Share the Tensor Tea: How Databases can Leverage the Machine Learning Ecosystem

Yuki Asada*[2], Victor Fu*[2], Apurva Gandhi*[1], Advitya Gemawat*[1], Lihao Zhang*[2], Dong He[4], Vivek Gupta[3], Ehi Nosakhare[1], Dalitso Banda[1], Rathijit Sen[1], Matteo Interlandi[1]
[1,2,3]Microsoft, [4]University of Washington
{[1]<firstname>.<lastname>,[2]<firstname><lastname>,[3]vivgupt}@microsoft.com, [4]donghe@cs.washington.edu

ABSTRACT
We demonstrate Tensor Query Processor (TQP): a query processor that automatically compiles relational operators into tensor programs. By leveraging tensor runtimes such as PyTorch, TQP is able to: (1) integrate with ML tools (e.g., Pandas for data ingestion, Tensorboard for visualization); (2) target different hardware (e.g., CPU, GPU) and software (e.g., browser) backends; and (3) end-to-end accelerate queries containing both relational and ML operators. TQP is generic enough to supports the TPC-H benchmark, and it provides performance that are comparable to, and often better than, that of specialized CPU and GPU query processors.

PVLDB Reference Format:
Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, Matteo Interlandi. Share the Tensor Tea: How Databases can Leverage the Machine Learning Ecosystem. PVLDB, 15(12): 3598 - 3601, 2022.
doi:10.14778/3554821.3554853

1 INTRODUCTION

The free lunch is over! While from 1985 to 2010 the CPU performance doubled approximately every 1.5 years, going forward the expectation is that CPU performance will only double every 20 years [6]. In this post-Moore's Law era, taking advantage of hardware acceleration is paramount. Both the database and the machine learning (ML) communities are well aware of this. In fact, in the last decade we have been witnessing a growing investment in techniques able to exploit "commodity" accelerators such as SIMD instructions and GPGPUs. ML especially is becoming so predominant that many devices now come with ML-specific hardware accelerators (e.g., Apple Neural Engine, NVIDIA Tensor Cores), and the amount of money poured by venture capitalists on startups focusing on new hardware for ML is soaring [17]. This trend is mostly driven by the computation requirements of the state-of-the-art computer vision and NLP models.

However, programming hardware accelerators is notoriously hard. The ML community has a remarkable tool for easing the authoring and the deployment of ML models: the *tensor abstraction* [9]. Using this abstraction, ML practitioners can enjoy writing their models in high-level languages, while being free to run them on any hardware and backend of their choice: let it be CPUs, GPUs, custom ASICs, edge devices, or even browsers [11]. In the middle, ML frameworks such as PyTorch [15] are able to map tensor programs into efficient executions over the target backends. In this paper, we will refer to any ML framework supporting tensor programs as Tensor Computation Runtimes (TCRs).

Unfortunately, if we look at the analytical database space, we see a different picture. In the last several decades, the database community has been focusing on squeezing every last bit of performance from hardware devices (e.g., SIMD, GPUs), but, as a matter of fact, all these efforts are fragmented, specialized, and unrelated to each other, whereby users cannot enjoy the ease of deployment as well as the thriving ecosystem which makes Data Science (DS) and ML so popular.

In [8], we showed how this gap can be filled thanks to Tensor Query Processor (TQP [1]): a SQL query processor built on top of TCRs. In this demonstration we will showcase TQP features. Users of TQP can: (1) take advantage of the flexibility and performance of TCRs, and execute their relational queries over any supported hardware (e.g., CPU, GPU, TPU) and software (e.g., browser) backends; (2) end-to-end accelerate queries containing mix of relational and ML operators such as prediction queries [10]; and (3) seamlessly integrate with DS workflows and ML tools (e.g., TQP is `pip`-installable, and it leverages ML libraries such as Pandas and Tensorboard for data ingestion and visualization, respectively). TQP is integrated with Apache Spark [20] and PyTorch: it accepts input as a Spark SQL physical plan, and it uses a novel compilation stack based on [12] to lower relational operators into tensor programs implemented in PyTorch. TQP is expressive enough to support all the 22 queries composing the TPC-H benchmark, and it often outperforms specialized CPU and GPU query processors. For example, on Q6 and Q14 at scale factor 1, TQP is more than 3× faster than Apache Spark on CPU, and more than 4× faster than BlazingSQL [4] on GPU. While TQP is still in the prototype phase, we are actively investigating how it can be leveraged by Microsoft products, e.g., through ONNX Runtime (ORT) integration [10].

To showcase the versatility and performance of TQP, in this demo we will guide the audience through three different scenarios:

- *In the first scenario, we will show how TQP can be integrated with DS tools and libraries* (Section 3.1). Specifically, we will showcase our integration with Tensorboard [1], and how this can be used for visualizing query plans (tensor programs), and profiling query characteristics as well as operator performance.
- *In the second scenario, we will show how TQP can compile and execute TPC-H queries on different backends* (Section 3.2). We will use an Azure VM equipped with an NVIDIA GPU device, and show the query performance over CPU, GPU, as well as browser execution on Web Assembly through ORT.

* Equal contribution.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554853

[1]Pronounced 'Teacup'.

- *In the third scenario, we will show how TQP can both express and end-to-end accelerate prediction queries* containing an ML model embedded into a SQL query (Section 3.3). Here we will show: (1) how we extended the Spark SQL syntax with a PREDICT keyword allowing to execute, within SQL statements, both traditional ML (e.g., scikit-learn [16]) models and pre-trained neural networks; and (2) how relational operators and ML models are compiled into a unique tensor program that can be end-to-end executed on a GPU.

All the scenarios will be presented as a Python Notebook. Audience will be allowed to modify the queries and observe the performance tradeoffs; modify the ML models (Scenario 3), and interact with the UI (Scenario 1).

2 SYSTEM OVERVIEW

In this section, we summarize TQP's data representation (Section 2.1), and query compilation and execution (Section 2.2) previously introduced in [8]. Finally, we report some experimental number on two TPC-H queries (Section 2.3). We refer readers to [8] for more details on the system design, implementation, and algorithms mapping relational operators into tensor programs.

2.1 Data Representation

TQP internally represents tabular data in a columnar format. Each column of a table is represented as a $(n \times m)$ tensor, where n is the input number of rows, and m is the maximum length needed to store the column values. *Numerical* columns are represented as $(n \times 1)$ tensors, i.e., a single vector column is enough to represent the numerical values. *Date* columns are also represented as an $(n{\times}1)$ numeric tensors: i.e., the UNIX Epoch in nanoseconds. Finally, *string* columns are harder to manipulate efficiently using tensors. Currently, TQP represents string columns using a $(n \times m)$ numeric tensor, where m is the maximum character length of any value for that column. To store a string value in the tensor, TQP right-pads it with 0s if its length is less than m. Currently, TQP only supports UTF-8 encoded string values. TQP automatically transform input data into the tensor format. Data transformation is in general zero-copy, except date and strings columns that require data conversion.

2.2 Query Compilation and Execution

The TQP workflow for running a query is composed of two phases: in the *compilation* phase, the input query is transformed into an executable program; in the *execution* phase, the program is run.

Compilation. TQP's compilation phase is composed of 4 main layers: (1) the *parsing layer* converts the input SQL statements into an internal *Intermediate Representation (IR) plan* representing the input query's physical plan. The query physical plan is generated within TQP by an external frontend database system. (2) the *optimization layer* implements a rule-based optimizer providing IR-to-IR transformations; (3) the *planning layer* translates the IR plan generated in the previous layer into an *operator plan* where each operator in the IR is mapped into a tensor program implementation; finally (4) the *execution layer* generates an *Executor* from the operator plan. The Executor is the program that will be triggered at runtime on the selected target backend TCR and hardware.

In its current implementation, TQP uses vanilla PyTorch as implementation target of tensor programs in the planning layer. PyTorch programs are then lowered into different targets in the execution layer. Currently, PyTorch, TorchScript, ONNX, and TVM are the supported lower-level targets. On the frontend side, TQP currently support only Apache Spark, but we are planning to add support also for other databases. Since TQP translates physical query plans into an IR, the architecture decouples the physical plan specification from the other layers, therefore allowing to plug different frontends. TQP implements novel algorithms for expressing relational operators into tensor programs.

Execution. TQP first converts tabular data into the tensor format (as described in Section 2.1), and then the converted data is fed into the Executor program for generating the query result.

2.3 Performance Evaluation

We report a performance evaluation comparing TQP executing TPC-H queries 6 and 14 on CPU, GPU and web browser, versus Apache Spark (on CPU) at scale factor 1. These two queries will also be used through the demo. Note that the queries, albeit simple, are not trivial: query 6 contains four filters (over different columns data types), and aggregation. Query 14 contains a join, as well as aggregation over a CASE expression containing a LIKE statement. For CPU and GPU, we instrument TQP to use the TorchScript backend. For the web backend, we internally convert the queries in ONNX and run them on ORT with Web Assembly support [11] in a JavaScript environment.

We use an Azure NC6 v2 machine equipped with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA P100 GPU (with 16 GB of memory). We run both TQP-CPU and Spark over all cores. We report the median of the execution time over 5 runs, after 5 warmup runs. The web backend runs on our personal laptop (a Surface Book 3) to mirror a common client-facing scenario. As shown in Figure 1, TQP on CPU is around 3× faster than Apache Spark on both queries, while GPU execution is 20×, and 6× faster, respectively. As expected, the web execution is quite slow. Besides the weaker computation power on the personal laptop, the performance of TQP on the web browser is depending on the efficiency of ORT, Web Assembly, and the JavaScript environment. Nevertheless, this last result shows the flexibility of TQP, and its ability to leverage available open source ML tools.

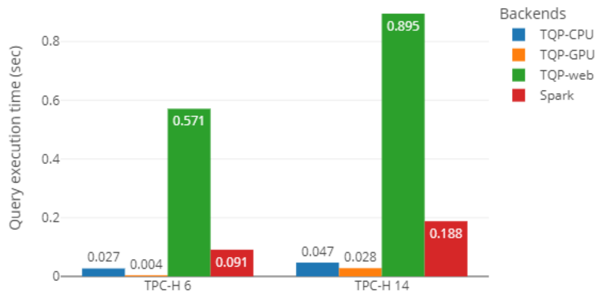
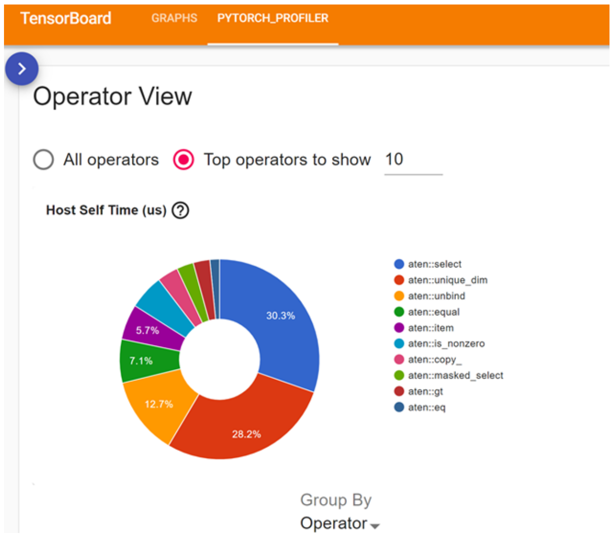


Figure 1: Query execution times for TPC-H 6 and 14 on Spark, and TQP on CPU, GPU, and web browser.

3 DEMONSTRATION

In this Section we will introduce the three scenarios the TQP demo consists of. Each scenario uses a Python Notebook run either locally on a laptop, or on an Azure VM with similar characteristics as the one described in Section 2.3. In each of the three scenarios, we will



Top operators to show: 10

Figure 2: Runtime breakdown of the top operators for a selected query. The view is automatically generated within TQP by the Pytorch Profiler, and visualized on TensorBoard.

allow audience to interact with the Python Notebook: for example, by editing the queries taken as input by TQP, by changing the backend/device configurations, or by exploring the UI.

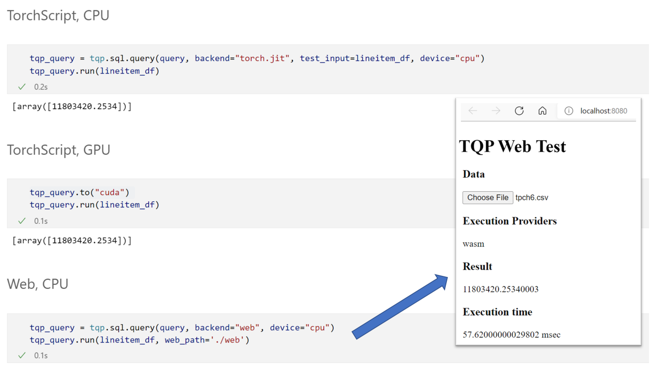
3.1 Scenario 1: Integration with DS Tools

TQP directly integrates with popular DS tools to offer a familiar developer experience. For instance, TQP is `pip`-installable, it is fully implemented in Python, and it only depends on well-known, enterprise-grade libraries. Users on TQP leverages Apache Spark (`pyspark`) to author their queries, and Pandas [13], Numpy [18] and Apache Arrow [3] for data ingestion. Finally, generating tensor programs for query execution opens the door to utilizing ML tools for profiling relational operations. In the first scenario, we will go over: (1) the integration with Apache Spark for plan generation; (2) integration with Pandas for data ingestion; and finally (3) integration with Tensorboard [1] for query profiling.

In this scenario, the audience will guided through the following steps: (1) we will show how TQP can be easily `pip`-installed and imported within a notebook; then (2) we will import the LINEITEM dataset scale factor 1 into the notebook as a Pandas dataframe; (3) we will show how the selected queries can be compiled in TQP, and executed over the input dataframe; finally, (4) we will re-execute the query with the profiler activated and investigate runtime breakdowns of the query, execution trace, and memory utilization of the CPU/CUDA kernels in Tensorboard. Figure 2 highlights one of the runtime charts for TPC-H query 6. Moreover, leveraging TensorBoard's out-of-the-box capabilities also helps graphically visualize the generated tensor program of a query, as depicted in Figure 4 for the query described in Scenario 3.

3.2 Scenario 2: Execution on Different Backends

Extensive efforts have been made by the ML community to enable TCRs' compatibility with different software and hardware backends. For instance, PyTorch has full support for CPU and GPU. Furthermore, various ML formats like TorchScript and ONNX have


TorchScript, CPU

```
tqp_query = tqp.sql.query(query, backend="torch.jit", test_input=lineitem_df, device="cpu")
tqp_query.run(lineitem_df)
✓ 0.2s
[array([11803420.2534])]
```

TorchScript, GPU

```
tqp_query.to("cuda")
tqp_query.run(lineitem_df)
✓ 0.1s
[array([11803420.2534])]
```

Web, CPU

```
tqp_query = tqp.sql.query(query, backend="web", device="cpu")
tqp_query.run(lineitem_df, web_path='./web')
✓ 0.1s
```

localhost:8080

TQP Web Test

Data

Choose File tpch6.csv

Execution Providers

wasm

Result

11803420.25340003

Execution time

57.62000000029802 msec

Figure 3: TPC-H query 6 executing in TQP on CPU and GPU devices using TorchScript (`torch.jit`) and over the `web` backend. Switching between different backends and hardware devices in TQP only needs one line of code change.

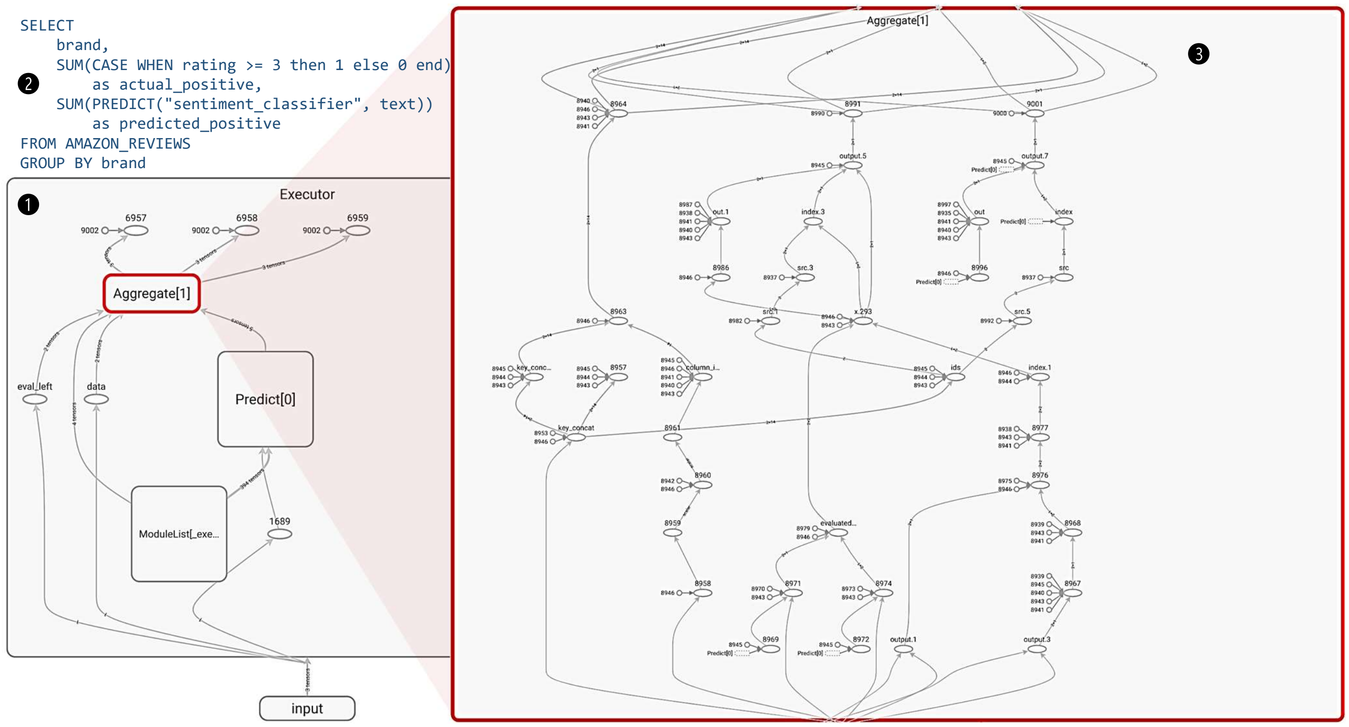
been developed to improve the inference performance through optimizations, as well as enabling the deployment of models in non-Python environments such as edge applications or web browsers.

To showcase how relational queries can leverage the above features, in this scenario we will demo how TQP can compile and execute the two selected queries on: (1) the CPU using the TorchScript backend (`torch.jit`); (2) switching to a GPU device; and (3) on CPU on a web browser using the `web` backend. [2] The workflow for this scenario is as follows: (1) initially we import all the required libraries; then (2) we import the LINEITEM dataset scale factor 1 into the notebook as a Pandas dataframe; and (3) we show the TPC-H 6 and 14 queries. Then, as presented in Figure 3 for query 6, (4) we will show how TQP compiles the queries into tensor programs, and how we can easily change the target device and the backend. Finally, (5) we will run the input data through the compiled queries (or load the data into the web browser for the `web` backend), and show how all of them generate the same correct result. We also plan to compare the performance of TQP with Spark over the two TPC-H queries. Figure 1 shows the expected results.

3.3 Scenario 3: Prediction Queries

Many past works have explored how to run ML workloads on relational engines (e.g., [2, 14]). Doing so enables in-database ML scenarios with benefits such as improved performance by minimizing data movement, or reducing engineering effort by eliminating the need of a dedicated service for model predictions. This latter feature is already available in commercial database systems. For example, since 2019, Microsoft SQL Server provides the ability to run in-database ML predictions through the PREDICT keyword. This feature allows users to combine, in a single SQL query, model predictions and standard relational operations. However, today such in-database ML integrations are typically done by spawning separate runtimes for relational and ML computations [14] (e.g., ORT in the SQL Server example). Given that TQP unifies relational and ML runtimes, embedding ML into relational queries is even more natural. To showcase

[2]Note that we could also run queries on the web backend targeting the GPU device through WebGL. However, the current implementation of ORT for WebGL does not cover all the ONNX operators, and hence execution currently fallback to CPU.
[3]The Executor graph used in Figure 4 can be accessed at https://tensorboard.dev/experiment/3mxfryX2QIaXWLVHlHkdIw/#graphs.



```
SELECT
    brand,
    SUM(CASE WHEN rating >= 3 then 1 else 0 end)
        as actual_positive,
    SUM(PREDICT("sentiment_classifier", text))
        as predicted_positive
FROM AMAZON_REVIEWS
GROUP BY brand
```

Figure 4: Executor graph for a prediction query (❶). The query (❷) combines an ML model, with a group-by-aggregate statement embedding a CASE expression. The query is used to predict the sentiment for each review, and compares the predictions with the user-provided ratings. The query-graph is interactive and the audience can further double-click on the various components to visualize how each operator is implemented as tensor operations (❸ shows a zoom-in for the aggregation operator). [3]

this integration, in the third scenario we will describe our implementation of Microsoft SQL Server's PREDICT keyword in TQP. Since TQP uses PyTorch as its default TCR backend, TQP natively supports predictions over any models built using PyTorch. Furthermore, since TQP integrates and expands HUMMINGBIRD [12], predictions over traditional ML models (e.g., created by libraries such as scikit-learn [16]) are supported as well. Finally, hybrid ML-SQL queries too profit from all the features described in the previous scenarios, e.g., end-to-end acceleration on GPU and integration with DS tools such as TensorBoard.

In this demo scenario, we will guide the audience through the creation and execution of predictive queries for two tasks: (1) Sentiment Classification on the Amazon Product Reviews Dataset [5]; and (2) Regression on the Iris Dataset [7]. The audience will be able to try a variety of models ranging from state-of-the-art, pre-trained transformers models from the HuggingFace Transformers library [19], to traditional ML models such as those available in the scikit-learn library. We will also show how to combine these models with relational operations such as filters or aggregates in a single SQL query. Figure 4 shows an example of such a query, and its execution graph visualized in the TensorBoard tool.

ACKNOWLEDGMENTS

We would like to thank Magdalena Balazinska, Soundar Srinivasan, Konstantinos Karanasos, Jesús Camacho-Rodrigues, Carlo Curino, and Raghu Ramakrishnan for their insightful feedback and support.

REFERENCES

[1] Martín Abadi and et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016).

[2] Ashvin Agrawal and et al. 2019. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. *arXiv e-prints* (2019). arXiv:1909.00084

[3] Apache Arrow. 2022. *Apache Arrow*. https://arrow.apache.org/docs/index.html

[4] BlazingSQL. 2020. . https://blazingsql.com/

[5] Datafiniti. 2019. *Consumer Reviews of Amazon Products*. https://www.kaggle.com/datafiniti/consumer-reviews-of-amazon-products

[6] Jeffrey Dean. 2019. The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design. arXiv:1911.05289 [cs.LG]

[7] Ronald A Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of eugenics* 7, 2 (1936), 179–188.

[8] Dong He and et al. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 12 (2022).

[9] Dimitrios Koutsoukos and et al. 2021. Tensors: An abstraction for general data processing. *Proc. VLDB Endow.* 14, 10 (2021), 1797–1804.

[10] Microsoft. 2021. *PREDICT in T-SQL*. https://docs.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-ver15

[11] Microsoft. 2022. *ORT Web*. https://cloudblogs.microsoft.com/opensource/onnx-runtime-web-running-your-machine-learning-model-in-browser/

[12] Supun Nakandala and et al. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *OSDI*. 899–917.

[13] The pandas development team. 2020. *pandas-dev/pandas: Pandas*.

[14] Kwanghyun Park and et al. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *SIGMOD '22*. 587–601.

[15] Adam Paszke and et al. 2017. Automatic differentiation in PyTorch. In *NIPS*.

[16] Fabian Pedregosa and et al. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

[17] Statista. 2022. *AI hardware market revenues*. https://www.statista.com/statistics/1003890/worldwide-artificial-intelligence-hardware-market-revenues/

[18] S. van der Walt and et a. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering* 13, 2 (2011), 22–30.

[19] Thomas Wolf and et al. 2020. Transformers: State-of-the-art natural language processing. In *EMNLP*. 38–45.

[20] Matei Zaharia and et al. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.