Benchmarking Machine Learning Software and Hardware for Quantitative Economics

Victor Duarte
UIUC

Diogo Duarte
FIU

Julia Fonseca
UIUC

Alexis Montecinos
Suffolk

June 2019

Abstract

We investigate the performance of machine learning software and hardware for quantitative economics. We show that the use of machine learning software and hardware can significantly reduce computational time in compute-intensive tasks. Using a sovereign default model and the Least Squares Monte Carlo option pricing algorithm as benchmarks, we show that specialized hardware and software speeds up calculations by up to four orders of magnitude when compared to programs written in popular high-level programming languages, such as MATLAB, Julia, Python/Numpy, and R, and high-performing low-level languages such as C++.

1 Introduction

In the last ten years, machine learning revolutionized many fields of research, from image recognition (Krizhevsky et al. 2012), to machine translation (Wu et al. 2016), to intertemporal optimization (Mnih et al. 2015). These breakthroughs were possible due to major advances in computer hardware, software, and methods.

Importantly, advances in each of these fronts tend to spur advances in the others. For instance, the development of new computer chips reduces the run time of experiments, which allows researchers to explore more ideas, an essential part of developing new methods. New methods and powerful computers make machine learning attractive to more researchers, increasing the pool of software developers who contribute to open-source projects. Better software leads to lower barriers to entry into machine learning, while a larger pool of potential customers provides economic incentives for hardware development. This positive feedback loop helped create a rich ecosystem of software and hardware that has been largely untapped by researchers in quantitative economics.

In this paper, we argue that research in finance and economics could benefit from this technological spillover. Specifically, we show that the adoption of machine learning software and hardware can accelerate standard operations in computational economics by up to four orders of magnitude, compared to popular programming languages—such as MATLAB, Python/Numpy, Julia, C++, and R—running on ordinary computers. We use two benchmark models, a strategic sovereign default model (Arellano 2008) and the Least Squares Monte Carlo (LSMC) method of pricing American options (Longstaff and Schwartz 2001). The former is solved using a standard value iteration algorithm, and therefore representative of many dynamic programming problems studied in economics. The key elements of (Longstaff and Schwartz 2001), simulation and regression, are also common ingredients in many tasks in economics and are tasks machine learning frameworks are designed to excel in.

Our paper is closest to Aruoba and Fernández-Villaverde (2015), who compare different programming languages commonly used in economics by benchmarking the solution of a real business cycle model using a value iteration algorithm. The authors show that the choice of tool can make a significant difference in both execution and development time. Specifically, the authors show that compiled languages, such as C++ and Fortran, can run hundreds of times faster than scripting languages, like Python or MATLAB for a typical economic application. At the same time, the authors point out that compiled languages are more complex and therefore harder to master, demanding more development time. We show that machine learning software can potentially eliminate this trade-off.

This paper is also related to computational economics literature that explores modern hardware to accelerate computation in quantitative economics. Aldrich et al. (2011) show that the use of Graphics Processing Units (GPUs) speed up value iteration by up to 200 times, while Fernández-Villaverde and Valencia (2018) present a guide on parallelization for both GPUs and ordinary Central Processing Units (CPUs). This paper adds to this existing work by benchmarking machine learning software and also a new type of hardware designed specifically for machine learning applications, called Tensor Processing Units (TPUs).

The second objective of the paper is to lay out a blueprint for future work on quantitative economics. To the best of our knowledge, this is the first paper to provide an assessment of machine learning software and hardware for computational economics.[1] By providing replication files in our Github repository, we hope to stimulate the adoption of these tools by researchers in quantitative economics.[2]

[1]Note that this is not the first paper to explore the potential of machine learning *algorithms* for economic applications. To name a few recent studies, Gu et al. (2018), Athey et al. (2019), Fernandez-Villaverde (2019) and Duarte (2019) employ machine learning algorithms for either empirical or computational work. With the exception of the latter, these works do not use specialized software (such as TensorFlow or PyTorch) or hardware (such as GPUs or TPUs). And while Duarte (2019) uses both TensorFlow and GPUs, the study does not benchmark these tools against traditional software and hardware.

[2]Code can be found at github.com/vduarte/benchmarkingML.

The rest of the paper is organized as follows. Section 2 describes the software and hardware, focusing on recent advances and on how these new tools can be used to accelerate compute-intensive tasks in quantitative economics. Section 3 discusses the first benchmarking exercise, the sovereign default model of Arellano (2008). Section 4 details the second benchmarking exercise, using the LSMC method of pricing American options. Section 5 concludes.

2 Software and Hardware

It is well understood that there is a trade-off between compiled programming languages, such as Fortran and C++, and scripting languages, such as Python, Julia, R, and MATLAB (Aruoba and Fernández-Villaverde 2015). Compiled languages have relatively fast execution speed but higher development time due to their complexity. As computers become more powerful, researchers tend to favor development speed.

As a consequence, most of the research in computational economics is done in either MATLAB, Julia, R, or Python.[3] Accordingly, we benchmark specialized machine learning software and hardware against these four high-level languages. We also include C++ in our benchmarking exercise, which is perhaps the most powerful among low-level languages and is one of the most popular.[4] We benchmark this software in a MacBook Pro laptop, and also use specialized hardware which we describe below.[5]

Among the traditional programming languages that we select for our benchmarking exercise, Julia and Python are perhaps the least well known within economics. Julia is a modern, flexible language, designed for high performance and suitable for scientific

[3]See, for instance, Coleman et al. (2018) for a recent comparison of programming languages in economics.

[4] C++ is ranked third in the May 2019 edition TIOBE Index of programming language popularity, behind Java and C.

[5]The full specification of the laptop is Mac OS 10.12.6 Sierra, 2.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3, and 256 GB PCIe-based storage. The specification for all hardware used in our benchmarking exercise can be found in Table A.1.

and numerical computing. Its syntax follows MATLAB's closely, and it promises superior performance on a number of tasks. Python is a general-purpose programming language that has become increasingly popular among academics and practitioners due to its syntax simplicity and flexibility. Most numerical work using Python relies on the Numpy library, a large collection of classes and functions for linear algebra and numerical computing. Throughout the text, we refer to the combination of Python and the Numpy library as Python/Numpy.

For some computational-intensive applications in economics, such as dynamic programming, structural estimation, option pricing, and Bayesian inference, computing power is still one of the most central bottlenecks. Even with the increased power of modern computers, the need for higher performance often drives researchers in these fields away from scripting languages and towards compiled languages.[6]

Interestingly, this trade-off is not confined to academia. Several tech companies, which need to maximize software performance while minimizing development time, experience the same challenge routinely. In the last three years, some of the largest tech companies addressed that problem by developing and open sourcing machine learning frameworks. Google open sourced TensorFlow in 2015, followed by Microsoft with its Cognitive Toolkit (CNTK) in 2016 and Facebook with PyTorch in 2016.

We benchmark two of these frameworks, TensorFlow and PyTorch, which have sparked considerable interest in recent years. Figure 1 shows the number of active developers for the top five machine learning frameworks in 2019. As of 2019, TensorFlow had nearly 2,000 active developers, followed by PyTorch with around 1,000 developers. These frameworks feature a Python front end for fast development and experimentation and highly optimized kernels written in C++ and CUDA for number crunching. Therefore, the scripting language serves as the glue that binds together pieces of code written in low-level languages.

[6]See, for instance, Guvenen (2009), Ju and Miao (2012), and Kaplan and Violante (2014).

Crucially, both PyTorch and TensorFlow can completely eliminate the overhead stemming from data transfers between the system memory and the CPU or GPU. While MATLAB and Python/Numpy also rely on fast C/C++ kernels to execute linear algebra operations, these languages return intermediary results to the interpreter after the execution of each operation. The composition of many such operations slows down the program when the computation involves large arrays as memory transfers become costly.[7] PyTorch and Tensorflow, on the other hand, create a computation graph of the entire program and execute it on the CPU or GPU without ever transfering intermediary results from the C++ or CUDA kernels to the interpreter.[8]

Another distinctive feature of these modern numerical libraries is that they make efficient use of available hardware. Modern machine learning frameworks, for instance, automatically offload compute-intensive parts of code to a GPU if such a unit is available. GPUs are specialized hardware designed to excel in massively parallel linear algebra operations, originally developed to accelerate the processing of video game graphics.

In an influential study, Raina et al. (2009) show that GPUs can be used to speed up machine learning applications by up to two orders of magnitude. Aldrich et al. (2011) is one of the first studies to show that the same gains can be obtained in economic applications. Aldrich et al. (2011) also note the challenges associated with GPU programming, such as memory management, but predict that these considerations would eventually become irrelevant for the average user. Much like the authors predicted, a researcher today does not need to have specialized knowledge of GPU programming, as machine learning software automatically makes efficient use of these units. We leverage this feature of specialized software by running the exact same TensorFlow and PyTorch code on a desktop computer with a consumer-grade

[7]See, for instance, Alted (2010) and Bergstra et al. (2010).

[8]TensorFlow executes in this so called "graph mode" by default, while PyTorch requires the use of JIT decorators.

GPU.[9]

The adoption of modern numerical frameworks also opens up the possibility of using specialized hardware that is not otherwise available, such as Google's Tensor Processing Unit (TPU). TPUs are Application-Specific Integrated Circuits (ASICs) designed to execute specific tasks—such as multiplying large matrices—as fast as possible. Google's TPU powers many of Google's compute-intensive services, such as Gmail and Google Translate. Currently, TPUs are only available through cloud computing via Google's Compute Engine or via Colab, a cloud service intended for researchers.[10] We benchmark Colab, which is available free of charge, and use both its TPU and GPU capabilities.[11]

3 Experiment 1: Sovereign Default Model

We start by benchmarking machine learning frameworks using the stochastic general equilibrium with endogenous default risk of Arellano (2008). We choose this model for its popularity and because we can solve it through value function iteration, a widely used solution method for a large class of models.

3.1 Model Description

A benevolent government chooses a consumption plan $\{c_t\}$ and government asset holdings of one-period discount bonds $\{B_{t+1}\}$ to maximize its citizens expected discounted utility

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\gamma}}{1-\gamma},$$

[9] The full specification for this and all hardware used can be found in Table A.1.

[10] Colab can be accessed at https://colab.research.google.com/

[11] See footnote 9.

where $\beta \in (0,1)$ is the time preference parameter and $\gamma > 0$ the relative risk aversion parameter. In addition, households receive a stochastic stream of tradable good $\{y_t\}$.

At each period, the government can choose to default on its obligation or to repay its debt. If the government decides to default on its obligation, it is immediately excluded from international financial markets and households consume output y_t^{def}, which is lower during financial autarky

$$c_t = y_t^{def} \equiv \begin{cases} y_t & \text{if } y_t < \hat{y} \\ \hat{y} & \text{if } y_t \geq \hat{y} \end{cases},$$

for some exogenous threshold $\hat{y}$. If the government decides to pay its debt, it can access international financial markets to buy one-period bonds B_{t+1} at a competitive endogenous price $q(B_{t+1}, y_t)$. The resource constraint is given by

$$c_t = y_t + B_t - q(B_{t+1}, y_t)B_{t+1}.$$

Lenders are risk-neutral and lend at the constant rate $r > 0$. The bond price then given by

$$q(B_{t+1}, y_t) = \frac{1 - \delta(B_{t+1}, y_t)}{1 + r},$$

where $\delta(B_{t+1}, y_t)$ is the endogenous probability of default.

Denoting the government's value function by $\nu^o(B, y)$, we can write this problem recursively as

$$\nu^o(B, y) = \max_{\{c,d\}} \left\{ \nu^c(B, y), \nu^d(y) \right\},$$

where

$$\nu^d(y) = u(y^{def}) + \beta \left[\theta \mathbb{E}\nu^o(0, y') + (1-\theta)\mathbb{E}\nu^d(y')\right],$$
$$\nu^c(B, y) = \max_{B'} \left\{u(y - q(B', y)B' + B) + \beta \mathbb{E}\nu^o(B', y')\right\}.$$

The parameter θ represents the probability of the government regaining access to international credit markets. Finally, the probability of default is given by

$$\delta(B', y) = \mathbb{E}\left[\nu^d(y') > \nu^c(B', y')\right]$$

3.2 Results

Results for this benchmarking exercise can be found in Table 1. Each panel of Table 1 refers to different hardware and each row contains the average run time in milliseconds of one iteration of the solution algorithm for a different software/hardware combination.[12] In each column, we vary the grid size of government bond holdings. Note that we implement and run the exact same algorithm in each programming language, so the solutions across different platforms are identical up to machine precision.

In the first panel of Table 1, we benchmark C++, Julia, Matlab, Python/Numpy, PyTorch, TensorFlow, and R, in a MacBook Pro laptop. Our first result is that TensorFlow and PyTorch outperform scripting languages, and the difference in performance increases with grid size. For the smallest grid size we consider, with 151 points, TensorFlow is 4.5 times faster than MATLAB and more than 8 times faster than Julia. For the finest grid, with 1551 points, TensorFlow is around 42 times faster than MATLAB and 25 times faster than Julia.

[12] We average the run time across 500 iterations of the solution algorithm. The full configuration of all hardware used can be found in Table A.1. Software versions are detailed in Table A.2.

Secondly, we find that TensorFlow outperforms our C++ implementation. TensorFlow ranges from 1.5 to 2 times faster than C++, depending on grid size. This result suggests that machine learning software may do away with the trade-off between software performance and ease of development for typical economic applications. Note that TensorFlow runs C++ kernels on its back end, so it is not entirely surprising that it achieves similar performance.[13] It is slightly more surprising that it *outperforms* C++, but that is potentially attributable to our C++ implementation being less optimized than that of TensorFlow, which was programmed by highly skilled software engineers at Google. Our main takeaway from this result is that, for the average researcher, machine learning frameworks offer C++ performance without the high cost of development usually associated with low-level languages.

So far we have benchmarked programming languages on a personal laptop. But one of the key features of machine learning software is its seamless integration with specialized hardware, which provides striking gains in performance. In the second panel of Table 1, we show results obtained by running the same TensorFlow and PyTorch code on a desktop computer with a consumer-grade GPU.[14] The gains in performance range from 2 orders of magnitude (e.g. TensorFlow vs. MATLAB for a grid of 151 points) to 4 orders of magnitude (e.g. TensorFlow vs. Python/Numpy for a grid of 1551 points). In particular, the performance of the same software with or without the use of a GPU is as high as 3 orders of magnitude (for TensorFlow with a grid of 1551 points). This is a sizable performance gain which comes at no additional development cost to the researcher.

Finally, we benchmark the performance of machine learning software in Google's Colab, a cloud service available free of charge, and show results of this exercise in the last two panels of Table 1. We first benchmark Colab's GPU capabilities and find that performance is

[13] TensorFlow runs C++ kernels on its back end in the absence of GPUs, such as in the case we describe. In the presence of GPUs the back end of both TensorFlow and PyTorch is CUDA.

[14] PyTorch requires one additional line of code to set the back end kernel to CUDA for use with a GPU and TensorFlow requires two additional lines of code to initialize use with a TPU. These additional commands do not vary with the application or algorithm and the code is otherwise identical.

comparable, if slightly inferior, to what we obtain on a desktop computer with a high-end consumer-grade GPU. We then benchmark Colab's TPU, which in this application is slower than its GPU by an order of magnitude for coarser grids but comparable for finer grids.[15]

4 Experiment 2: American-Style Option Pricing

Next, we turn to the Least Squares Monte Carlo (LSMC) method of Longstaff and Schwartz (2001) for pricing American options. We choose this as a benchmark because the two key components of the method, simulation and regression, are common ingredients in most compute-intensive tasks in economics and finance.

4.1 Model Description

The discrete time approximation of an American option is the so-called Bermuda option, where the holder has the option to exercise the contract in a finite number of dates $0 < t_1 < t_2 < ... < t_{K-1} < t_K = T$.

Under the assumption of no arbitrage, the put option price V_0 is the solution of the following optimal stopping problem

$$V_0 = \sup_{\tau \in \mathcal{T}_0} \mathbb{E}^{\mathbb{Q}} \left[f(\tau, S_\tau) | \mathcal{F}_0 \right],$$

where S_τ is the underlying asset, $f(\cdot, \cdot)$ is the discounted payoff function, the expectation is taken under the risk-neutral measure $\mathbb{Q}$, $\mathcal{F}_0$ represents the information set at the initial time, and the stopping time τ belongs to the class of all $\{0, ..., T\}$-valued stopping times, represented by $\mathcal{T}_0$.

[15] As of now, only TensorFlow is compatible with the use of TPUs, although support for PyTorch and Julia is under development.

At the exercise date t_i, the continuation value q_{t_i} satisfies

$$q_{t_i} = \sup_{\tau \in \mathcal{T}_{t_i}} \mathbb{E}^{\mathbb{Q}} \left[f(\tau, S_\tau) | \mathcal{F}_{t_i} \right], \tag{1}$$

where $\mathcal{F}_{t_i}$ is the information set at time t_i and $\mathcal{T}_{t_i}$ is the class of all $\{t_{i+1}, ..., T\}$-valued stopping times. The continuation values are determined by the recursive equations

$$q_{t_i} = \mathbb{E}^{\mathbb{Q}} \left[\max \left\{ f(t_{i+1}, S_{t_{i+1}}), q_{t_{i+1}} \right\} | \mathcal{F}_{t_i} \right], \; i \in \{0, 1, ..., K-1\},$$

with terminal condition $q_T = 0$.

LSMC uses a linear combination of orthonormal basis functions to approximate the expectation in (1). Starting at time t_{K-1}, the continuation value is approximated by

$$q_{t_{K-1}} = \sum_{j=0}^{M} a_j p_j(S_{t_{K-1}}), \tag{2}$$

where $a_j \in \mathbb{R}$ are the regression coefficients, $p_j(\cdot)$ are the polynomial basis, and M represents the degree of the polynomial basis.

The coefficients are determined by solving the least squares problem of minimizing the distance between the approximate option price in equation (2) and realized payoffs one period ahead. To alleviate the problem of multicollinearity of the regressors, we solve the ordinary least squares problem with ridge regression using a L_2 penalty $\lambda = 100$, and repeat this procedure until the first exercise date.

4.2 Results

Results for this benchmarking exercise can be found in Table 2. Each panel of Table 2 refers to different hardware and each row shows the run time in milliseconds of the full solution algorithm for a different software/hardware combination.[16] In each column, we vary the order of the polynomial used to approximate the continuation value. As in the previous experiment, we implement and run the exact same algorithm in each programming language.

Our findings for this exercise are in line with those from the previous section. In the comparison of all languages in a MacBook Pro laptop, shown in the first panel of Table 2, we again find that TensorFlow and PyTorch outperform other software. For a basis order of 5, TensorFlow is nearly 8 times faster than MATLAB and 17 times faster than Julia. For a basis order of 25, the largest we report, TensorFlow is 5 times faster than MATLAB and 14 times faster than Julia.

Moreover, we again obtain considerable gains when running the same TensorFlow and PyTorch code on a desktop computer with a consumer-grade GPU, which are shown in the second panel of Table 2. Performance gains, in this case, are highest for a basis order of 25 and can exceed two orders of magnitude (e.g. TensorFlow vs. R). Finally, in the last two panels of Table 2, we benchmark Colab's GPU and TPU capabilities. Performance gains are again comparable to what we obtain with a desktop computer with a high-end, consumer-grade GPU, if slightly inferior.

[16] As before, the full configuration of all hardware used can be found in Table A.1 and software versions can be found in Table A.2.

5 Conclusion

In this paper, we investigate the performance of machine learning software and hardware for typical applications in economics and finance. Machine learning software is designed to excel in massively parallel tasks, such as simulation, regression, and matrix operations, which are staples of many algorithms in quantitative economics. Moreover, this software is designed to make efficient use of available specialized hardware, such as GPUs or TPUs.

We show that modern numerical frameworks can produce substantial performance gains without the added complexity of compiled languages. By making our replication files publicly available in our GitHub repository, we hope to facilitate the adoption of these tools by a wide range of researchers.

References

Aldrich, Eric M, Jesús Fernández-Villaverde, A Ronald Gallant, and Juan F Rubio-Ramírez. Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, 35(3):386–393, 2011.

Alted, Francesc. Why modern cpus are starving and what can be done about it. *Computing in Science & Engineering*, 12(2):68–71, 2010. doi: 10.1109/MCSE.2010.51. URL `https://aip.scitation.org/doi/abs/10.1109/MCSE.2010.51`.

Arellano, Cristina. Default risk and income fluctuations in emerging economies. *American Economic Review*, 98(3):690–712, 2008.

Aruoba, S Borağan, and Jesús Fernández-Villaverde. A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control*, 58:265–273, 2015.

Athey, Susan, Mohsen Bayati, Guido Imbens, and Zhaonan Qu. Ensemble Methods for Causal Effects in Panel Data Settings. *arXiv e-prints*, art. arXiv:1903.10079, Mar 2019.

Bergstra, James, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proceedings of the 9th Python in Science Conference*, pages 3–10, 2010.

Coleman, Chase, Spencer Lyon, Lilia Maliar, and Serguei Maliar. Matlab, python, julia: What to choose in economics? CEPR Discussion Papers 13210, C.E.P.R. Discussion Papers, 2018.

Duarte, Victor. Machine learning for continuous-time finance. Working paper, 2019.

Fernandez-Villaverde, Jesus. Financial frictions and the wealth distribution. Working paper, 2019.

Fernández-Villaverde, Jesús, and David Zarruk Valencia. A practical guide to parallelization in economics. Working paper, 2018.

Gu, Shihao, Bryan Kelly, and Dacheng Xiu. Empirical asset pricing via machine learning. 2018.

Guvenen, Fatih. A parsimonious macroeconomic model for asset pricing. *Econometrica*, 77 (6):1711–1750, 2009.

Ju, Nengjiu, and Jianjun Miao. Ambiguity, learning, and asset returns. *Econometrica*, 80(2): 559–591, 2012.

Kaplan, Greg, and Giovanni L. Violante. A model of the consumption response to fiscal stimulus payments. *Econometrica*, 82(4):1199–1239, 2014.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Pereira, F., C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

Longstaff, Francis A, and Eduardo S Schwartz. Valuing american options by simulation: a simple least-squares approach. *The Review of Financial Studies*, 14(1):113–147, 2001.

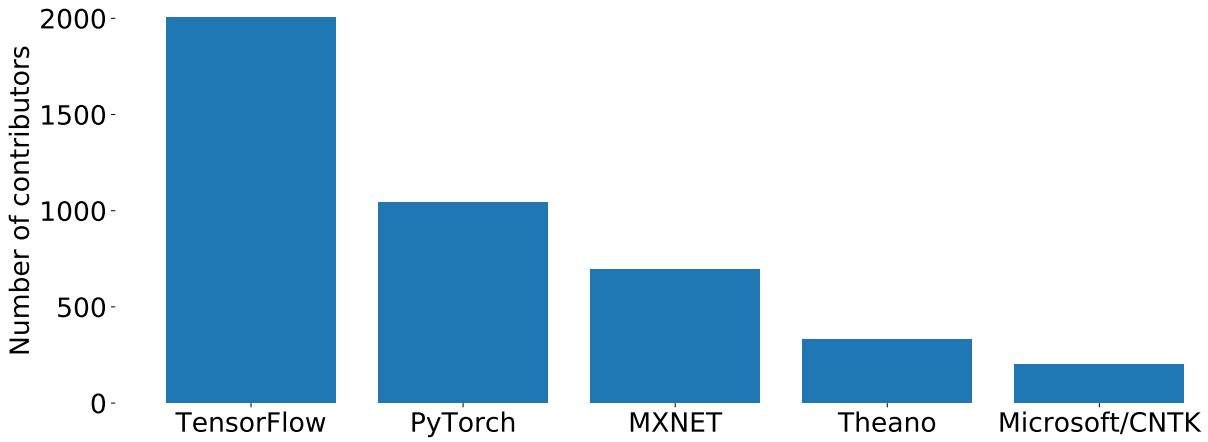
Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

Raina, Rajat, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on*

Machine Learning, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

Figure 1: GitHub Contributors



This figure shows the number of active GitHub contributors for each framework, according to data from GitHub.

Table 1: Performance Comparison for Sovereign Default Model

Hardware	**Software**	**Grid Size (for Bond Holdings)**							
		151	351	551	751	951	1151	1351	1551
Laptop	C++	37	178	578	1010	1674	2335	3158	4161
	Julia	169	725	1826	3370	5310	9188	28741	58269
	Matlab	91	318	792	1546	5215	23862	60801	98609
	Python/Numpy	133	667	1662	3068	11646	31228	51633	124027
	PyTorch	73	371	900	1648	2672	3969	5396	7445
	R	430	2237	5379	9917	15993	23726	33416	45791
	TensorFlow	20	117	291	533	859	1249	1752	2306
Desktop with GPU	PyTorch	0.32	1.40	3.63	7.37	12.02	16.14	19.37	20.67
	TensorFlow	0.42	0.79	1.25	1.75	2.50	3.44	4.79	6.18
Google Colab (GPU)	PyTorch	0.48	2.41	5.90	11.18	17.90	26.80	35.75	49.86
	TensorFlow	0.74	1.28	1.98	2.95	4.07	5.70	7.97	10.15
Google Colab (TPU)	TensorFlow	3.27	4.21	5.44	4.59	5.09	5.19	6.53	7.36

This table shows the average execution time (in milliseconds) of one iteration of the solution algorithm for the sovereign default model described in section 3.1. We average the run time across 500 iterations of the solution algorithm. Each row represents a combination of software and hardware and each column represents a grid size. Details on hardware configuration and software version can be found in Tables A.1 and A.2, respectively.

Table 2: Performance Comparison for LSMC Method for Option Pricing

Hardware	**Software**	**Basis Order**		
		5	10	25
Laptop	C++	920	1076	1864
	Julia	1610	1792	3056
	Matlab	718	843	1182
	Python/Numpy	826	986	2051
	PyTorch	262	333	497
	R	2546	3550	4157
	TensorFlow	95	121	218
Desktop with GPU	PyTorch	22	24	35
	TensorFlow	7	10	11
Google Colab (GPU)	PyTorch	42	51	70
	TensorFlow	11	14	16
Google Colab (TPU)	TensorFlow	20	23	31

This table shows the execution time (in milliseconds) of the solution algorithm for the LSMC method for option pricing described in section 4.1. Each row represents a combination of software and hardware and each column represents a basis order. Details on hardware configuration and software version can be found in Tables A.1 and A.2, respectively.

A Additional Tables

Table A.1: Hardware Configuration

	CPU	**RAM**	**Accelerator**	**OS**
Laptop	Intel dual-core i5 2.3 ghz	8GB	None	Windows 10
Desktop	Intel six-core i7 3.7ghz	16GB	NVIDIA RTX 2080 GPU	Ubuntu 18.04
Google Colab (GPU)	Intel Xeon 2.3ghz (2 cores)	14GB	NVIDIA Tesla T4	Ubuntu 18.04
Google Colab (TPU)	Intel Xeon 2.3ghz (2 cores)	14GB	TPU v2	Ubuntu 18.04

This table contains the configuration of all hardware utilized. While Google Colab's processor has multiple cores, the cloud allocates two cores per session.

Table A.2: Software Version

C++ Compiler	Julia	Matlab	Python/Numpy	PyTorch	R	TensorFlow
Visual Studio 2019	1.1.0	2019a	Anaconda 3.7	1.1	Microsoft R Open 3.5.1	1.13.1

This table contains the version of all software utilized. We use C++ with the Armadillo library version 9.400.