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

1. Introduction

Recently, deep learning frameworks have been increasing a lot, but PyTorch is one of the best performing python libraries by running dynamic tensor computations immediately through automatic differentiation and GPU acceleration.

2. Background

Large-scale parallel hardware such as GPUs, such as automating differentials, making software open source, and increasing productivity by creating libraries to deal with mathematical elements, are on the rise. PyTorch provides array-based programming models based on this trend.

3. Design Principles

Pytorch's success stems from the following four original positions

- The interface can be maintained and worked within the Python ecosystem so that people familiar with this language can use it without difficulty.

- Easily and productively create models, data loaders, and optimization tools.

- It provides ease of use and should have good performance.

- By keeping the internal implementation of pytorch simple, the time saved can be used to adapt to new situations, implement additional features, and keep up with fast progress.

4. Usability centric design

a. To support developing and increasing complexity, pytorch preserves python's imperative programming model. In addition, the philosophy of 'everything is just a program' applies to optimizers and data loaders, not limited to the model.

b. PyTorch allows two-way data exchange with external libraries, making it easy and efficient to scale the system.

c. PyTorch computes automatic differentiation, a gradient-based optimization essential for deep learning.

5. Performance focused implementation

a. Despite being a Python-based ecosystem, most of PyTorch's core libraries are written in C++ for high performance.

b. Separation between control and data flow is maintained, which is handled by an optimized C++ code. PyTorch is designed to execute CUDA kernel calls on the GPU asynchronously, so it can saturate the GPU and reach the highest performance.

c. PyTorch dynamically allocates output tensors from the CPU using optimized libraries [39,40,41].

d. Python's multiprocessing module was expanded, which improves performance and weakens process isolation. This makes it easy to implement large-scale parallel programs, and makes it easy to implement technologies such as Hogwild by transparently processing CUDA tensor sharing.

e. The reference calculation method is used to reference count the number of uses of each tensor and release the default memory. The memory is released when the tensor is not needed.

6. Evaluation

In this section, we compare the performance of PyTorch with several other deep learning libraries.

a. In asynchronous dataflows, we quantitatively measured PyTorch's GPU's ability to run dataflows asynchronously. We show that queuing tasks by host CPUs is much faster than performing operations on GPUs.

b. Initially, CUDA memory management functions tend to block CPU threads for long periods of time, reducing GPU utilization. However, this effect disappears when the PyTorch caching memory allocator starts reusing the previously allocated area.

c. We compare the single-machine easy mode performance to three popular graph-based deep learning frameworks (CNTK, MXNet, and TensorFlow). Our results are attributed to the fact that PyTorch's performance is around 17% of the fastest framework, and that these tools move most of the computations to the same version of cuDNN and cuBLAS libraries.

d. Since January 2017, when PyTorch was first released, we have counted the number of times the word "PyTorch" has been mentioned in arXive-Prints. Figure 3 shows the monthly percentage of the word "PyTorch" in all of the papers mentioned during the main deep learning framework.

7. Conclusion and future work

PyTorch has become a popular tool in deep learning research due to its ease of use and performance considerations, and plans to continuously improve it.