the compiled models, the variance in MFU increased. This suggests that compilation introduces some degree of instability in hardware utilization, which could be a consequence of how computations are reordered or optimized by the compiler. Despite this, the variance of MFU was significantly lower for the larger GPT-2 model compared to the smaller Shakespeare model, indicating that larger models may benefit more from the optimizations offered by compilation.

6.3 The ResNet18 Anomaly

The anomaly observed in the ResNet18 benchmarking results warrants further investigation. Unlike the ResNet34 model, which exhibited improved performance with compilation, the ResNet18 model demonstrated an unexpected increase in execution time during compilation with the Inductor backend. One possible explanation is that PyTorch may apply internal optimizations during the eager execution of smaller models, which could be disabled or made less effective when using compilation. This suggests that the advantages of compilation might not be fully realized for smaller models, potentially due to heuristics or thresholds employed by the framework. More detailed profiling of the PyTorch internals during model training would be necessary to identify the root cause of this behavior.

6.4 Verification of PyTorch Compilation Claims

Our findings generally validate the claims made by the PyTorch Foundation regarding the benefits of using `torch.compile()`. Across multiple models and hardware configurations, we observed that (1) `torch.compile()` works as expected without requiring significant changes to the codebase, (2) it frequently provides notable performance improvements, particularly in terms of execution time and GPU utilization, and (3) it increases hardware efficiency as evidenced by the enhanced MFU values. These results are in line with PyTorch's documentation, which emphasizes the ease of use and automatic optimization features of `torch.compile()`.

6.5 Business Implications

The implications of these findings extend beyond academic research and can be directly applied in industry, particularly for businesses looking to optimize training time and resource utilization. For larger models such as GPT-2, where compilation provides a clear reduction in execution time and improved GPU utilization, adopting `torch.compile()` can result in substantial cost savings, especially in environments where model training is computationally expensive.

However, caution is advised for smaller models like ResNet18, where compilation did not provide the expected performance gains. In such cases, alternative compilation strategies, such as using TorchScript, might offer better performance without introducing the overhead seen with the Inductor backend. This highlights the need for businesses to tailor their model optimization strategies based on the specific characteristics of their models and workloads.

6.6 Limitations and Future Work

Although our benchmarking results are comprehensive, they are constrained by the hardware limitations that prevented the evaluation of larger models. Future work should focus on extending the benchmarks to larger CNN and LLM models to better understand how scaling affects compilation performance. Additionally, further investigation into the internal optimizations employed by PyTorch during eager execution could shed light on the observed anomalies with smaller models, such as ResNet18. Profiling tools could also be employed to provide more detailed insights into how compilation affects memory management and computational efficiency, particularly in mixed-precision training scenarios.

Overall, this study demonstrates the potential benefits of model compilation but also highlights the importance of considering model size and hardware constraints when optimizing training workflows. The general applicability of `torch.compile()` to a range of models suggests that it can be a valuable tool in improving training efficiency, but the presence of edge cases like the ResNet18 anomaly underscores the need for further research.

7 Conclusion

This study has evaluated the impact of model compilation on the training performance of Convolutional Neural Networks (CNNs) and Transformer-based models (Large Language Models, LLMs). Our findings demonstrate that model compilation, particularly with the Inductor backend, generally leads to significant reductions in execution time and improvements in hardware utilization, as evidenced by increased Mean Flop Utilization (MFU). These effects were particularly prominent in larger models, such as GPT-2 and Shakespeare, where compilation led to performance gains of up to 24% in execution time and increased GPU efficiency.

However, the results for smaller models, such as ResNet18, present a more complex picture. In some cases, compiled models exhibited slower execution times compared to eager execution, suggesting that model size and complexity play a critical role in determining the efficacy of compilation. This observation points to potential internal optimizations in PyTorch that may be more favorable for smaller models under eager execution.

Overall, this study confirms the advantages of utilizing PyTorch's `torch.compile()` for model training, especially for large-scale models. Nevertheless, the observed discrepancies in smaller models, such as the ResNet18 anomaly, highlight the need for further investigation into the conditions under which model compilation may be less effective. Future research should explore larger models and additional optimization strategies to refine the use of compilation techniques, particularly in varied hardware environments.

In conclusion, while model compilation offers significant benefits in terms of execution speed and hardware utilization, practitioners must consider model size and underlying optimizations when implementing these techniques in production environments. This study provides valuable insights into the trade-offs involved in model compilation and underscores the need for continued exploration in this area to fully unlock its potential across a broader range of applications.

References

[] *Apache TVM — tvm.apache.org.* `https://tvm.apache.org/`. [Accessed 19-09-2024].

[Dos+21] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.* 2021. arXiv: `2010.11929 [cs.CV]`. URL: `https://arxiv.org/abs/2010.11929`.

[Foua] PyTorch Foundation. *PyTorch 2.0: Our next generation release that is faster, more Pythonic and Dynamic as ever — pytorch.org.* `https://pytorch.org/blog/pytorch-2.0-release/`. [Accessed 10-08-2024].

[Foub] The PyTorch Foundation. *PyTorch Design Philosophy 2014; PyTorch 2.4 documentation — pytorch.org.* `https://pytorch.org/docs/stable/community/design.html`. [Accessed 12-09-2024].

[He+16] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: June 2016, pp. 770–778. DOI: `10.1109/CVPR.2016.90`.

[Kar] Andrew Karpathy. *GitHub - karpathy/nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs. — github.com.* `https://github.com/karpathy/nanoGPT`. [Accessed 21-08-2024].

[LeC+89] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667. DOI: `10.1162/neco.1989.1.4.541`. eprint: `https://direct.mit.edu/neco/article-pdf/1/4/541/811941/neco.1989.1.4.541.pdf`. URL: `https://doi.org/10.1162/neco.1989.1.4.541`.

[Rot+18] Nadav Rotem et al. *Glow: Graph Lowering Compiler Techniques for Neural Networks.* May 2018.

[Rus+15] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[teaa] ONNX dev team. *ONNX | Home — onnx.ai.* `https://onnx.ai/`. [Accessed 19-09-2024].

[teab] OpenXLA dev team. *OpenXLA Project — openxla.org.* `https://openxla.org/`. [Accessed 19-09-2024].

[TKC19] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: `10.1145/3315508.3329973`. URL: `https://doi.org/10.1145/3315508.3329973`.

[Wig] Ross Wightman. *GitHub - huggingface/pytorch-image-models: The largest collection of PyTorch image encoders / backbones. Including train, eval, inference, export scripts, and pretrained weights – ResNet, ResNeXT, EfficientNet, NFNet, Vision Transformer (ViT), MobileNetV4, MobileNet-V3 & V2, RegNet, DPN, CSPNet, Swin Transformer, MaxViT, CoAtNet, ConvNeXt, and more — github.com.* `https://github.com/huggingface/pytorch-image-models`. [Accessed 10-08-2024].

[Xco17] X.com. *X (formerly Twitter).* `https://x.com/karpathy/status/868178954032513024`. May 2017.

A Work sharing

There was no work sharing during this project.

B Code samples

B.1 Reproducing LLM training

Find below config and diff files to reproduce our results for LLM training. For more information, see Method section.

```bash
#! /bin/bash
#SBATCH -c 6
#SBATCH --mem 32G
#SBATCH -p grete:shared
#SBATCH -t 720
#SBATCH -G A100:1
source ~/.bashrc
mamba activate scap
[REPLACE THIS WITH TRAIN COMMAND]
```

Listing 1: .sbatch configuration file for reproducing the results for LLM benchmarking

```
Train commands for training shakespeare LLM:
python train.py config/train_shakespeare_char.py --compile=True \
--compile_inductor=True
python train.py config/train_shakespeare_char.py --compile=True \
--compile_cudagraphs=True
python train.py config/train_shakespeare_char.py --compile=False

Train commands for training GPT-2:
python train.py config/train_gpt2.py --compile=True \
--compile_inductor=True
python train.py config/train_gpt2.py --compile=True \
--compile_cudagraps=True
python train.py config/train_gpt2.py --compile=False
```

Listing 2: Commands for reproducing the results for LLM benchmarking to be used in 1

B.2 Reproducing CNN training

Find below config and diff files to reproduce our results for CNN training. Please note that no modification of timm source code is required. To exactly reproduce our experiments, use timm version 1.0.3. For more information, see Method section.

```
diff --git a/train.py b/train.py
index 951bda9..63c579a 100644
--- a/train.py
+++ b/train.py
@@ -72,6 +72,8 @@ backend = 'nccl' # 'nccl', 'gloo', etc.
 device = 'cuda' # examples: 'cpu', 'cuda', 'cuda:0', 'cuda:1' etc., \
 or try 'mps' on macbooks
 dtype = 'bfloat16' if torch.cuda.is_available() and \
 torch.cuda.is_bf16_supported() else 'float16' # 'float32', 'bfloat16',\
 or 'float16', the latter will auto implement a GradScaler
 compile = True # use PyTorch 2.0 to compile the model to be faster
+compile_inductor = False
+compile_cudagraphs = False
 # --------------------------------------------------------------------\
 ---------
 config_keys = [k for k,v in globals().items() if not k.startswith('_') \
 and isinstance(v, (int, float, bool, str))]
 exec(open('configurator.py').read()) # overrides from command line or \
 config file
@@ -205,7 +207,21 @@ checkpoint = None # free up memory
 if compile:
     print("compiling the model... (takes a ~minute)")
     unoptimized_model = model
-    model = torch.compile(model) # requires PyTorch 2.0
+
+    if compile_inductor:
+        print("Compiling with inductor")
+        model = torch.compile(model, backend="inductor")
+    elif compile_cudagraphs:
+        print("Compiling with cudagraphs")
+        torch.compile(model, backend="cudagraphs")
+    else:
+        print("compiling with default settings")
+        model = torch.compile(model) # requires PyTorch 2.0
+else:
+    print("Not compiling the model")
+
+
+

 # wrap model into DDP container
 if ddp:

```

Listing 3: Changes made to karpathy/nanoGPT for benchmarking different compilation methods

```bash
#! /bin/bash
#SBATCH -c 12
#SBATCH --mem 64G
#SBATCH -p grete-h100
#SBATCH -t 2880
#SBATCH -G H100:1
source ~/.bashrc
mamba activate scap_timm
[REPLACE THIS WITH TRAIN COMMAND]
```

Listing 4: .sbatch configuration file for reproducing the results for CNN benchmarking

```bash
Train commands for training resnet 18
python train.py /scratch/usr/nimdalin/scap_timm/imagenet-1k/data \
--model resnet18
python train.py /scratch/usr/nimdalin/scap_timm/imagenet-1k/data \
--model resnet18 --torchcompile inductor
python train.py /scratch/usr/nimdalin/scap_timm/imagenet-1k/data \
--model resnet18 --torchscript

Train commands for training resnet 34
python train.py /scratch/usr/nimdalin/scap_timm/imagenet-1k/data \
--model resnet34
python train.py /scratch/usr/nimdalin/scap_timm/imagenet-1k/data \
--model resnet34 --torchcompile inductor
python train.py /scratch/usr/nimdalin/scap_timm/imagenet-1k/data \
--model resnet34 --torchscript
```

Listing 5: Commands for reproducing the results for LLM benchmarking to be used in 4