PyTorch: An Imperative Style, High-Performance Deep Learning Library

Adam Paszke
University of Warsaw
adam.paszke@gmail.com

Sam Gross
Facebook AI Research
sgross@fb.com

Francisco Massa
Facebook AI Research
fmassa@fb.com

Adam Lerer
Facebook AI Research
alerer@fb.com

James Bradbury
Google
jekbradbury@gmail.com

Gregory Chanan
Facebook AI Research
gchanan@fb.com

Trevor Killeen
Self Employed
killeent@cs.washington.edu

Zeming Lin
Facebook AI Research
zlin@fb.com

Natalia Gimelshein
NVIDIA
ngimelshein@nvidia.com

Luca Antiga
Orobix
luca.antiga@orobix.com

Alban Desmaison
Oxford University
alban@robots.ox.ac.uk

Andreas Köpf
Xamla
andreas.koepf@xamla.com

Edward Yang
Facebook AI Research
ezyang@fb.com

Zach DeVito
Facebook AI Research
zdevito@cs.stanford.edu

Martin Raison
Nabla
martinraison@gmail.com

Alykhan Tejani
Twitter
atejani@twitter.com

Sasank Chilamkurthy
Qure.ai
sasankchilamkurthy@gmail.com

Benoit Steiner
Facebook AI Research
benoitsteiner@fb.com

Lu Fang
Facebook
lufang@fb.com

Junjie Bai
Facebook
jbai@fb.com

Soumith Chintala
Facebook AI Research
soumith@gmail.com

Abstract

Deep learning frameworks have often focused on either usability or speed, but not both. PyTorch is a machine learning library that shows that these two goals are in fact compatible: it provides an imperative and Pythonic programming style that supports code as a model, makes debugging easy and is consistent with other popular scientific computing libraries, while remaining efficient and supporting hardware accelerators such as GPUs.

In this paper, we detail the principles that drove the implementation of PyTorch and how they are reflected in its architecture. We emphasize that every aspect of PyTorch is a regular Python program under the full control of its user. We also explain how the careful and pragmatic implementation of the key components of its runtime enables them to work together to achieve compelling performance.

We demonstrate the efficiency of individual subsystems, as well as the overall speed of PyTorch on several common benchmarks.

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

1 Introduction

With the increased interest in deep learning in recent years, there has been an explosion of machine learning tools. Many popular frameworks such as Caffe [1], CNTK [2], TensorFlow [3], and Theano [4], construct a static dataflow graph that represents the computation and which can then be applied repeatedly to batches of data. This approach provides visibility into the whole computation ahead of time, and can theoretically be leveraged to improve performance and scalability. However, it comes at the cost of ease of use, ease of debugging, and flexibility of the types of computation that can be represented.

Prior work has recognized the value of dynamic eager execution for deep learning, and some recent frameworks implement this define-by-run approach, but do so either at the cost of performance (Chainer [5]) or using a less expressive, faster language (Torch [6], DyNet [7]), which limits their applicability.

However, with careful implementation and design choices, dynamic eager execution can be achieved largely without sacrificing performance. This paper introduces PyTorch, a Python library that performs immediate execution of dynamic tensor computations with automatic differentiation and GPU acceleration, and does so while maintaining performance comparable to the fastest current libraries for deep learning. This combination has turned out to be very popular in the research community with, for instance, 296 ICLR 2019 submissions mentioning PyTorch.

2 Background

Four major trends in scientific computing have become increasingly important for deep learning.

First, starting in the 1960s, the development of domain specific languages such as APL [8], MATLAB [9], R [10] and Julia [11], turned multidimensional arrays (often referred to as tensors) into first-class objects supported by a comprehensive set of mathematical primitives (or operators) to manipulate them. Separately, libraries such as NumPy[12], Torch[6], Eigen[13] and Lush[14] made **array-based programming** productive in general purpose languages such as Python, Lisp, C++ and Lua.

Second, the development of **automatic differentiation** [15] made it possible to fully automate the daunting labor of computing derivatives. This made it significantly easier to experiment with different machine learning approaches while still allowing for efficient gradient based optimization. The autograd [16] package popularized the use of this technique for NumPy arrays, and similar approaches are used in frameworks such as Chainer [5], DyNet [7], Lush [14], Torch [6], Jax [17] and Flux.jl [18].

Third, with the advent of the free software movement, the scientific community moved away from closed proprietary software such as Matlab[9], and towards the **open-source Python ecosystem** with packages like NumPy [12], SciPy [19], and Pandas [20]. This fulfilled most of the numerical analysis needs of researchers while allowing them to take advantage of a vast repository of libraries to handle dataset preprocessing, statistical analysis, plotting, and more. Moreover, the openness, interoperability, and flexibility of free software fostered the development of vibrant communities that could quickly address new or changing needs by extending the existing functionality of a library or if needed by developing and releasing brand new ones. While there is a rich offering of open-source software for neural networks in languages other than Python, starting with Lush [14] in Lisp, Torch [6] in C++, Objective-C and Lua, EBLearn [21] in C++, Caffe [1] in C++, the network effects of a large ecosystem such as Python made it an essential skill to jumpstart one's research. Hence, since 2014, most deep learning frameworks converged on a Python interface as an essential feature.

Finally, the availability and commoditization of general-purpose massively parallel hardware such as GPUs provided the computing power required by deep learning methods. Specialized libraries such as cuDNN [22], along with a body of academic work (such as [23] and [24]), produced a set of high-performance reusable deep learning kernels that enabled frameworks such as Caffe [1], Torch7 [25], or TensorFlow [3] to take advantage of these **hardware accelerators**.

PyTorch builds on these trends by providing an array-based programming model accelerated by GPUs and differentiable via automatic differentiation integrated in the Python ecosystem.

3 Design principles

PyTorch's success stems from weaving previous ideas into a design that balances speed and ease of use. There are four main principles behind our choices:

Be Pythonic Data scientists are familiar with the Python language, its programming model, and its tools. PyTorch should be a first-class member of that ecosystem. It follows the commonly established design goals of keeping interfaces simple and consistent, ideally with one idiomatic way of doing things. It also integrates naturally with standard plotting, debugging, and data processing tools.

Put researchers first PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible. The complexity inherent to machine learning should be handled internally by the PyTorch library and hidden behind intuitive APIs free of side-effects and unexpected performance cliffs.

Provide pragmatic performance To be useful, PyTorch needs to deliver compelling performance, although not at the expense of simplicity and ease of use. Trading 10% of speed for a significantly simpler to use model is acceptable; 100% is not. Therefore, its *implementation* accepts added complexity in order to deliver that performance. Additionally, providing tools that allow researchers to manually control the execution of their code will empower them to find their own performance improvements independent of those that the library provides automatically.

Worse is better [26] Given a fixed amount of engineering resources, and all else being equal, the time saved by keeping the internal implementation of PyTorch simple can be used to implement additional features, adapt to new situations, and keep up with the fast pace of progress in the field of AI. Therefore it is better to have a simple but slightly incomplete solution than a comprehensive but complex and hard to maintain design.

4 Usability centric design

4.1 Deep learning models are just Python programs

In a surprisingly short amount of time, machine learning grew from recognizing individual digits [27] into autonomously playing StarCraft [28]. Consequently, the neural networks themselves evolved rapidly from simple sequences of feed forward layers into incredibly varied numerical programs often composed of many loops and recursive functions. To support this growing complexity, PyTorch foregoes the potential benefits of a graph-metaprogramming based approach to preserve the imperative programming model of Python. This design was pioneered for model authoring by Chainer[5] and Dynet[7]. PyTorch extends this to all aspects of deep learning workflows. Defining layers, composing models, loading data, running optimizers, and parallelizing the training process are all expressed using the familiar concepts developed for general purpose programming.

This solution ensures that any new potential neural network architecture can be easily implemented with PyTorch. For instance, layers (which in modern machine learning should really be understood as stateful functions with implicit parameters) are typically expressed as Python classes whose constructors create and initialize their parameters, and whose forward methods process an input activation. Similarly, models are usually represented as classes that compose individual layers, but let us state again that nothing forces the user to structure their code in that way. Listing 1 demonstrates how an entire model can be created by composing functionality provided by PyTorch such as 2d convolution, matrix multiplication, dropout, and softmax to classify gray-scale images. Note that linear layers are of course part of the library, but we show an example implementation to highlight how simple it is.

```
class LinearLayer(Module):
   def __init__(self, in_sz, out_sz):
      super().__init__()
      t1 = torch.randn(in_sz, out_sz)
      self.w = nn.Parameter(t1)
      t2 = torch.randn(out_sz)
      self.b = nn.Parameter(t2)

   def forward(self, activations):
      t = torch.mm(activations, self.w)
      return t + self.b
```

```
class FullBasicModel(nn.Module):
   def __init__(self):
      super().__init__()
      self.conv = nn.Conv2d(1, 128, 3)
      self.fc = LinearLayer(128, 10)

   def forward(self, x):
      t1 = self.conv(x)
      t2 = nn.functional.relu(t1)
      t3 = self.fc(t1)
      return nn.functional.softmax(t3)
```

Listing 1: A custom layer used as a building block for a simple but complete neural network.

This "everything is a just a program" philosophy is not limited to just the models, and applies to optimizers and data loaders as well. This facilitates the experimentation of new training techniques. For example, to implement the very popular generative adversarial networks, one needs to specify two separate models (the generator and the discriminator), and two loss functions that depend on both models at the same time. Rigid APIs would struggle with this setup, but the simple design employed in PyTorch easily adapts to this setting as shown in Listing 2.

```
discriminator = create_discriminator()
generator = create_generator()
optimD = optim.Adam(discriminator.parameters())
optimG = optim.Adam(generator.parameters())

def step(real_sample):
  # (1) Update Discriminator
  errD_real = loss(discriminator(real_sample), real_label)
  errD_real.backward()
  fake = generator(get_noise())
  errD_fake = loss(discriminator(fake.detach(), fake_label)
  errD_fake.backward()
  optimD.step()
  # (2) Update Generator
  errG = loss(discriminator(fake), real_label)
  errG.backward()
  optimG.step()
```

Listing 2: Simplified training of a generative adversarial networks.

Since PyTorch programs execute eagerly, all the features of Python are available throughout the whole design process. Print statements, standard debuggers, and common visualization tools like matplotlib all work as expected. Users do not have to wait for lengthy compilation before they can start running their programs, and more importantly intermediate computations can be observed to understand how a model works and whether its results are correct.

4.2 Interoperability and extensibility

Easy and efficient interoperability is one of the top priorities for PyTorch because it opens the possibility to leverage the rich ecosystem of Python libraries as part of user programs. Hence, PyTorch allows for bidirectional exchange of data with external libraries. For example, it provides a mechanism to convert between NumPy arrays and PyTorch tensors using the `torch.from_numpy()` function and `.numpy()` tensor method. Similar functionality is also available to exchange data stored using the DLPack [29] format. Note that this exchange happens in both cases without any data copying – objects on both sides only describe how to interpret a memory region which is shared among them. Hence, those operations are actually extremely cheap, and take constant time no matter how large the converted arrays are.

Moreover, many of the critical systems are designed specifically to be extensible. For instance, the automatic differentiation system allows users to add support for custom differentiable functions. To do that users can define a new subclass of `torch.autograd.Function` that implements `forward()` and `backward()` methods, which specify the function and its derivative (or more formally the vector-Jacobian product). Similarly new datasets can be added by subclassing `torch.utils.data.Dataset` and implementing two methods: `__getitem__` (the indexing operator) and `__len__` (the length operator), making datasets behave like (possibly lazy) lists. How these work is completely up to the implementer, and many users leverage other Python packages for data loading. The `DataLoader` class consumes objects conforming to this interface and provides an iterator over the data which takes care of shuffling, batching, parallelization, and management of pinned CUDA memory to improve throughput.

Most importantly, users are free to replace any component of PyTorch that does not meet the needs or performance requirements of their project. They are all designed to be completely interchangeable, and PyTorch takes great care not to impose any particular solution.

4.3 Automatic differentiation

Since gradient based optimization is vital to deep learning, PyTorch must be able to automatically compute gradients of models specified by our users, and those can be arbitrary Python programs. However, Python is a dynamic programming language that allows changing most behaviors at runtime, making ahead of time source-to-source differentiation cumbersome. Instead, PyTorch uses the operator overloading approach, which builds up a representation of the computed function every time it is executed. In its current implementation [30], PyTorch performs reverse-mode automatic differentiation, which computes the gradient of a scalar output with respect to a multivariate input. Differentiating functions with more outputs than inputs is more efficiently executed using forward-mode automatic differentiation, but this use case is less common for machine learning applications. PyTorch can be easily extended to perform forward-mode differentiation using array-level dual numbers [31, 32].

Another interesting and uncommon feature of our system is that it can differentiate through code employing mutation on tensors, which is one of the basic building blocks of imperative programs. To ensure safety, we have implemented a versioning system for tensors, which lets us track their modifications and ensure that we always use the data we expect. One interesting tradeoff is that while we could utilize techniques like copy-on-write to support arbitrary programs, we chose to not go down this path, as performance-wise it is usually beneficial for the users to rewrite their code to ensure that no copies have to be performed. Hence, while most mutations are benign and can be handled automatically, the really complicated cases result in a user error, which lets them know that they likely want to restructure the program. This allows us to avoid introducing subtle and hard-to-find performance cliffs.

5 Performance focused implementation

Running deep learning algorithms efficiently from a Python interpreter is notoriously challenging: for instance, the global interpreter lock [33] effectively ensures that only one of any number of concurrent threads is running at any given time. Deep learning frameworks based on the construction of a static data-flow graph sidestep this problem by deferring the evaluation of the computation to a custom interpreter.

PyTorch solved the problem differently, by carefully optimizing every aspect of its execution while simultaneously empowering its users to easily leverage additional optimization strategies.

5.1 An efficient C++ core

Despite being closely integrated in the Python ecosystem, most of PyTorch is written in C++ to achieve high performance. This core `libtorch` library implements the tensor data structure, the GPU and CPU operators, and basic parallel primitives. It also provides the automatic differentiation system, including the gradient formulas for most built-in functions. This ensures that the computation of the derivatives of functions composed of core PyTorch operators is executed entirely in a multithreaded evaluator which does not require holding the Python global interpreter lock [33]. Python bindings

are generated using YAML meta-data files. An interesting side-effect of this approach is that it allowed our community to quickly create bindings to multiple other languages resulting in projects like NimTorch [34], hasktorch [35] and others.

This design also allowed us to create first-class C++ bindings and modeling libraries that can be used in places where Python is inconvenient, such as the game engine for Starcraft [36] or on mobile platforms. It is even possible to take the Python code describing a PyTorch model and run it without Python using the TorchScript engine [37].

5.2 Separate control and data flow

PyTorch maintains a strict separation between its control (i.e. program branches, loops) and data flow (i.e. tensors and the operations performed on them). The resolution of the control flow is handled by Python and optimized C++ code executed on the host CPU, and result in a linear sequence of operator invocations on the device. Operators can be run either on CPU or on GPU.

PyTorch is designed to execute operators asynchronously on GPU by leveraging the CUDA stream mechanism [38] to queue CUDA kernel invocations to the GPUs hardware FIFO. This allows the system to overlap the execution of Python code on CPU with tensor operators on GPU. Because the tensor operations usually take a significant amount of time, this lets us saturate the GPU and reach peak performance even in an interpreted language with fairly high overhead like Python. Note that this mechanism is nearly invisible to the user. Unless they implement their own multi-stream primitives all of the CPU-GPU synchronization is handled by the library.

PyTorch could leverage a similar mechanism to also execute operators asynchronously on the CPU. However the costs of cross-thread communication and synchronization would negate the performance benefit of such an optimization.

5.3 Custom caching tensor allocator

Almost every operator must dynamically allocate an output tensor to hold the result of its execution. It is therefore critical to optimize the speed of the dynamic memory allocators. PyTorch can rely on optimized libraries [39, 40, 41] to handle this task on CPU. However, on GPU the `cudaFree` routine may block its caller until all previously queued work on all GPUs completes. To avoid this bottleneck, PyTorch implements a custom allocator which incrementally builds up a cache of CUDA memory and reassigns it to later allocations without further use of CUDA APIs. The incremental allocation is also crucial for better interoperability, because taking up all GPU memory ahead of time would prevent the user from utilizing other GPU-enabled Python packages.

To further improve its effectiveness, this allocator was tuned for the specific memory usage patterns of deep learning. For example, it rounds up allocations to multiples of 512 bytes to avoid fragmentation issues. Moreover, it maintains a distinct pool of memory for every CUDA stream (work queue).

The one-pool-per-stream design assumption simplifies the implementation and improves the performance of the allocator: because the CPU runs ahead of the GPU, memory is freed on the CPU *before* its last use on the GPU finishes. Since streams serialize execution, if the free precedes the reallocation on the CPU, the same order will occur on the GPU. So the allocator can reallocate memory freed on the CPU immediately as long as the new allocation is used on the same stream as the freed region. However, if an allocation was last used on one stream and then allocated on another, additional synchronization is needed.

The one-pool-per-stream design seems limiting since the allocations end up fragmented per stream, but in practice PyTorch almost never uses multiple streams. It is notoriously hard to write CUDA kernels in a way that would let them cooperatively share the GPU because exact scheduling is hardware controlled. In practice, kernel writers usually resort to monolithic kernels that combine multiple tasks. Data loading and distributed computing utilities are exceptions to the one stream design, and they carefully insert additional synchronization to avoid bad interactions with the allocator.

While this design is susceptible to certain corner cases, it almost never exhibits unwanted behaviors in practical code. Most of our users are not aware of its existence.

5.4 Multiprocessing

Due to the global interpreter lock (GIL) Python's default implementation does not allow concurrent threads to execute in parallel. To alleviate this problem, the Python community has established a standard `multiprocessing` module, containing a number of utilities that allow users to easily spawn child processes and implement basic inter-process communication primitives.

However, the implementation of the primitives uses the same form of serialization used for on-disk persistence, which is inefficient when dealing with large arrays. Hence, PyTorch extends the Python `multiprocessing` module into `torch.multiprocessing`, which is a drop-in replacement for the built in package and automatically moves the data of tensors sent to other processes to shared memory instead of sending it over the communication channel.

This design greatly improves performance and makes the process isolation weaker, resulting in a programming model which more closely resembles regular threaded programs. Users can easily implement heavily parallel programs that operate on independent GPUs but later synchronize gradients using all-reduce style primitives.

Another unique feature of this system is that it transparently handles sharing of CUDA tensors, making it easy to implement techniques like Hogwild [42].

5.5 Reference counting

Users often design their models to utilize all memory available during training, and increasing batch sizes is a common technique of speeding up the process. Therefore, to deliver great performance, PyTorch has to treat memory as a scarce resource that it needs to manage carefully.

Libraries with eager semantics have to manage tensor memory without knowing how it will be used in the future. Garbage collection is the typical way to handle this automatically because it has good amortized performance. In this approach, the runtime periodically investigates the state of the system, enumerates used objects and frees everything else. However, by deferring the deallocation, it causes the program to use more memory overall [43]. Given the scarcity of GPU memory, these overheads are unacceptable. In fact, Torch7 utilized the garbage collector built into Lua, and a common anti-pattern among the users was to sprinkle the program with explicit triggers to the garbage collector, hoping that the memory errors go away.

PyTorch takes a different approach: it relies on a reference counting scheme to track the number of uses of each tensor, and frees the underlying memory *immediately* once this count reaches zero. Note that PyTorch tracks both references internal to the `libtorch` library and external references made by users in their Python code by integrating with Python's own reference counting mechanism. This ensures that memory is released exactly when tensors become unneeded.

One notable caveat is that we can only guarantee the desired performance characteristics in implementations of languages that either already utilize reference counting (CPython, Swift, but not PyPy or many scripting languages such as Lua), and those that allow for user-defined behavior for assignment, copies, and moves (e.g. C++, Rust). Bindings to implementations that do not satisfy those criteria will have to implement their own specialized memory management on top of PyTorch.

6 Evaluation

In this section we compare the performance of PyTorch with several other commonly-used deep learning libraries, and find that it achieves competitive performance across a range of tasks. All experiments were performed on a workstation with two Intel Xeon E5-2698 v4 CPUs and one NVIDIA Quadro GP100 GPU.

6.1 Asynchronous dataflow

We start by quantifying the ability of PyTorch to asynchronously execute dataflow on GPU. We use the built-in profiler [44] to instrument various benchmarks and record a timeline of the execution of a single training step.

Figure 1 shows a representative timeline of execution for the first few operations of a ResNet-50 model. The host CPU which queues the work quickly outpaces the execution of the operators on the GPU. This allows PyTorch to achieve almost perfect device utilization. In this example, GPU execution takes around three times longer than CPU scheduling. The exact ratio depends on the relative performance of the host CPU and the GPU, as well as the number of elements in each tensor and the average arithmetic complexity of the floating point computations to be performed on the GPU.

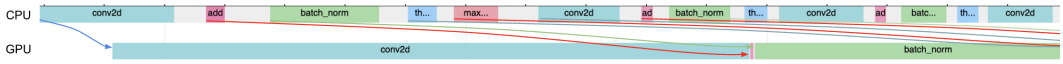

Figure 1: A trace of the first few operators of Resnet-50. The top row depicts the execution of the control flow running on the host CPU. The gray areas are Python code executed by its interpreter. The colored areas correspond to the work done on the host CPU to queue various operators (convolution, batch normalization, and so on). The bottom row shows the corresponding execution of those operators on the GPU. The arrows pair the two events in time.

6.2 Memory management

We used the NVIDIA profiler to trace the execution of the CUDA runtime as well as the execution of the CUDA kernels launched during one training iteration of the ResNet-50 model. As shown in Figure 2, the behavior of the first iteration differs significantly from that of subsequent ones. At first, calls to the CUDA memory management functions (`cudaMalloc` and `cudaFree`) slow down the execution quite dramatically by blocking the CPU thread for long periods of time, hence lowering the utilization of the GPU. This effect disappears in subsequent iterations as the PyTorch caching memory allocator starts reusing previously allocated regions.

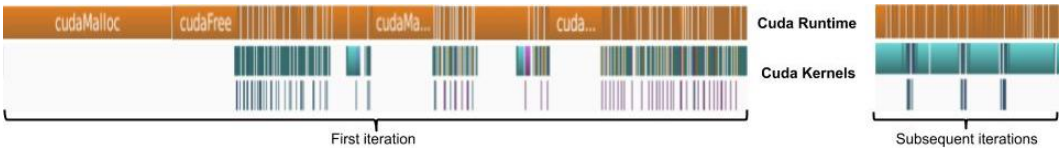

Figure 2: Annotated traces of the execution of ResNet-50 on GPU.

6.3 Benchmarks

Finally, we can get an overall sense of single-machine eager mode performance of PyTorch by comparing it to three popular graph-based deep learning frameworks (CNTK, MXNet and TensorFlow), a define-by-run framework (Chainer), and production oriented platform (PaddlePaddle). The Appendix details all the steps needed to reproduce our setup.

Our results are summarized in Table 1. On all the benchmarks, the performance of PyTorch is within 17% of that of of the fastest framework. We attribute this result to the fact that these tools offload most of the computation to the same version of the cuDNN and cuBLAS libraries.

Framework	*Throughput (higher is better)*					
	AlexNet	VGG-19	ResNet-50	MobileNet	GNMTv2	NCF
Chainer	778 ± 15	N/A	**219** ± 1	N/A	N/A	N/A
CNTK	845 ± 8	84 ± 3	210 ± 1	N/A	N/A	N/A
MXNet	**1554** ± 22	113 ± 1	218 ± 2	444 ± 2	N/A	N/A
PaddlePaddle	933 ± 123	112 ± 2	192 ± 4	**557** ± 24	N/A	N/A
TensorFlow	1422 ± 27	66 ± 2	200 ± 1	216 ± 15	$9631 \pm 1.3\%$	$4.8e6 \pm 2.9\%$
PyTorch	1547 ± 316	**119** ± 1	212 ± 2	463 ± 17	**15512** $\pm 4.8\%$	**5.4e6** $\pm 3.4\%$

Table 1: Training speed for 6 models using 32bit floats. Throughput is measured in images per second for the AlexNet, VGG-19, ResNet-50, and MobileNet models, in tokens per second for the GNMTv2 model, and in samples per second for the NCF model. The fastest speed for each model is shown in bold.

6.4 Adoption

The validity of design decisions and their impact on ease-of-use is hard to measure. As a proxy, we tried to quantify how well the machine learning community received PyTorch by counting how often various machine learning tools (including Caffe, Chainer, CNTK, Keras, MXNet, PyTorch, TensorFlow, and Theano) are mentioned on arXiv e-Prints since the initial release of PyTorch in January 2017. In Figure 3 we report the monthly number of mentions of the word "PyTorch" as a percentage of all mentions among these deep learning frameworks. We counted tools mentioned multiple times in a given paper only once, and made the search case insensitive to account for various spellings.

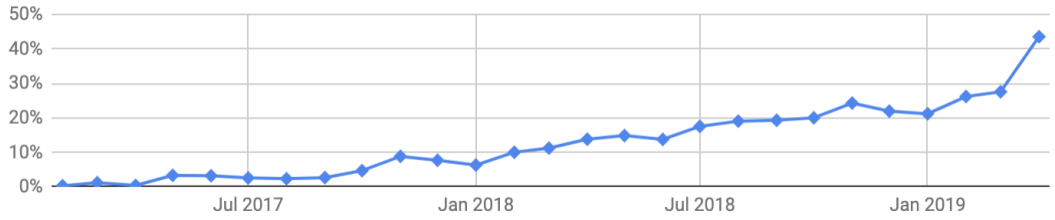

Figure 3: Among arXiv papers each month that mention common deep learning frameworks, percentage of them that mention PyTorch.

7 Conclusion and future work

PyTorch has become a popular tool in the deep learning research community by combining a focus on usability with careful performance considerations. In addition to continuing to support the latest trends and advances in deep learning, in the future we plan to continue to improve the speed and scalability of PyTorch. Most notably, we are working on the PyTorch JIT: a suite of tools that allow PyTorch programs to be executed outside of the Python interpreter where they can be further optimized. We also intend to improve support for distributed computation by providing efficient primitives for data parallelism as well as a Pythonic library for model parallelism based around remote procedure calls.

8 Acknowledgements

We are grateful to the PyTorch community for their feedback and contributions that greatly influenced the design and implementation of PyTorch. We thank all the PyTorch core team members, contributors and package maintainers including Ailing Zhang, Alex Suhan, Alfredo Mendoza, Alican Bozkurt, Andrew Tulloch, Ansha Yu, Anthony Shoumikhin, Bram Wasti, Brian Vaughan, Christian Puhrsch, David Reiss, David Riazati, Davide Libenzi, Dmytro Dzhulgakov, Dwaraj Rajagopal, Edward Yang, Elias Ellison, Fritz Obermeyer, George Zhang, Hao Lu, Hong Xu, Hung Duong, Igor Fedan, Ilia Cherniavskii, Iurii Zdebskyi, Ivan Kobzarev, James Reed, Jeff Smith, Jerry Chen, Jerry Zhang, Jiakai Liu, Johannes M. Dieterich, Karl Ostmo, Lin Qiao, Martin Yuan, Michael Suo, Mike Ruberry, Mikhail Zolothukhin, Mingzhe Li, Neeraj Pradhan, Nick Korovaiko, Owen Anderson, Pavel Belevich, Peter Johnson, Pritam Damania, Raghuraman Krishnamoorthi, Richard Zou, Roy Li, Rui Zhu, Sebastian Messmer, Shen Li, Simon Wang, Supriya Rao, Tao Xu, Thomas Viehmann, Vincent Quenneville-Belair, Vishwak Srinivasan, Vitaly Fedyunin, Wanchao Liang, Wei Yang, Will Feng, Xiaomeng Yang, Xiaoqiang Zheng, Xintao Chen, Yangqing Jia, Yanli Zhao, Yinghai Lu and Zafar Takhirov.

References

[1] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[2] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[3] 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, 2015. Software available from tensorflow.org.

[4] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[5] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[6] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[7] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin. DyNet: The Dynamic Neural Network Toolkit. *ArXiv e-prints*, January 2017.

[8] Philip S. Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970.

[9] The MathWorks, Inc., Natick, Massachusetts, United States. *MATLAB and Statistics Toolbox*.

[10] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

[11] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[12] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006. http://www.numpy.org/.

[13] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[14] Y LeCun and L Bottou. Lush reference manual. Technical report, code available at http://lush.sourceforge.net, 2002.

[15] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, January 2017.

[16] Dougal Maclaurin. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. PhD thesis, Harvard University, April 2016.

[17] Matthew Johnson et. al. Jax. `https://github.com/google/jax`, 2018.

[18] Mike Innes et. al. Flux.jl. `https://github.com/FluxML/Flux.jl`, 2018.

[19] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. http://www.scipy.org/.

[20] Wes McKinney. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference, 51-56*, 2010.

[21] Pierre Sermanet, Koray Kavukcuoglu, and Yann LeCun. Eblearn: Open-source energy-based learning in c++. In *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 693–697. IEEE, 2009.

[22] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan D. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[23] Andrew Lavin. maxdnn: An efficient convolution kernel for deep learning with maxwell gpus, January 2015.

[24] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[25] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *NIPS 2011*, 2011.

[26] Richard Gabriel. The rise of worse is better. http://dreamsongs.com/RiseOfWorseIsBetter.html.

[27] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/.

[28] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[29] DMLC. Dlpack: Open in memory tensor structure. https://github.com/dmlc/dlpack.

[30] 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 Workshop*, 2017.

[31] Dan Piponi. Automatic differentiation, C++ templates, and photogrammetry. *J. Graphics, GPU, & Game Tools*, 9(4):41–55, 2004.

[32] Holger Leuck and Hans-Hellmut Nagel. Automatic differentiation facilitates of-integration into steering-angle-based road vehicle tracking. In *1999 Conference on Computer Vision and Pattern Recognition (CVPR '99), 23-25 June 1999, Ft. Collins, CO, USA*, pages 2360–2365, 1999.

[33] The Python team. The cpython global interpreter lock. https://wiki.python.org/moin/GlobalInterpreterLock.

[34] Giovanni Petrantoni and Jörg Wollenschläger. Nimtorch. https://github.com/fragcolor-xyz/nimtorch.

[35] Austin Huang, Junji Hashimoto, and Sam Stites. Hasktorch. https://github.com/hasktorch/hasktorch.

[36] G. Synnaeve, Z. Lin, J. Gehring, D. Gant, V. Mella, V. Khalidov, N. Carion, and N. Usunier. Forward modeling for partial observation strategy games - a starcraft defogger. In *Advances in Neural Information Processing Systems*, pages 10761–10771, 2018.

[37] The PyTorch team. *Torch Script*. https://pytorch.org/docs/stable/jit.html.

[38] Justin Luitjens. Cuda streams. GPU technology conference, 2014.

[39] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[40] J. Evans. A scalable concurrent malloc(3) implementation for freebsd. In *In BSDCan — The Technical BSD Conference*, May 2006.

[41] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc.

[42] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pages 693–701, 2011.

[43] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM.

[44] The PyTorch team. *Pytorch Autograd Profiler*. https://pytorch.org/docs/1.0.1/autograd.html#profiler.