



Trends in Machine Learning

1. [Compute Trends Across Three eras of Machine Learning](#)
2. [Parameter counts in Machine Learning](#)
3. [Estimating training compute of Deep Learning models](#)
4. [What's the backward-forward FLOP ratio for Neural Networks?](#)
5. [How to measure FLOP/s for Neural Networks empirically?](#)
6. [Projecting compute trends in Machine Learning](#)
7. [Compute Trends — Comparison to OpenAI's AI and Compute](#)

Compute Trends Across Three eras of Machine Learning

Crossposted from the [AI Alignment Forum](https://arxiv.org/abs/2202.05924). May contain more technical jargon than usual.

<https://arxiv.org/abs/2202.05924>

What do you need to develop advanced Machine Learning systems? Leading companies don't know. But they are very interested in figuring it out. They dream of replacing all these pesky workers with reliable machines who take no leave and have no morale issues.

So when they heard that [throwing processing power at the problem might get you far along the way](#), they did not sit idly on their GPUs. But, how fast is their demand for compute growing? And is the progress regular?

Enter us. We have [obsessively analyzed](#) trends in the amount of compute spent training milestone Machine Learning models.

Our analysis shows that:

- **Before the Deep Learning era**, training compute approximately followed Moore's law, doubling every ≈ 20 months.
- The **Deep Learning era** starts somewhere between 2010 and 2012. After that, doubling time speeds up to ≈ 5 -6 months.
- Arguably, between 2015 and 2016 a separate **trend of large-scale models** emerged, with massive training runs sponsored by large corporations. During this trend, the amount of training compute is 2 to 3 orders of magnitude (OOMs) bigger than systems following the Deep Learning era trend. However, the growth of compute in large-scale models seems slower, with a doubling time of ≈ 10 months.

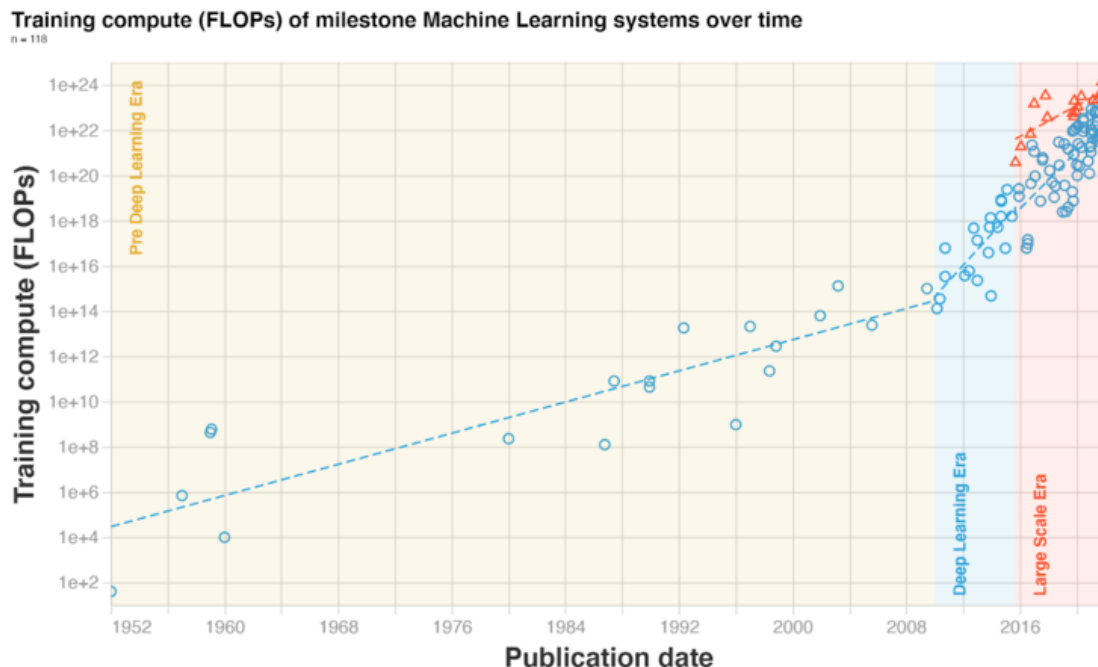


Figure 1: Trends in $n=118$ milestone Machine Learning systems between 1950 and 2022. We distinguish three eras. Note the change of slope circa 2010,

matching the advent of Deep Learning; and the emergence of a new large scale trend in late 2015.

Period	Data	Scale (start to end)	Slope	Doubling time
1952 to 2010	All models	3e+04 to 2e+14 FLOPs	0.2 OOMs/year	21.3 months
Pre Deep Learning Trend	($n = 19$)		[0.1; 0.2; 0.2]	[17.0; 21.2; 29.3]
2010 to 2022	Regular-scale models	7e+14 to 2e+18 FLOPs	0.6 OOMs/year	5.7 months
Deep Learning Trend	($n = 72$)		[0.4; 0.7; 0.9]	[4.3; 5.6; 9.0]
September 2015 to 2022	Large scale models	4e+21 to 8e+23 FLOPs	0.4 OOMs/year	9.9 months
Large-Scale Trend	($n = 16$)		[0.2; 0.4; 0.5]	[7.7; 10.1; 17.1]

Table 2: Summary of our main results. In 2010 the trend accelerated along the with the popularity of Deep Learning, and in late 2015 a new trend of large-scale models emerged.

Table 1. Doubling time of training compute across three eras of Machine Learning. The notation [low, median, high] denotes the quantiles 0.025, 0.5 and 0.975 of a confidence interval.

Not enough for you? Here are some fresh takeaways:

- Trends in compute are **slower than previously reported**! But they are **still ongoing**. I'd say slow and steady, but the rate of growth is blazingly fast, still doubling every 6 months. This probably means that you should double the timelines for all [previous analyses](#) that relied on *AI and Compute*'s previous result.
- We think the framing of the **three eras of ML** is very helpful! Remember, we are suggesting to split the history of ML into the **Pre-Deep Learning Era**, the **Deep Learning Era** and the **Large-Scale Era**. And we think this framing can help you make sense of what has happened in the last two decades of ML research.
- We have curated an awesome [public database of milestone ML models](#)! Please use it for your own analyses (don't forget to cite us!). If you want to play around with the data, we are maintaining an interactive visualization of it [here](#).

Compute is a strategic resource for developing advanced ML models. Better understanding the progress of our compute capabilities will help us better navigate the advent of transformative AI.

In the future, we will also be looking at the other key resource for training machine learning models: *data*. [Stay tuned for more!](#)

[Read the full paper now on the arXiv](#)

Parameter counts in Machine Learning

Crossposted from the [AI Alignment Forum](#). May contain more technical jargon than usual.

In short: we have compiled information about the date of development and trainable parameter counts of $n=139$ machine learning systems between 1952 and 2021. This is, as far as we know, the biggest public dataset of its kind. You can access our dataset [here](#), and the code to produce an interactive visualization is available [here](#).

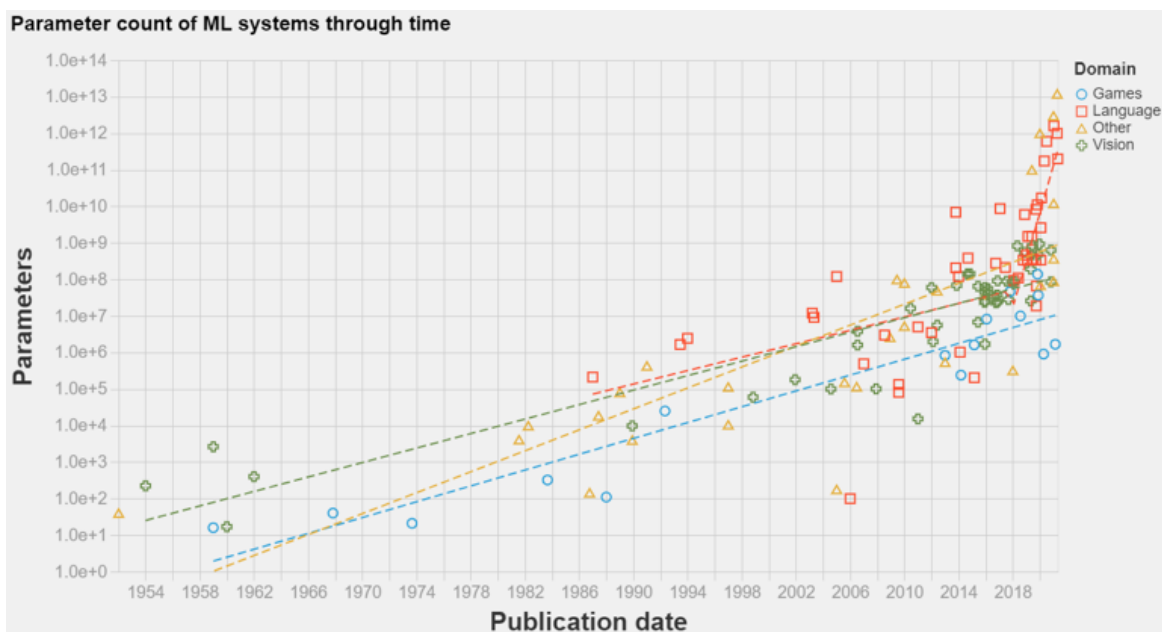
We chose to focus on parameter count because previous work indicates that it is an important variable for model performance [1], because it helps as a proxy of model complexity and because it is information usually readily available or easily estimable from descriptions of model architecture.

We hope our work will help AI researchers and forecasters understand one way in which models have become more complex over time, and ground their predictions of how the field will progress in the future. In particular, we hope this will help us tease apart how much of the progress in Machine Learning has been due to algorithmic improvements versus increases in model complexity.

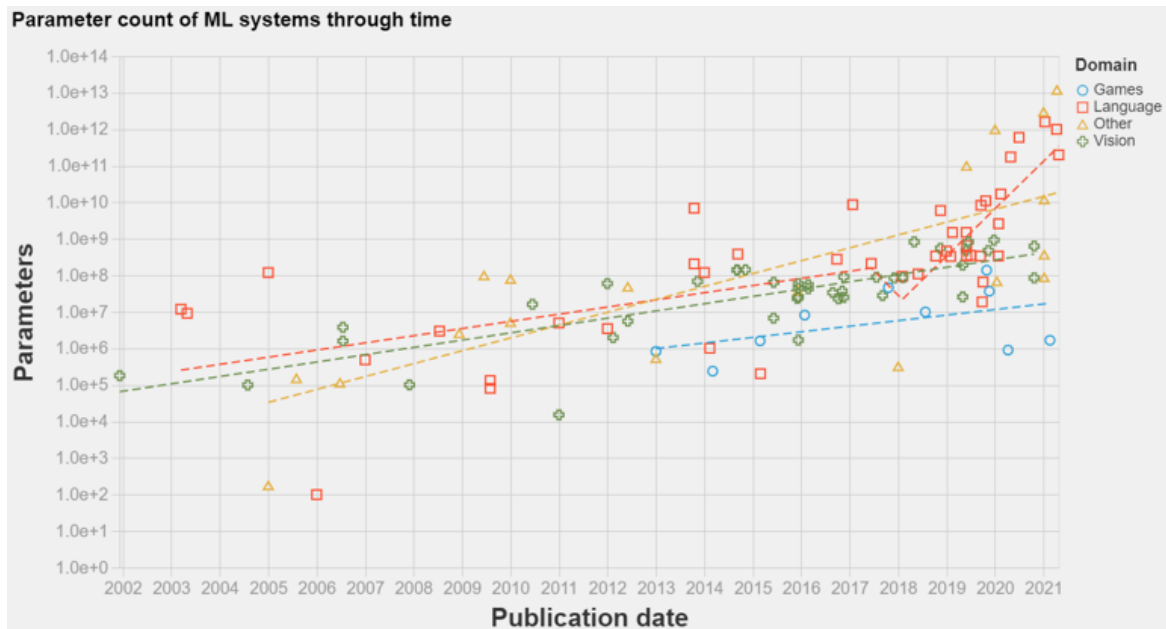
It is hard to draw firm conclusions from our biased and noisy dataset. Nevertheless, our work seems to give weak support to two hypotheses:

- There was no discontinuity in any domain in the trend of model size growth in 2011-2012. This suggests that the Deep Learning revolution was not due to an algorithmic improvement, but rather the point where the trend of improvement of Machine Learning methods caught up to the performance of other methods.
- In contrast, it seems there has been a discontinuity in model complexity for language models somewhere between 2016-2018. Returns to scale must have increased, and shifted the trajectory of growth from a doubling time of ~ 1.5 years to a doubling time of between 4 to 8 months.

The structure of this article is as follows. We first describe our dataset. We point out some weaknesses of our dataset. We expand on these and other insights. We raise some open questions. We finally discuss some next steps and invite collaboration.



Model size of popular new Machine Learning systems between 1954 and 2021. Includes n=139 datapoints. See expanded and interactive version of this graph [here](#).



Model size of popular new Machine Learning systems between 2000 and 2021. Includes n=114 datapoints. See expanded and interactive version of this graph [here](#).

Features of the dataset

- The dataset spans systems from 1952 to 2020, though we included far more information about recent systems (from 2010 onwards).
- The systems we include encompass many types, including neural networks, statistical models, support vector machines, bayesian networks and other more exotic architectures. However we mostly included systems of the neural network kind.
- The systems are from many domains and were trained to solve many tasks. However we mostly focused on systems trained to solve vision, language and gaming tasks.
- We relied on a subjective criteria of notability to decide which systems to include. Our decisions were informed by citation counts (papers with more than 1000 citations), external validation (papers that received some kind of paper of the year award or similar) and historical importance (papers that were cited by other work as seminal). The references to this post include some overviews we used as a starting point to curate our dataset [2-26].
- Several models have versions at multiple scales. Whenever we encountered this in their original publication, we recorded whichever was presented in the paper as the main one, or the largest presented version. Sometimes we recorded multiple versions when we felt it was warranted, e.g. when multiple different versions were trained to solve different tasks.

Caveats

- It is important to take into account that model size is hardly the most important parameter to understand the progress of ML systems. Other arguably more important indicators of non-algorithmic progress in ML systems include training compute and training dataset size [1].

- Model size as a metric of model complexity is hardly comparable across domains or even architectures. For example, a mixture-of-expert model can achieve higher parameter counts but invest far less compute into training each parameter.
- Our selection of systems is biased in many important ways. We are biased towards academic publications (since information on commercial systems is harder to come by). We include more information about recent systems. We tended to include information about papers where the parameter counts were readily available, in particular larger models that were developed to test the limits of how large a model can be. We are biased towards papers published in English. We mostly focused on systems on vision, language and gaming tasks, while we have comparatively fewer papers on e.g. speech recognition, recommender systems or self driving. Lastly, we are biased towards systems we personally found interesting or impressive.
- Recollecting the information was a time consuming exercise that required us to read through hundreds of technical papers to gather the parameter counts. It is quite likely we have made some mistakes.

Insights

- Unsurprisingly, there is an upward trend in model size. The trend seems exponential, and seems to have picked up its pace recently for language models. An eyeball estimate of the slope of progress suggests that the doubling rate was between 18 and 24 months from 2000 to 2016-2018 in all domains, and between 3 and 5 months from 2016-2018 onward in the language domain.
- The biggest models in terms of trainable parameters can be found in the language and recommender system domains. The biggest model we found was the 12 trillion parameter Deep Learning Recommender System from Facebook. We don't have enough data on recommender systems to ascertain whether recommender systems have been historically large in terms of trainable parameters.
- Language models have been historically bigger than in other domains. This was because of statistical models whose parameterization scales with vocabulary size (e.g. as in the Hiero Machine Translation System from 2005) and word embeddings that also scale with vocabulary size (e.g. as in Word2Vec from 2013).
- Arguably Deep Learning started to proliferate in computer vision before it reached language processing (both circa 2011-2013), however the parameter counts of the second far surpass those of the first today. In particular, somewhere between 2016-2018 the trend of growth in language model size apparently greatly accelerated its pace, to a doubling time of between 4 and 8 months.
- Architectures on the game domain are small in terms of trainable parameters, below vision architectures while apparently growing at a similar rhythm. Naively we expected otherwise, since playing games seems more complicated. However, in hindsight, what determines model size is what are the returns to scale; in more complex domains we should expect lower effective model sizes, as the models are more constrained in other ways.
- The trend of growth in model size has been relatively stable through the transition into the deep learning era in 2011-2012 in all domains we studied (though it is hard to say with certainty given the amount of data). This suggests that the deep learning revolution was less of a paradigm change and more of a natural continuation of existing tendencies, which finally surpassed other non-machine learning methods.

Open questions

- Why is there a discrepancy in the trainable parameters magnitude and trend of growth in e.g. vision systems versus e.g. language systems? Some hypotheses are that language architectures scale better with size, that vision models are more bottlenecked on training data, that vision models require more compute per parameter or that the

language processing ML community is ahead in experiment with large scale models (e.g. because they have access to more compute and resources).

- What caused the explosive growth in the size of language models from 2018 onwards? Was it a purely social phenomena as people realized the advantages of larger models, was it enabled by the discovery of architectures that scaled better with size, compute and data (e.g. transformers?) or was it caused by something else entirely?
- Do the scaling laws of Machine Learning for pre-and-post-deep-learning actually differ significantly? So far model size seems to suggest otherwise, what about other metrics?
- How can we more accurately estimate the rates of growth for each domain and period? For how long will current rates of growth be sustained?

Next steps


- We are interested in collaborating with other researchers to grow this dataset to be more representative and correcting any mistakes. As an incentive, we will pay \$5 per mistake found or system addition (up to \$600 total among all submissions; please contact us if you want to contribute with a donation to increase the payment cap). You can send your submissions to [jaimesevillamolina at gmail dot com](mailto:jaimesevillamolina@gmail.com), preferably in spreadsheet format.
- We are interested in including other information about the systems, most notably compute and training dataset size.
- We want to include more information on other domains, specially on recommender systems.
- We want to look harder for systematic reviews and other already curated datasets of AI systems.

Acknowledgements

This article was written by Jaime Sevilla, Pablo Villalobos and Juan Felipe Cerón. Jaime's work is supported by a Marie Curie grant of the NL4XAI Horizon 2020 program.

We thank Girish Sastry for advising us on the beginning of the project, the Spanish Effective Altruism community for creating a space to incubate projects such as this one, and Haydn Belfield, Pablo Moreno and Ehud Reiter for discussion and system submissions.

Bibliography

1. Kaplan et al., "Scaling Laws for Neural Language Models," 08361.
2. 1.6 History of Reinforcement Learning. (n.d.). Retrieved June 19, 2021, from <http://incompleteideas.net/book/first/ebook/node12.html>
3. AI and Compute. (n.d.). Retrieved June 19, 2021, from <https://openai.com/blog/ai-and-compute/>
4. AI and Efficiency. (2020, May 5). OpenAI. <https://openai.com/blog/ai-and-efficiency/>
5. AI Progress Measurement. (2017, June 12). Electronic Frontier Foundation. <https://www.eff.org/ai/metrics>
6. Announcement of the 2020 ACL Test-of-Time Awards (ToT) | ACL Member Portal. (n.d.). Retrieved June 19, 2021, from <https://www.aclweb.org/portal/content/announcement-tot#:~:text=Each%20year%2C%20the%20ACL%20Test,papers%20from%2010%20years%20earlier.&text=The%20winners%20were%20announced%20at%20ACL%202020.>
7. Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? . *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–623. <https://doi.org/10.1145/3442188.3445922>

8. *Best paper awards—ACL Wiki*. (n.d.). Retrieved June 19, 2021, from https://aclweb.org/aclwiki/Best_paper_awards
9. *bnlearn—Bayesian Network Repository*. (n.d.). Retrieved June 19, 2021, from <https://www.bnlearn.com/bnrepository/>
10. *Brian Christian on the alignment problem*. (n.d.). 80,000 Hours. Retrieved June 19, 2021, from <https://80000hours.org/podcast/episodes/brian-christian-the-alignment-problem/>
11. *Computer Vision Awards – The Computer Vision Foundation*. (n.d.). Retrieved June 19, 2021, from https://www.thecvf.com/?page_id=413
12. DARPA Grand Challenge. (2021). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=DARPA_Grand_Challenge&oldid=1021627196
13. Karim, R. (2020, November 28). *Illustrated: 10 CNN Architectures*. Medium. <https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d>
14. Mohammad, S. M. (2020). Examining Citations of Natural Language Processing Literature. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 5199–5209. <https://doi.org/10.18653/v1/2020.acl-main.464>
15. Mudigere, D., Hao, Y., Huang, J., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., Luo, L., Yang, J. A., Gao, L., Ivchenko, D., Basant, A., Hu, Y., Yang, J., Ardestani, E. K., Wang, X., Komuravelli, R., ... Rao, V. (2021). High-performance, Distributed Training of Large-scale Deep Learning Recommendation Models. *ArXiv:2104.05158 [Cs]*. <http://arxiv.org/abs/2104.05158>
16. Nilsson, N. (1974). *Artificial Intelligence*. IFIP Congress. <https://doi.org/10.7551/mitpress/11723.003.0006>
17. Posey, L. (2020, April 28). *History of AI Research*. Medium. <https://towardsdatascience.com/history-of-ai-research-90a6cc8adc9c>
18. Raschka, S. (2019). A Brief Summary of the History of Neural Networks and Deep Learning. *Deep Learning*, 29.
19. Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2020). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *ArXiv:1910.01108 [Cs]*. <http://arxiv.org/abs/1910.01108>
20. Thompson, N. C., Greenewald, K., Lee, K., & Manso, G. F. (2020). The Computational Limits of Deep Learning. *ArXiv:2007.05558 [Cs, Stat]*. <http://arxiv.org/abs/2007.05558>
21. Vidal, R. (n.d.). *Computer Vision: History, the Rise of Deep Networks, and Future Vistas*. 60.
22. Wang, B. (2021). *Kingoflolz/mesh-transformer-jax* [Jupyter Notebook]. <https://github.com/kingoflolz/mesh-transformer-jax> (Original work published 2021)
23. *Who Invented Backpropagation?* (n.d.). Retrieved June 19, 2021, from <https://people.idsia.ch/~juergen/who-invented-backpropagation.html>
24. Xie, Q., Luong, M.-T., Hovy, E., & Le, Q. V. (2020). Self-training with Noisy Student improves ImageNet classification. *ArXiv:1911.04252 [Cs, Stat]*. <http://arxiv.org/abs/1911.04252>
25. Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent Trends in Deep Learning Based Natural Language Processing. *ArXiv:1708.02709 [Cs]*. <http://arxiv.org/abs/1708.02709>
26. Zhang, B., Xiong, D., Su, J., Lin, Q., & Zhang, H. (2018). Simplifying Neural Machine Translation with Addition-Subtraction Twin-Gated Recurrent Networks. *ArXiv:1810.12546 [Cs]*. <http://arxiv.org/abs/1810.12546>
27. Zoph, B., & Le, Q. V. (2016). *Neural Architecture Search with Reinforcement Learning*. <https://arxiv.org/abs/1611.01578v2>

Estimating training compute of Deep Learning models

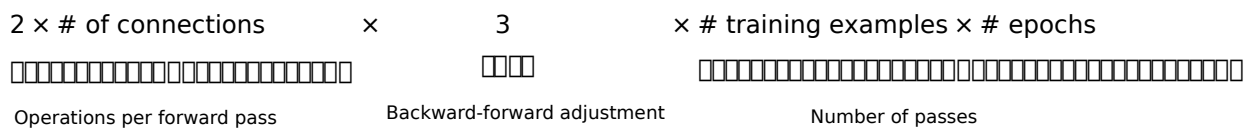
Crossposted from the [AI Alignment Forum](#). May contain more technical jargon than usual.

by Jaime Sevilla, Lennart Heim, Marius Hobbhahn, Tamay Besiroglu, and Anson Ho

You can find the complete article [here](#). We provide a short summary below.

In short: To estimate the compute used to train a Deep Learning model we can either: 1) directly count the number of operations needed or 2) estimate it from GPU time.

Method 1: Counting operations in the model



Method 2: GPU time

$$\text{training time} \times \# \text{ cores} \times \text{peak FLOP/s} \times \text{utilization rate}$$

We are uncertain about what utilization rate is best, but our recommendation is 30% for Large Language Models and 40% for other models.

You can read more about method 1 [here](#) and about method 2 [here](#).

Other parts of interest of this article include:

- We argue that the ratio of operations of backward and forward pass of neural networks is often close to 2:1. [More](#).
- We discuss how the formula of method 1 changes for recurrent models. [More](#).
- We argue that dropout does not affect the number of operations per forward and backward pass. [More](#).
- We have elaborated a table with parameter and operation counts for common neural network layers. [More](#).
- We give a detailed example of method 1. [More](#).
- We discuss commonly used number representation formats in ML. [More](#).
- We share an estimate of the average performance of GPU cards each year. [More](#).
- We share some reported GPU usages in real experiments. [More](#).
- We give a detailed example of method 2. [More](#).
- We compare both methods and conclude they result in similar estimates. [More](#).
- We discuss the use of profilers to measure compute. [More](#).

Complete Article

You can find the article [here](#).

What's the backward-forward FLOP ratio for Neural Networks?

Crossposted from the [AI Alignment Forum](#). May contain more technical jargon than usual.

Summary:

1. *Classic settings*, i.e. deep networks with convolutional layers and large batch sizes, **almost always have backward-forward FLOP ratios close to 2:1**.
2. Depending on the following criteria we can encounter **ratios between 1:1 and 3:1**
 1. **Type of layer:** Passes through linear layers have as many FLOP as they use to do weight updates. Convolutional layers have many more FLOP for passes than for weight updates. Therefore, in CNNs, FLOP for weight updates basically play no role.
 2. **Batch size:** Weights are updated after the gradients of the batch have been aggregated. Thus, FLOP for passes increase with batch size but stay constant for weight updates.
 3. **Depth:** The first layer has a backward-forward ratio of 1:1 while all others have 2:1. Therefore, the overall ratio is influenced by the fraction of FLOP in first vs. FLOP in other layers.
3. We assume the network is being optimized by stochastic gradient descent ($w \leftarrow w - \alpha \cdot dw$) and count the weight update as part of the backward pass. Other optimizers would imply different FLOP counts and could create ratios even larger than 3:1 for niche settings (see appendix B). However, the ratio of 2:1 in the classic setting (see point 1) should still hold even when you use momentum or Adam.

Compute-intensity of the weight update	Most compute-intensive layers	Backward-forward ratio
Large batch size OR compute-intensive convolutional layer	First layer	1:1
	Other layers	2:1
Small batch size AND no compute-intensive convolutional layers	First layer	
	Other layers	3:1

Introduction:

How many more floating-point operations (FLOP) does it take to compute a backward pass than a forward pass in a neural network? We call this the backward-forward FLOP ratio.

This ratio is useful to estimate the total amount of training compute from the forward compute; something we are interested in the context of our study of [Parameter, Compute and Data Trends in Machine Learning](#).

In this post, we first provide a theoretical analysis of the ratio, and we then corroborate our findings empirically.

Theory:

To understand where the differences in ratios come from, we need to look at the classical [equations of backpropagation](#).

Summary: the equations of backpropagation

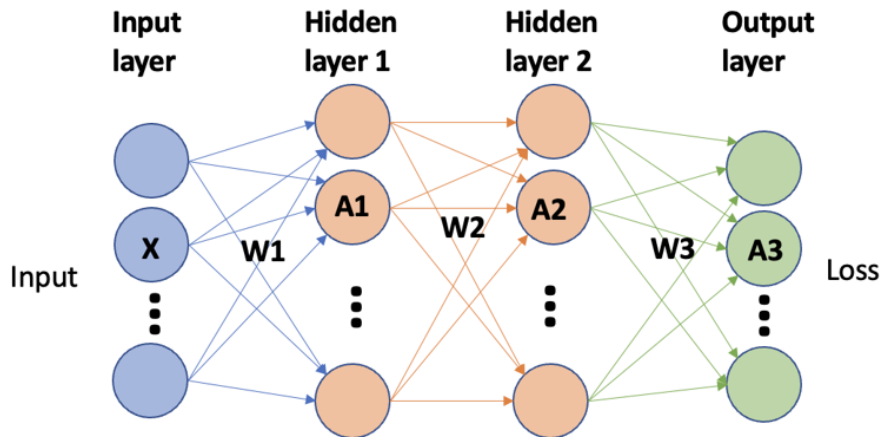
$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (\text{BP1})$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (\text{BP2})$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (\text{BP3})$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (\text{BP4})$$

Let's start with a simple example---a neural network with 2 hidden layers.



In this example, we have the following computations for forward and backward pass assuming linear layers with ReLU activations. The "@"-symbols denote matrix multiplications.

Operation	Computation	FLOP forward	Computation	FLOP backward
Input	$A1 = W1 @ X$	$2 * \#input * \#hidden1 * \#batch$	$dL/dW1 = \delta1 @ X$	$2 * \#input * \#hidden1 * \#batch$
ReLU	$A1R = \text{ReLU}(A1)$	$\#hidden1 * \#batch$	$\delta1 = d\delta1R/dA1$	$\#hidden1 * \#batch$
Derivative			$\delta1R = dL/dA2 = W2 @ \delta2$	$2 * \#hidden1 * \#hidden2 * \#batch$
Hidden1	$A2 = W2 @ A1R$	$2 * \#hidden1 * \#hidden2 * \#batch$	$dL/dW2 = \delta2 @ A1R$	$2 * \#hidden1 * \#hidden2 * \#batch$
ReLU	$A2R = \text{ReLU}(A2)$	$\#hidden2 * \#batch$	$\delta2 = d\delta2R/dA2$	$\#hidden2 * \#batch$
Derivative			$\delta2R = dL/dA3 = W3 @ \delta3$	$2 * \#hidden2 * \#output * \#batch$

Hidden2	$A3 = W3 @ A2R$	$2 * \# \text{hidden2} * \# \text{output} * \# \text{batch}$	$dL/dW3 = \delta 3 @ A2R$	$2 * \# \text{hidden2} * \# \text{output} * \# \text{batch}$
ReLU	$A3R = \text{ReLU}(A3)$	$\# \text{output} * \# \text{batch}$	$\delta 3 = d\delta 3R/dA3$	$\# \text{output} * \# \text{batch}$
Loss	$L = \text{loss}(A3R, Y)$	$\# \text{output} * \# \text{batch}$	$\delta 3R = dL/dA3R$	$\# \text{output} * \# \text{batch}$
Update			$W += lr * \delta W$	$2 * \# \text{weights}$

We separate the weight update from the individual layers since the update is done after aggregation, i.e. we first add all gradients coming from different batches and then multiply with the learning rate.

From this table we see

1. ReLUs and the loss function contribute a negligible amount of FLOP compared to layers.
2. For the first layer, the backward-forward FLOP ratio is 1:1
3. For all other layers, the backward-forward FLOP ratio is 2:1 (ignoring ReLUs)

In equation form, the formula for the backward-forward FLOP ratio is:

backward / forward =

$(\text{FIRST LAYER FORWARD FLOP} + 2 * \text{OTHER LAYERS FORWARD FLOP} + \text{WEIGHT UPDATE}) / (\text{FIRST LAYER FORWARD FLOP} + \text{OTHER LAYERS FORWARD FLOP})$

There are two considerations to see which terms dominate in this equation:

1. How much of the computation happens in the first layer?
2. How many operations does the weight update take compared to the computation in the layers? If the batch size is large or many parameters are shared, this term can be dismissed. Otherwise, it can be approximated as $\text{WEIGHT UPDATE} \approx \text{FIRST LAYER FORWARD FLOP} + \text{OTHER LAYERS FORWARD FLOP}$.

This leads us to four possible cases:

	Big weight update	Small weight update
First layer dominant	$2 * \text{FIRST LAYER FORWARD FLOP} / \text{FIRST LAYER FORWARD FLOP} = \mathbf{2:1}$	$\text{FIRST LAYER FORWARD FLOP} / \text{FIRST LAYER FORWARD FLOP} = \mathbf{1:1}$
Other layers dominant	$3 * \text{OTHER LAYERS FORWARD FLOP} / \text{OTHER LAYERS FORWARD FLOP} = \mathbf{3:1}$	$2 * \text{OTHER LAYERS FORWARD FLOP} / \text{OTHER LAYERS FORWARD FLOP} = \mathbf{2:1}$

The norm in modern Machine Learning is **deep networks** with **large batch sizes**, where our analysis predicts a ratio close to **2:1**.

In short, our theoretical analysis predicts that the backward-forward FLOP ratio will be between **1:1** and **3:1**, with **2:1** being the typical case.

Empirical results:

To corroborate our analysis we use [NVIDIA's pyprof profiler](#) to audit the amount of FLOP in each layer during the backward and forward pass.

In this section we will explore:

- The difference between the backward-forward ratio in the first and the rest of the layers.
- The difference between the weight update in convolutional and linear layers.
- The effect of a large batch size on the weight update.
- The effect of depth on the backward-forward ratio.
- The combined effects of batch-size, convolutional layers and depth.

In short, our empirical results confirm our theoretical findings.

In a [previous post](#), we tried to estimate utilization rates. As detailed in the previous post, the profiler does under- and overcounting. Thus, we believe some of the estimates are slightly off.

We have tried to correct them as much as possible. In particular, we eliminate some operations which we believe are double-counted, and we add the operations corresponding to multiplication by the learning rate which we believe are not counted in stochastic gradient descent.

Backward and forward FLOP in the first and the rest of the layers:

We can investigate this empirically by looking at a simple linear network (code in appendix).

It results in the following FLOP counts:

Direction	Op	FLOPs
0	fprop	linear 1233125376
1	fprop	relu 4096
2	fprop	linear 1048576
3	fprop	relu 128
4	fprop	linear 2560
5	fprop	cross_entropy 0
6	fprop	backward 0
7	bprop	cross_entropy 0
8	bprop	linear 2560
9	bprop	linear 2560
10	bprop	relu 128
11	bprop	linear 1048576
12	bprop	linear 1048576
13	bprop	relu 4096
14	bprop	linear 1233125376
15	bprop	add_ 1234184980

We can see that the first layer (red) has the same flop count for forward and backward pass while the other layers (blue, green) have a ratio of 2:1. The final weight update (yellow) is 2x the number of parameters of the network.

Type of layer:

The number of FLOP is different for different types of layers.

Layer	Number of parameters	Number of floating-point operations
Fully connected layer from N neurons to M neurons	$N \cdot M + M \approx N \cdot M$	$2 \cdot N \cdot M + M + M \approx 2 \cdot N \cdot M$
		WEIGHTS BIASES NONLINEARITIES
CNN' from a tensor of shape $H \times W \times C$ with D filters of shape $K \times K \times C$, applied with stride S and padding P	$D \cdot K \cdot K \cdot C + K \cdot K \cdot C \approx D \cdot K \cdot K \cdot C$	$2 \cdot H \times W \times C \cdot H' \times W' \times D + H' \times W' \times D + H' \times W' \times D \approx 2 \cdot H^2 \cdot W^2 \cdot C \cdot D / S^2$ <div> where $H' = [(H - K + 2P + 1) / S]$ $W' = [(W - K + 2P + 1) / S]$ so that $H' \times W' \times D$ is the output shape </div>
		WEIGHTS BIASES NONLINEARITIES

As we can see, the number of FLOP for linear layers is 2x their number of parameters. For CNNs the number of FLOP is much higher than the number of parameters. This means that the final weight update is basically negligible for CNNs but relevant for linear networks.

To show this empirically, we look at the profiler FLOP counts of a small CNN (code in appendix).

	Direction	Op	FLOPs
0	fprop	conv2d	118013952
1	fprop	relu	401408
2	fprop	max_pool2d	0
3	fprop	conv2d	157351936
4	fprop	relu	50176
5	fprop	max_pool2d	0
6	fprop	adaptive_avg_pool2d	0
7	fprop	linear	1280
8	fprop	cross_entropy	0
9	fprop	backward	0
10	bprop	cross_entropy	0
11	bprop	linear	1280
12	bprop	linear	1280
13	bprop	max_pool2d	0
14	bprop	relu	50176
15	bprop	conv2d	157351936
16	bprop	conv2d	157351936
17	bprop	max_pool2d	0
18	bprop	relu	401408
19	bprop	conv2d	118013952
20	bprop	add_	211412

Similar to the linear network, we can confirm that the backward-forward ratio for the first layer is 1:1 and that of all others 2:1. However, the number of FLOP in layers (red, blue, green) is much larger than for the weight update (yellow).

Batch size:

Gradients are aggregated before the weight update. Thus, the FLOP for weight updates stays the same for different batch sizes (yellow) while the FLOP for all other operations scales with the batch size (blue, green, red). As a consequence, larger batch sizes make the FLOP from weight updates negligibly small.

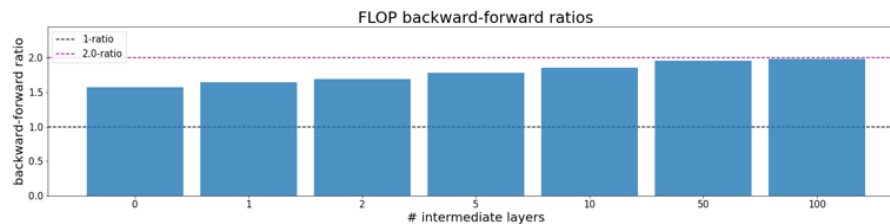
Direction	Op	FLOP_1	FLOP_64	FLOP_128	FLOP_512	FLOP_64/FLOP_1	FLOP_128/FLOP_1	FLOP_512/FLOP_1
0	fprop	conv2d	118013952	7552892928	15105785856	64	128	512
1	fprop	relu	401408	25690112	51380224	64	128	512
2	fprop	max_pool2d	0	0	0	nan	nan	nan
3	fprop	conv2d	157351936	10070523904	20141047808	64	128	512
4	fprop	relu	50176	3211264	6422528	64	128	512
5	fprop	max_pool2d	0	0	0	nan	nan	nan
6	fprop	adaptive_avg_pool2d	0	0	0	nan	nan	nan
7	fprop	linear	1280	81920	163840	64	128	512
8	fprop	cross_entropy	0	0	0	nan	nan	nan
9	fprop	backward	0	0	0	nan	nan	nan
10	bprop	cross_entropy	0	0	0	nan	nan	nan
11	bprop	linear	1280	81920	163840	64	128	512
12	bprop	linear	1280	81920	163840	64	128	512
13	bprop	max_pool2d	0	0	0	nan	nan	nan
14	bprop	relu	50176	3211264	6422528	64	128	512
15	bprop	conv2d	157351936	10070523904	20141047808	64	128	512
16	bprop	conv2d	157351936	10070523904	20141047808	64	128	512
17	bprop	max_pool2d	0	0	0	nan	nan	nan
18	bprop	relu	401408	25690112	51380224	64	128	512
19	bprop	conv2d	118013952	7552892928	15105785856	64	128	512
20	bprop	add_	211412	211412	211412	1	1	1

Depth:

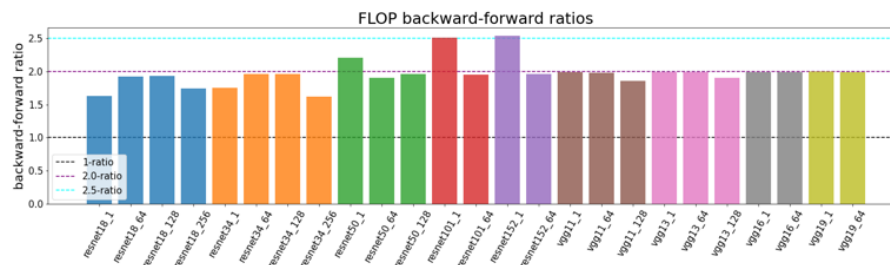
Depth, i.e. the number of layers only has an indirect influence. This stems from the fact that the first layer has a ratio of 1:1 while further layers have a ratio of 2:1. Thus, the true influence comes from FLOP in the first layer vs. every other layer.

To show this effect, we define a CNN with different numbers of intermediate conv layers (code in appendix).

We find that the backward-forward starts significantly below 2:1 for 0 intermediate layers and converges towards 2:1 when increasing the number of intermediate layers.



Most common deep learning CNN architectures are deep enough that the first layer shouldn't have a strong effect on the overall number of FLOP and thus the ratio should be close to 2:1. We have empirically tested this for multiple different types of resnets and batch sizes. We observe some diverge from the expected 2:1 ratio but we think that this is a result of the profiler undercounting certain operations. We have described problems with the profiler in the [previous post](#).



Backward-forward FLOP ratio in different architectures. Read the labels as architecture_batchsize.

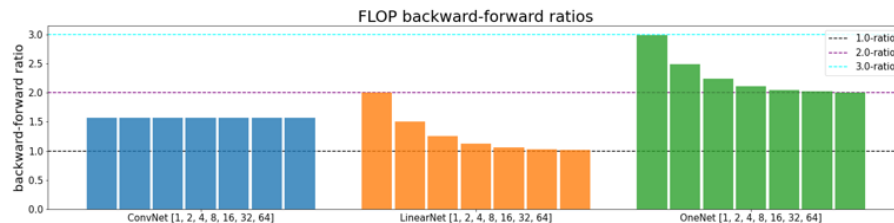
Combining all above:

There are interdependencies of batch size, type of layer and depth which we want to explore in the following. We compare the small CNN and the linear network that were already used before with a network we call OneNet (code in appendix). OneNet has only one input neuron and a larger second and third layer. Thus, the ratio between the first and

other layers is very small and we can see that the theoretical maximum for the backward-forward ratio of 3:1 can be observed in practice.

Furthermore, we look at exponentially increasing batch sizes for all three architectures. In the case of linear networks, i.e. LinearNet and OneNet, the ratio decreases with increasing batch size since the influence of the weight update is reduced. In the case of the CNN, the FLOP count is completely dominated by layers and the weight update is negligible. This effect is so strong that no change can be observed in the figure.

We see that LinearNet converges to a backward-forward ratio of 1:1 for larger batch sizes while OneNet converges to 2:1. This is because nearly all weights of LinearNet are in the first layer and nearly all weights of OneNet in the other layers.



Conclusion:

We have reasoned that the backward-forward FLOP ratio in Neural Networks will typically be between 1:1 and 3:1, and most often close to 2:1.

The ratio depends on the batch size, how much computation happens in the first layer versus the others, the degree of parameter sharing and the batch size.

We have confirmed this in practice. However, we have used a profiler with some problems, so we cannot completely rule out a mistake.

Acknowledgments

The experiments have been conducted by Marius Hobbhahn. The text was written by MH and Jaime Sevilla.

Lennart Heim helped greatly with discussion and support. We also thank Danny Hernandez and Girish Sastry for discussion.

Appendix A: Code for all networks

```

    """ linear network with large first layer and small later layers
class LinearNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(224*224*3, 4096)
        self.fc2 = nn.Linear(4096, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

""" linear network with just one input but larger intermediate layers
class OneNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1, 4096)
        self.fc2 = nn.Linear(4096, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

""" small conv net
class ConvNet(nn.Module):
    def __init__(self):

```

```

super(ConvNet, self).__init__()
self.conv1 = nn.Conv2d(3, 32, kernel_size=7, stride=2, padding=3, bias=False)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
self.conv2 = nn.Conv2d(32, 64, kernel_size=7, stride=2, padding=3, bias=False)
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc1 = nn.Linear(64, 10)

def forward(self, x):
    x = self.maxpool(self.relu(self.conv1(x)))
    x = self.maxpool(self.relu(self.conv2(x)))
    x = self.avgpool(x)
    x = torch.flatten(x, 1) # flatten all dimensions except batch
    x = self.fc1(x)
    return x

### conv net with different sizes for intermediate layers
class DeeperConvNet(nn.Module):

    def __init__(self):
        super(DeeperConvNet, self).__init__()
        self.first_layer = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=7, stride=2, padding=3, bias=False),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.conv_layer = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1, bias=False),
            nn.ReLU(inplace=True)
        )
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.convN = nn.Conv2d(32, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc1 = nn.Linear(64, 10)

    def forward(self, x):
        x = self.first_layer(x)
        for i in range(100):
            x = self.conv_layer(x)
        x = self.relu(self.convN(x))
        x = self.avgpool(x)
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.fc1(x)
        return x

```

Appendix B: Using other optimizers

Through this post we have assumed stochastic gradient descent (SGD) for the weight update. SGD involves multiplying the gradient by a learning rate and adding the result to the current weights. That is, it requires 2 FLOP per parameter.

Other optimizers require some extra work. For example, consider [adaptive moment estimation \(Adam\)](#). Adam's parameter update is given by:

$$\begin{aligned}
 g_t &\leftarrow \nabla_{\theta} f_t(\theta_{t-1}) \text{ (Get gradients w.r.t. stochastic objective at timestep } t\text{)} \\
 m_t &\leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \text{ (Update biased first moment estimate)} \\
 v_t &\leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \text{ (Update biased second raw moment estimate)} \\
 \hat{m}_t &\leftarrow m_t / (1 - \beta_1^t) \text{ (Compute bias-corrected first moment estimate)} \\
 \hat{v}_t &\leftarrow v_t / (1 - \beta_2^t) \text{ (Compute bias-corrected second raw moment estimate)} \\
 \theta_t &\leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) \text{ (Update parameters)}
 \end{aligned}$$

For a total of $\sim 3 + 4 + 3 + 3 + 5 = 18$ FLOP per parameter.

In any case, the choice of optimizer affects only the weight update and the amount of FLOP is proportional to the number of parameters. Since batch sizes are typically large, the difference will be small and won't affect the backward-forward ratio much.

How to measure FLOP/s for Neural Networks empirically?

Crossposted from the [AI Alignment Forum](#). May contain more technical jargon than usual.

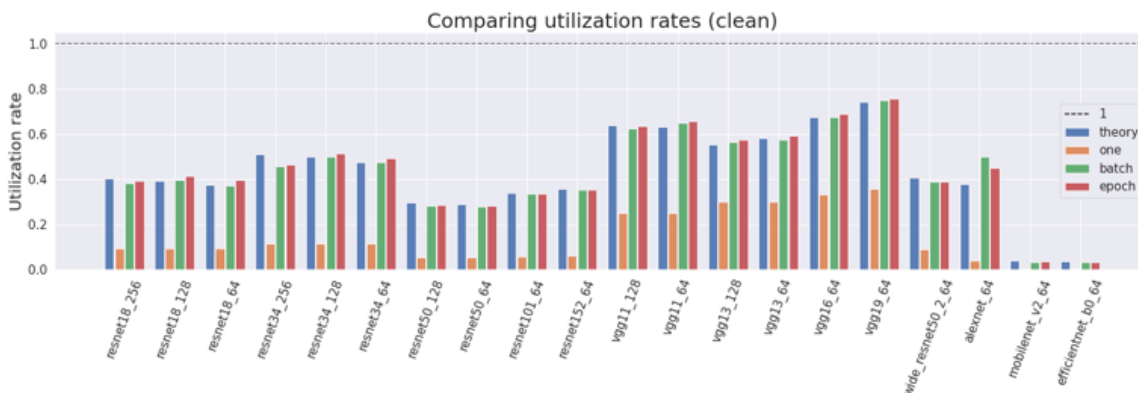
Experiments and text by Marius Hobbhahn. I would like to thank Jaime Sevilla, Jean-Stanislas Denain, Tamay Besiroglu, Lennart Heim, and Anson Ho for their feedback and support.

Summary:

We measure the utilization rate of a Tesla P100 GPU for training different ML models. Most architectures and methods result in a utilization rate between 0.3 and 0.75. However, two architectures result in implausible low utilization rates of lower than 0.04. The most probable explanation for these outliers is that FLOP for inverted bottleneck layers are not counted correctly by the profiler. In general, the profiler we use shows signs of under- and overcounting and there is a possibility we made errors.

Findings:

- Counting the FLOP for a forward pass is very simple and many different packages give correct answers.
- Counting the FLOP for the backward pass is harder and our estimator of choice makes weird overcounting and undercounting errors.
- After cleaning mistakes, it is very likely that the backward/forward ratio is 2:1 (at least for our setup).
- After correcting for the overcounting issues, we get empirical utilization rates between 0.3 and 0.75 for most architectures. Theoretical predictions and empirical measurements seem very consistent for larger batch sizes.



Estimated GPU utilization rates on different architectures, using four different estimation setups.

Introduction

In the "Parameter, Compute and Data Trends in Machine Learning" project we wanted to estimate GPU utilization rates for different Neural Networks and GPUs. While this sounds very easy in theory, it turned out to be hard in practice.

Utilization rate = empirical performance / peak performance

The post contains a lot of technical jargon. If you are just here for the results, skip to the Analysis section.

I don't have any prior experience in estimating FLOP. It is very possible that I made rookie mistakes. Help and suggestions are appreciated.

Other work on computing and measuring FLOP can be found in Lennart Heim's sequences [Transformative AI and Compute](#). It's really good.

Methods for counting FLOP

In this post, we use FLOP to denote floating-point operations and FLOP/s to mean FLOP per second.

We can look up the peak FLOP/s performance of any GPU by checking its datasheet (see e.g. [NVIDIA's Tesla P100](#)). To compare our empirical performance to the theoretical maximum, we need to measure the number of FLOP and time for one training run. This is where things get confusing.

Packages such as PyTorch's [fvcore](#), [ptflops](#) or [pthflops](#) hook onto your model and compute the FLOP for one forward pass for a given input. However, they can't estimate the FLOP for a backward pass. Given that we want to compute the utilization rate for the entire training, accurate estimates of FLOP for the backward pass are important.

PyTorch also provides a list of packages called profilers, e.g. in the [main package](#) and [autograd](#). The profilers hook onto your model and measure certain quantities at runtime, e.g. CPU time, GPU time, FLOP, etc. The profiler can return aggregate statistics or individual statistics for every single operation within the training period. Unfortunately, these two profilers seem to not count the backward pass either.

[NVIDIA offers an alternative way of using the profiler with Nsight Systems](#) that supposedly estimates FLOP for forward and backward pass accurately. This would suffice for all of our purposes. Unfortunately, we encountered problems with the estimates from this method. It shows signs of over- and undercounting operations. While we could partly fix these issues post-hoc, there is still room for errors in the resulting estimates.

NVIDIA also offers a profiler called [dlprof](#). However, we weren't able to run it in Google Colab (see appendix).

Our experimental setup

We try to estimate the empirical utilization rates of 13 different conventional neural network classification architectures (resnet18, resnet34, resnet50, resnet101, resnet152, vgg11, vgg13, vgg16, vgg19, wide_resnet50_2, alexnet, mobilenet_v2, efficientnet_b0) with different batch sizes for some of them. For all experiments, we use the [Tesla P100](#) GPU which seems to be the default for Google Colab. All experiments have been done in Google Colab and can be reproduced [here](#).

We estimate the FLOP for a forward pass with fvcore, ptflops, pthflops and the PyTorch profiler. Furthermore, we compare them to the FLOP for forward and backward pass estimated by the profiler + nsight systems method (which we name profiler_nvtx). We measure the time for all computations once with the profiler and additionally with profiler_nvtx to get comparisons.

One problem for the estimation of FLOP is that fvcore, ptflops and pthflops seem to count a [Fused Multiply Add \(FMA\)](#) as one operation while the profiler methods count it as 2. Since

basically all operations in NNs are FMAs that means we can just divide all profiler estimates by 2. We already applied this division to all estimates, so you don't have to do it mentally. However, this is one potential source for errors since some operations might not be FMAs.

Furthermore, it is not 100 percent clear which FMA convention was used for the peak performance. [On their website](#), NVIDIA states “The peak single-precision floating-point performance of a CUDA device is defined as the number of CUDA Cores times the graphics clock frequency multiplied by two. The factor of two stems from the ability to execute two operations at once using fused multiply-add (FFMA) instructions”.

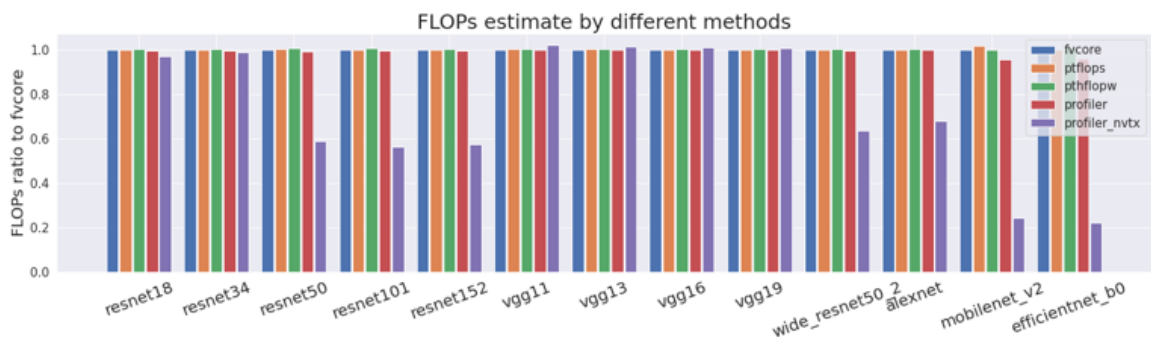
We interpret this statement to mean that NVIDIA used the $\text{FMA}=2\text{FLOP}$ assumption. However, PyTorch automatically transforms all single-precision tensors to half-precision during training. Therefore, we get a speedup factor of 2 (which cancels the $\text{FMA}=2\text{FLOP}$)

For all experiments, we use input data of sizes $3 \times 224 \times 224$ with 10 classes. This is similar to many common image classification setups. We either measure on single random batches of different sizes or on the test set of CIFAR10 containing 10000 images.

Analysis:

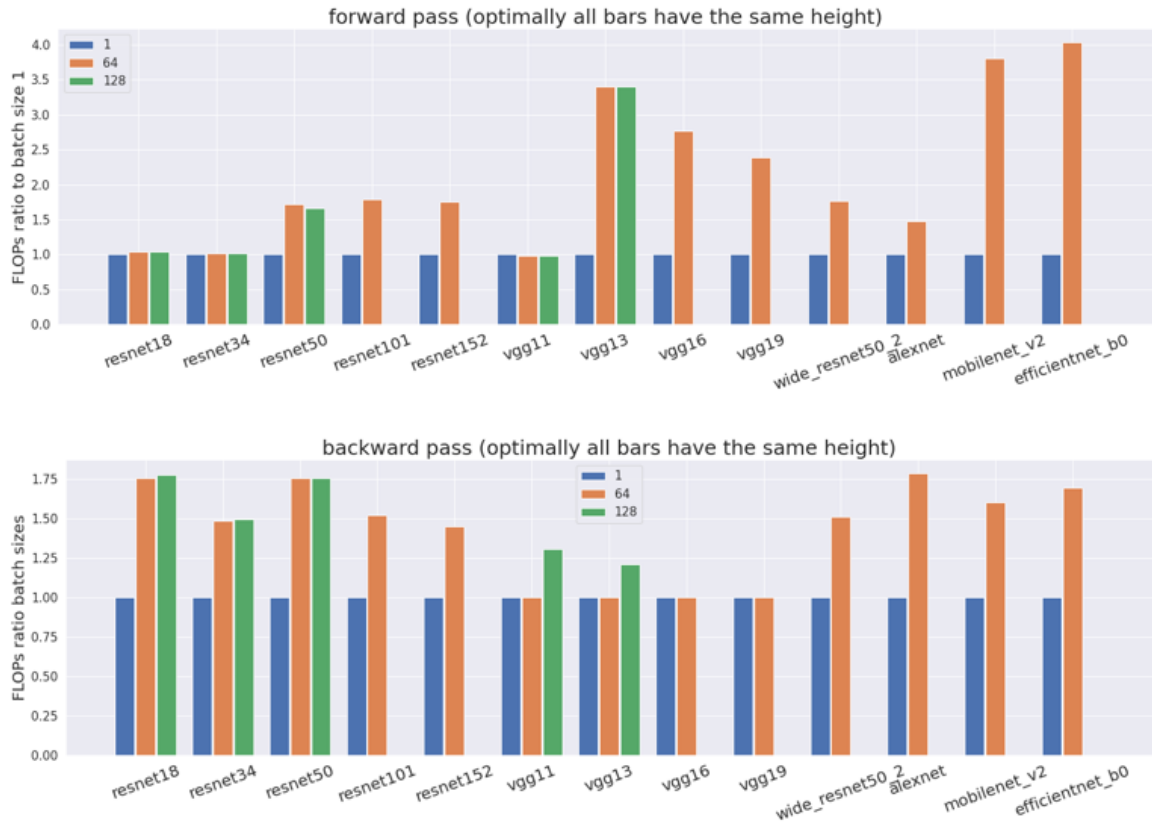
Something is fishy with profiler_nvtx

To understand the estimates for the profiler_nvtx better, we run just one single forward and backward pass with different batch sizes. If we compare the profiler_nvtx FLOP estimates for one forward pass on a random batch of size one, we see that they sometimes don't align with all other estimates.



The first four methods basically always yield very comparable estimates and just profiler_nvtx sometimes undercounts quite drastically.

But it gets worse —profiler_nvtx is inconsistent with its counting. We expected the number of FLOP for a batch size of 64 to be 64 times as large as for a batch size of 1, and the number of FLOP for a batch size of 128 to be 128 times as large as for a batch size of 1. However, this is not the case for both the forward and backward pass.



All FLOP estimates have been normalized by the batch size. Thus, if our profiler counted correctly, all bars would have exactly the same height. This is not what we observe in some networks, which suggests that something is off. Some networks don't have estimates for a batch size of 128 since it didn't fit into the GPU memory.

To check whether profiler_nvtx is over- or undercounting we investigate it further.

Investigating profiler_nvtx further:

Since the problems from above cause all analyses to be very uncertain, we try to find out what exactly is wrong and if we can fix it in the following section. If you don't care about that, skip to the Results section.

If we compare the counted FLOP by operation, e.g. on alexnet, we make multiple discoveries.

- **FMAs:** We find that profiler_nvtx counts exactly 2x as many FLOP as fvcare (red in table) since profiler_nvtx counts FMAs as 2 and fvcare as 1 FLOP. For the same reason, profiler_nvtx counts 128 as many operations when we use a batch size of 64 (blue in table).
- **Undercounting:** In some cases (green in table) profiler_nvtx just doesn't register an operation and therefore counts 0 FLOP.
- **Overcounting:** In other cases (yellow in table), profiler_nvtx counts the same operation multiple times for no apparent reason.

	Layer	fvcore	ptflops	nvtx_1	nvtx_64	nvtx_1/fvcore	nvtx_64/fvcore
0	Conv2d	70276800	70000000.0	0	8995430400	0.0	128.0
1	ReLU	0	0.0	193600	12390400	inf	inf
2	MaxPooling2d	0	0.0	0	0	NaN	NaN
3	Conv2d	223948800	224000000.0	0	28665446400	0.0	128.0
4	ReLU	0	0.0	139968	8957952	inf	inf
5	MaxPooling2d	0	0.0	0	0	NaN	NaN
6	Conv2d	112140288	112000000.0	224280576	14353956864	2.0	128.0
7	ReLU	0	0.0	64896	4153344	inf	inf
8	Conv2d	149520384	150000000.0	299040768	19138609152	2.0	128.0
9	ReLU	0	0.0	43264	2768896	inf	inf
10	Conv2d	99680256	100000000.0	199360512	12759072768	2.0	128.0
11	ReLU	0	0.0	43264	2768896	inf	inf
12	MaxPooling2d	0	0.0	0	0	NaN	NaN
13	AvgPooling	0	0.0	0	0	NaN	NaN
14	Dropout	0	0.0	46080	2949120	inf	inf
15	Linear	37748736	39000000.0	75497472	4831838208	2.0	128.0
16	Linear (x2)	0	0.0	75497472	0	inf	NaN
17	ReLU	0	0.0	4096	262144	inf	inf
18	Dropout	0	0.0	20480	1310720	inf	inf
19	Linear	16777216	17000000.0	33554432	2147483648	2.0	128.0
20	ReLU	0	0.0	4096	262144	inf	inf
21	Linear	40960	0.0	81920	5242880	2.0	128.0

This double-counting can happen in more extreme versions. In the forward pass of VGG13, for example, profiler_nvtx counts a single operation 16 times. That is 15 times too often. Obviously, this distorts the results.

5	Conv2d	924844032	926000000	3211264	118380036096
6	ReLU	0	2000000	1605632	102760448
7	Conv2d	1849688064	1851000000	3699376128	236760072192
8	Conv2d (x2)	0	0	0	236760072192
9	Conv2d (x3)	0	0	0	236760072192
10	Conv2d (x4)	0	0	0	236760072192
11	Conv2d (x5)	0	0	0	236760072192
12	Conv2d (x6)	0	0	0	236760072192
13	Conv2d (x7)	0	0	0	236760072192
14	Conv2d (x8)	0	0	0	236760072192
15	Conv2d (x9)	0	0	0	236760072192
16	Conv2d (x10)	0	0	0	236760072192
17	Conv2d (x11)	0	0	0	236760072192
18	Conv2d (x12)	0	0	0	236760072192
19	Conv2d (x13)	0	0	0	236760072192
20	Conv2d (x14)	0	0	0	236760072192
21	Conv2d (x15)	0	0	0	236760072192
22	Conv2d (x16)	0	0	0	236760072192
23	ReLU	0	2000000	1605632	102760448
24	MaxPooling2d	0	2000000	0	0

Furthermore, we can check the empirical backward/ forward ratios from profiler_nvtx in detail. We find that

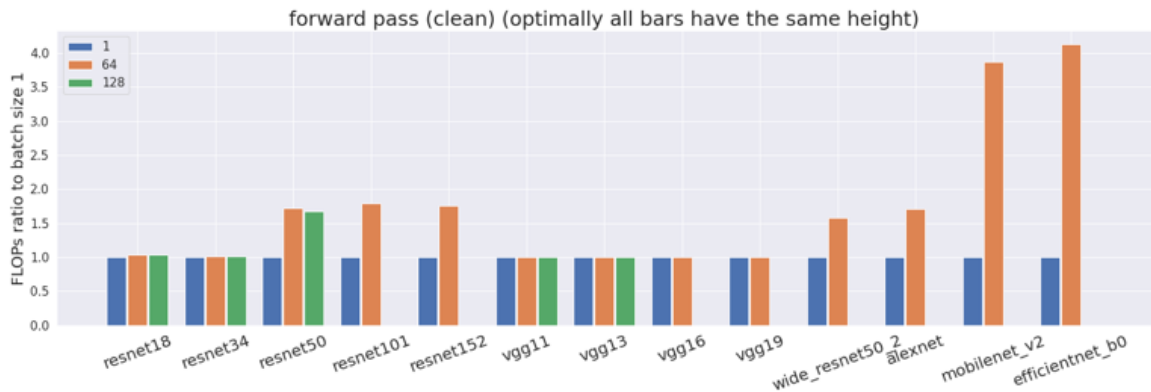
- Operations like conv2d and linear have a backward/forward ratio of 2:1.
- Operations like relu, dropout, maxpooling, avgpooling have a backward/forward ratio of 1:1.

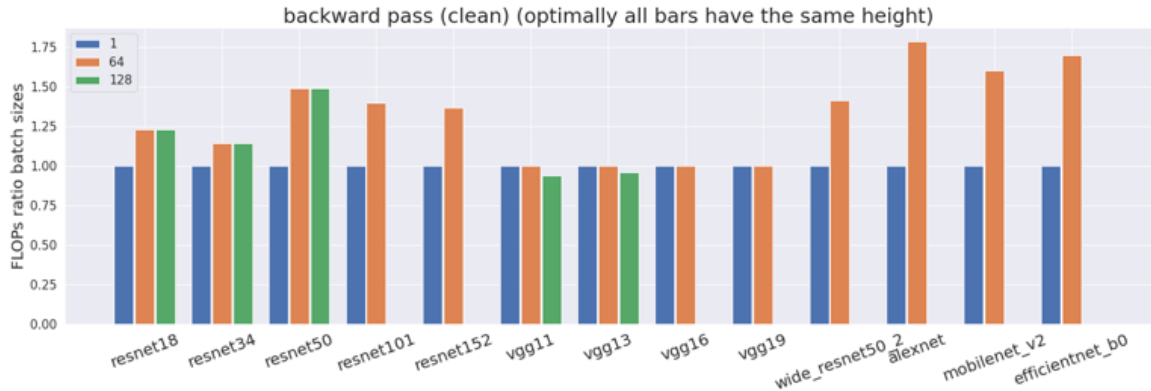
Since the vast majority of operations during training come from conv2d and linear layers, The overall ratio is therefore very close to 2:1.

	Idx	Direction	Module	Op	FLOPs	Sil(ns)
57	58	fprop	torch.nn.functional	conv2d	12759072768	688219
60	61	fprop	torch.nn.functional	relu	2768896	42528
72	73	fprop	torch.nn.functional	linear	5242880	134303
73	74	fprop	torch.nn.functional	cross_entropy	0	3744
75	76	fprop	Tensor	backward	0	2048
79	80	bprop	torch.nn.functional	linear	5242880	23616
80	81	bprop	torch.nn.functional	linear	5242880	13663
96	97	bprop	torch.nn.functional	relu	2768896	60416
100	01	bprop	torch.nn.functional	conv2d	12759072768	693276
104	05	bprop	torch.nn.functional	conv2d	12759072768	710843

To account for the double-counting mistakes from above, we cleaned up the original files and deleted all entries that mistakenly double-counted an operation. Note that we couldn't fix the undercounting issue so the following numbers still contain undercounts sometimes.

After fixing the double-counting issue we get slightly more consistent results for different batch sizes.





All remaining inconsistencies come from undercounting.

Results:

The following results are done on the cleaned version of the profiler data, i.e. double counting has been removed but undercounting still poses an issue.

The same analysis for the original (uncleaned) data can be found in the appendix.

Comparing batch sizes:

We trained some of our models with different batch sizes. We are interested in whether different batch sizes affect the time it takes to train models. We additionally compare the timings from the conventional profiler and profiler_nvtx.



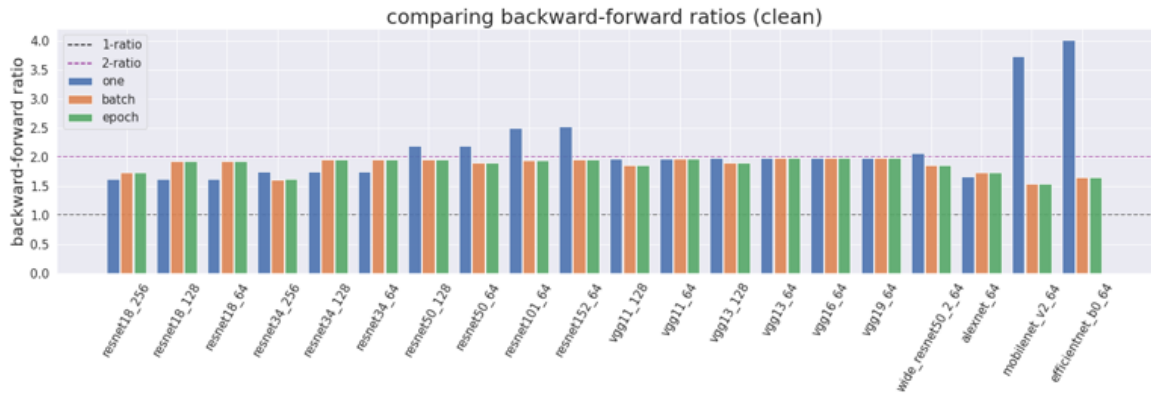
We find that, as expected, larger batch sizes lead to minimally shorter training times for 4 out of the 5 models. We are not sure why VGG13 is an exception. We would have expected the differences between batch sizes to be larger but don't have a strong explanation for the observed differences. A possible hypothesis is that our measurement of GPU time (compared to wall-clock time) hides some overhead that is usually reduced by larger batch sizes.

Shorter training times directly translate into higher utilization rates since training time is part of the denominator.

Backward-forward pass ratios:

From the detailed analysis of profiler_nvtx (see above), we estimate that the backward pass uses 2x as many FLOP as the forward pass (there will be a second post on comparing backward/forward ratios in more detail). [OpenAI has also used a ratio of 2](#) in the past.

We wanted to further test this ratio empirically. To check consistency we tested these ratios for a single forward pass with batch size one (one) an entire batch (batch) and an entire epoch (epoch).



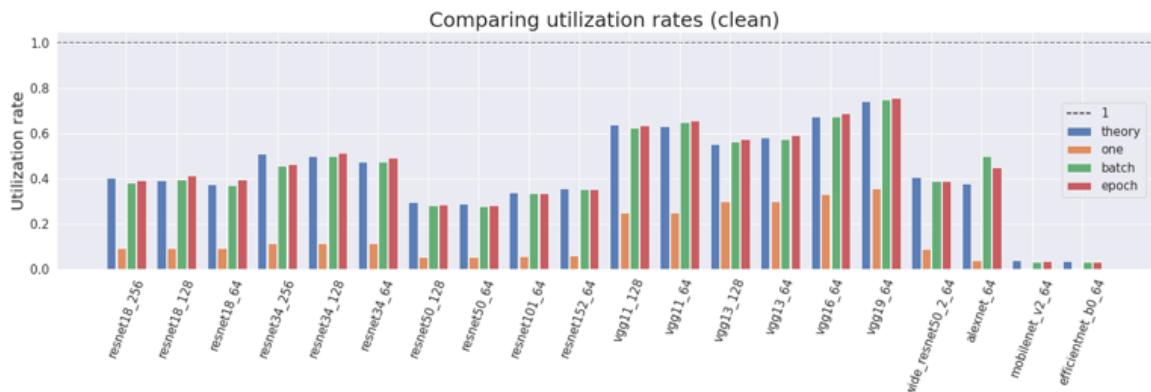
We find that the empirical backward/forward ratios are mostly around the 2:1 mark. Some of the exceptions are likely due to undercounting, i.e. profiler_nvtx just not registering an operation as discussed above.

We assume that the outliers in mobilenet and efficientnet come from the profiler incorrectly measuring FLOP for inverted bottleneck layers.

Utilization rates:

Ultimately, we want to estimate utilization rates. We compute them by using four different methods:

- Theory method: We get the forward pass FLOP estimate of fvcare and multiply it by 3.0 to account for the backward pass. Then, we divide it by the product of the GPU training time and the peak GPU performance of the Tesla P100.
- One method: We take the profiler_nvtx estimate for the forward and backward passes, and divide it by the product of the training time and maximal GPU performance.
- Batch method: We perform the same procedure for one batch.
- Epoch method: We perform the same procedure for one epoch.



We can see that the utilization rates predicted by the theory are often comparable to the empirical measurements for batch and epoch. We can also see that the batch and epoch

versions are usually very comparable while just forwarding and backwarding one sample is much less efficient. This is expected since the reason for larger batch sizes is that they utilize the GPU more efficiently.

Most realistic utilization rates are between 0.3 and 0.75. Interestingly (and ironically), the least efficient utilization rates come from efficientnet and mobilenet which have low values in all approaches. We assume that the outliers come from the profiler incorrectly measuring FLOP for inverted bottleneck layers.

Conclusion:

We use different methods to compute the utilization rate of multiple NN architectures. We find that most values lie between 0.3 and 0.75 and are consistent between approaches. Mobilenet and efficientnet pose two outliers to this rule with low utilization rates around 0.04. We assume that the outliers come from the profiler incorrectly measuring FLOP for inverted bottleneck layers.

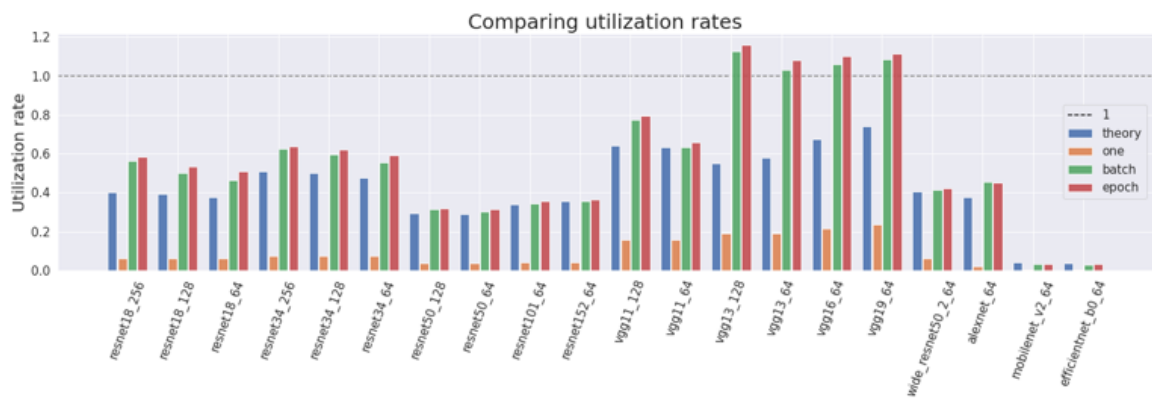
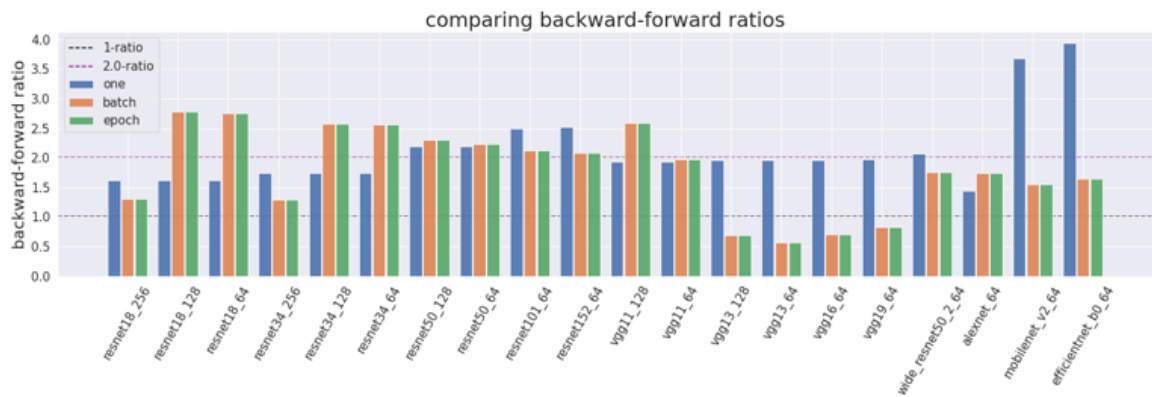
Appendix:

We tried to run [dlprof](#) since it looks like one possible solution to the issues with the profiler we are currently using. However, we were unable to install it since installing [dlprof with pip](#) (as is recommended in the instructions) always threw errors in Colab. I installed dlprof on another computer and wasn't able to get FLOP information from it.

Original versions of the main figures:

These versions are done without accounting for double counting. Thus, the results are wrong. We want to show them to allow readers to compare them to the cleaned-up versions.





Projecting compute trends in Machine Learning

Crossposted from the [AI Alignment Forum](#). May contain more technical jargon than usual.

Summary

Using [our dataset](#) of milestone Machine Learning models, and [our recent analysis of compute trends in ML](#), we project forward 70 years worth of trends in the amount of compute used to train Machine Learning models. Our simulations account for (a) uncertainty in estimates of the growth rates in compute usage during the Deep Learning (DL)-era and Pre-DL era, and (b) uncertainty over the ‘reversion date’, i.e. the date when the current DL-era compute trend (with a ~6 month doubling time) will end and revert to the historically more common trend associated with Moore’s law. Assuming a reversion date of between 8 to 18 years, and without accounting for algorithmic progress, our projections suggest that the median of [Cotra 2020](#)’s biological anchors may be surpassed around August 2046 [95% CI: Jun 2039, Jul 2060]. This suggests that historical rates of compute scaling, if sustained briefly (relative to how long these trends have been around so far), could result in the emergence of transformative models.

Our work can be replicated using [this Colab notebook](#).

Note: we present projections, not predictions. Our post answers the question of: “*What would historical trends over the past 70 years when naively extrapolated forward imply about the future of ML compute?*” It does not answer the question: “*What should our all-things-considered best guess be about how much compute we should expect will be used in future ML experiments?*”

Introduction

Recently, we put together [a dataset](#) of over a hundred milestone Machine Learning models, spanning from 1952 to today, annotated with the compute required to train them. Using this data, we produce simple projections of the amount of compute that might be used to train future ML systems.

The question of how much compute we might have available to train ML systems has received some attention in the past, most notably in Cotra’s Biological Anchors report. Cotra’s report investigates TAI timelines by analyzing: (i) the training compute required for the final training run of a transformative model (using biological anchors), and (ii) the amount of effective compute available at year Y. This article replaces (ii) the compute estimate by projecting 70 years worth of trends in the amount of compute used to train Machine Learning models.

Cotra’s amount of effective compute available at year Y is broken down into forecasts of (a) compute cost, (b) compute spending, and (c) algorithmic progress. By contrast, we do not decompose the estimate, and rather project it on our previous investigation of training compute of ML milestone systems. This trend includes the willingness to spend over time including the reduced compute costs over time; however, it does not address algorithmic progress. We explicitly do not forecast the cost of compute or compute spending.

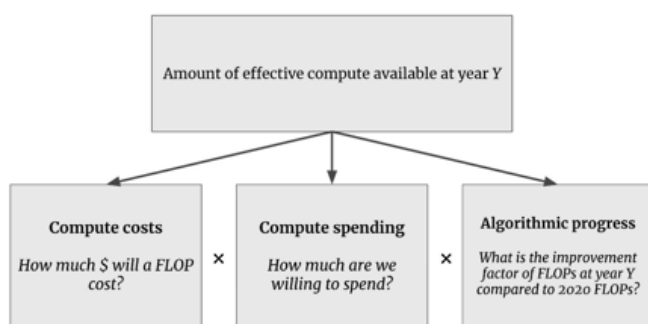


Figure 1a: Cotra's original composition of the compute estimate

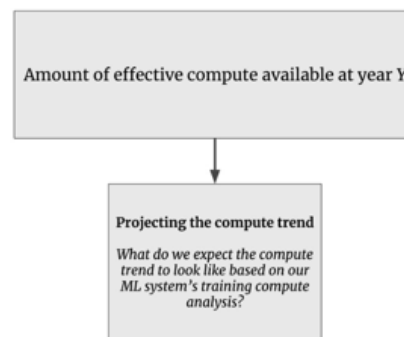


Figure 1b: Our compute estimates

Figure 1. Contrasting our work with that of [Cotra 2020](#)

In this post, we present projections based on previously observed trends and some basic insights about how long the current 6-month doubling time can be sustained. That is, our post answers the question of: what would current trends imply about the future if you naively extrapolate them forwards.

One key reason we don't expect these projections to be particularly good predictions is that it seems likely that Moore's law might break down in some important way over the next few decades. We therefore might expect that the doubling-time in compute usage, when the dollar-budgets to scale compute grow at the economic growth-rate, will be substantially longer than the historically common ~20-month doubling period.

When will the current scaling trend revert back to Moore's law?

In our recent analysis of compute trends in ML ([Sevilla et al., 2022](#)), we find that, since the advent of Deep Learning, the amount of compute used to train ML systems has been doubling every 6 months. This is much faster than the previous historical doubling time that we find to be roughly 20 months (which is roughly in line with Moore's law). Previous work ([Carey, 2018](#), and [Lohn and Musser, 2022](#)) has pointed out that a scaling-rate that outstrips Moore's law by a wide margin cannot be sustained for many years as a rate of growth in ML compute spending that far exceeds economic growth cannot be sustained for many years.

A key question, then, for projecting compute used in future ML systems, is: How long can the current fast trend continue, before it reverts to the historically much more common trend associated with Moore's law?

To answer this question, we replicate the analysis by [Carey, 2018](#), but instead of using the numbers from OpenAI's AI and Compute ([Amodei and Hernandez, 2018](#)), we use the numbers from [our recent analysis \(summary\)](#).^[1] This analysis, roughly, points to three scenarios:

- **Bearish:** slow compute cost-performance improvements and very little specialized hardware improvements. In this scenario, it takes 12 years for the cost of computation to fall by an OOM. The current 6-month doubling period can be maintained for another ~8 years.
- **Middle of the road:** Moderate compute cost-performance improvements and moderate improvements in specialized computing. In this scenario, it takes roughly 7 years for the cost of computation to fall by an OOM, and progress in specialized hardware helps sustain the trend ~3 additional years. The current 6-month doubling period can be maintained for another ~12 years.

- **Bullish:** Fast compute cost-performance improvements and substantial improvements in specialized computing. In this scenario, it takes 4 years for the cost of computation to fall by an OOM, and progress in specialized hardware helps sustain the trend ~6 additional years. The current 6-month doubling period can be maintained for another ~18 years.

Roughly, we might say that these scenarios are represented by the following distributions over 'reversion dates', i.e. dates when the scaling trends are more similar to Moore's law than they are to the current fast trend.

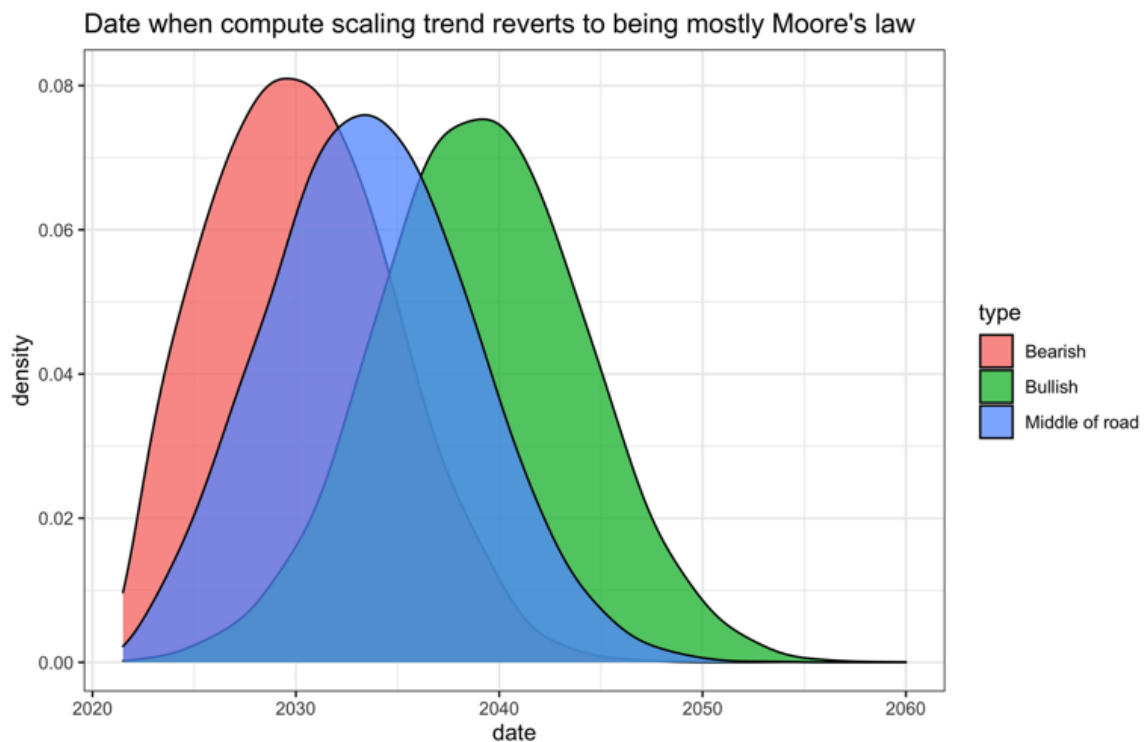


Fig 2. Distributions that roughly correspond to the three scenarios that come out of our replication of [Carey, 2018.^{\[1\]}](#)

We then produce a mixture of these distributions by creating a weighted linear pool where "Bearish" is assigned 0.75, "Middle of the road" is assigned 0.20, and "Bullish" 0.05, based on our best-guesses (you can apply your own weights using [this Colab notebook.](#))

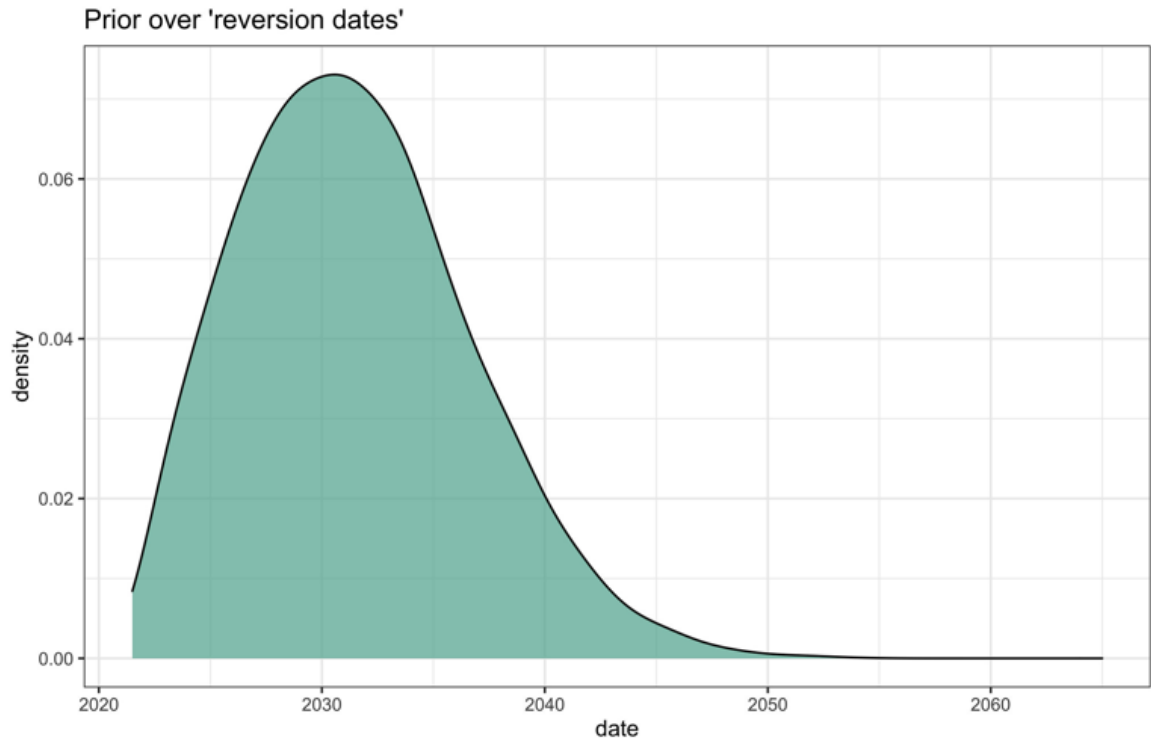


Fig 3. our best-guess for a prior over reversion dates, formed by mixing the previous distributions

We can use this as our prior over when the fast-trend will revert to the more historically common trend associated with Moore's law.

Projecting ML compute trends

We simulate compute paths based on (a) our estimates of the growth rates in compute usage during the DL-era and Pre-DL era, and (b) our prior over 'reversion date', i.e. the date when the current DL-era compute trend will end. We account for the uncertainty in both (a) and (b) in our simulations (see details [here](#)).

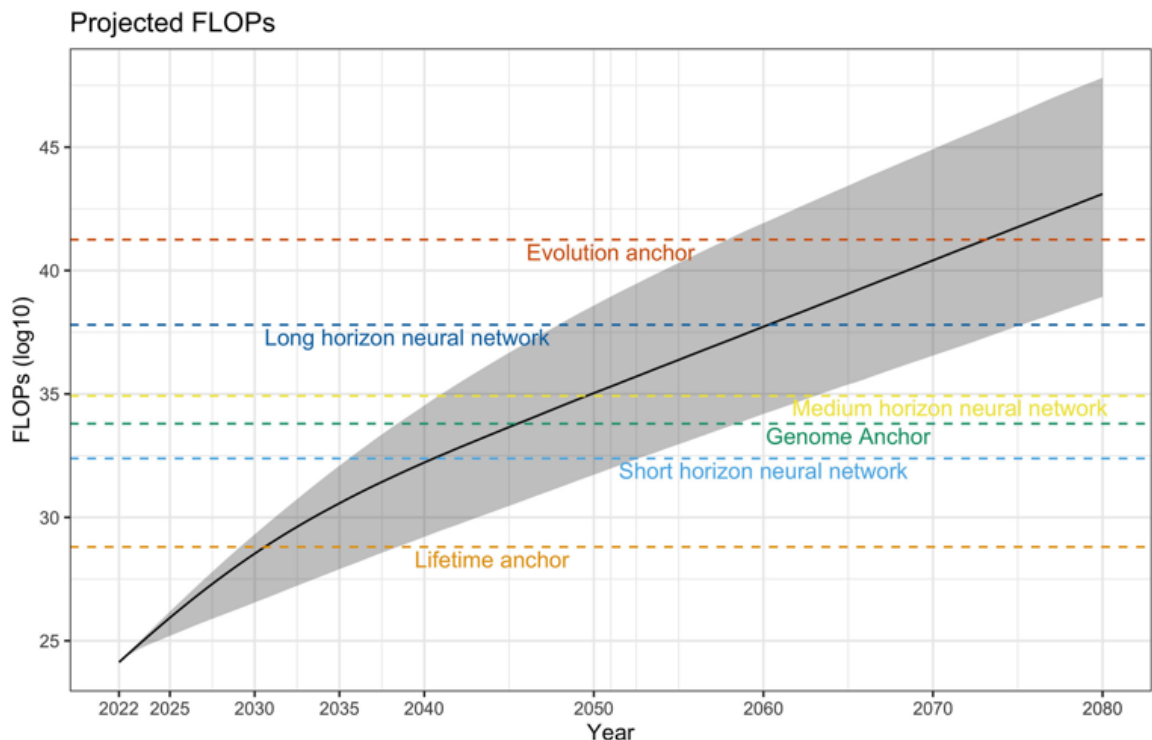


Fig 4. 10,000 projected compute paths. Solid line represents the median projected compute at each date, and the shaded region represents 2-standard deviations around the median.

Our simulations reveal the following projections about the amount of compute used to train ML models.

Year	Projected FLOPs used to train largest ML model	Enough for how many anchor's median compute requirements?
2025	$10^{25.90}$ [$10^{25.33}$, $10^{26.14}$]	0/6
2030	$10^{28.67}$ [$10^{26.71}$, $10^{29.47}$]	0/6
2040	$10^{32.42}$ [$10^{29.27}$, $10^{34.71}$]	1/6
2050	$10^{35.26}$ [$10^{31.78}$, $10^{38.86}$]	3/6
2060	$10^{38.10}$ [$10^{34.35}$, $10^{42.49}$]	5/6
2070	$10^{40.79}$ [$10^{36.83}$, $10^{45.49}$]	5/6
2080	$10^{43.32}$ [$10^{39.04}$, $10^{48.18}$]	6/6

Table 1: Projected FLOPs from 2025 to 2080

These projections suggest that, without accounting for algorithmic progress, the most modest of [Cotra 2020](#)'s biological anchors will be surpassed around August 2030 [95% CI: Jan 2029, May 2038], the median anchor ($\sim 10^{34.36}$ FLOPS) will be surpassed around August 2046 [95% CI: Jun 2039, Jul 2060], and the strongest of anchors will be surpassed around May 2072 [95% CI: Jan 2057, Jun 2089].

Conclusion

If we naively extrapolate the trends uncovered from 70-years worth of compute scaling in Machine Learning, we find that within roughly 25 years, large-scale ML experiments will use amounts of compute that exceed the half of the compute budgets that [Cotra 2020](#) has suggested may be sufficient for training a transformative model. This highlights the fact that historical rates of compute scaling in Machine Learning, even if sustained relatively briefly (relative to how long these trends have been around so far), could place us in novel territory where it might be likely that transformative systems would be trained. This work also suggests that understanding compute trends might be a promising direction for predicting ML progress,

Details of the simulations

We assume compute grows exponentially in time at some rate g :

$$C(t) = C(0) e^{gt}, \quad \text{where } t \geq 0.$$

In our projections, we replace g with g^* , defined as a weighted geometric mean of our best-guess of the growth rate during Moore's law (\tilde{g}_M), and the growth rate of our estimate of the growth rate during the Deep-Learning Era (\hat{g}_{DL}):

$$g^* = \hat{g}_{DL}^{w(t)} \tilde{g}_M^{1-w(t)}, \quad \text{where } w(t) \in [0, 1].$$

Here, \hat{g}_{DL} simply denotes the growth rate during the Deep Learning Era (2010 onwards) as estimated using OLS. In particular, we estimate the following model using our dataset:

$$\log C(t) = \beta + g_{DL} t, \quad \text{where } t > 2010.$$

\tilde{g}_M is defined as follows:

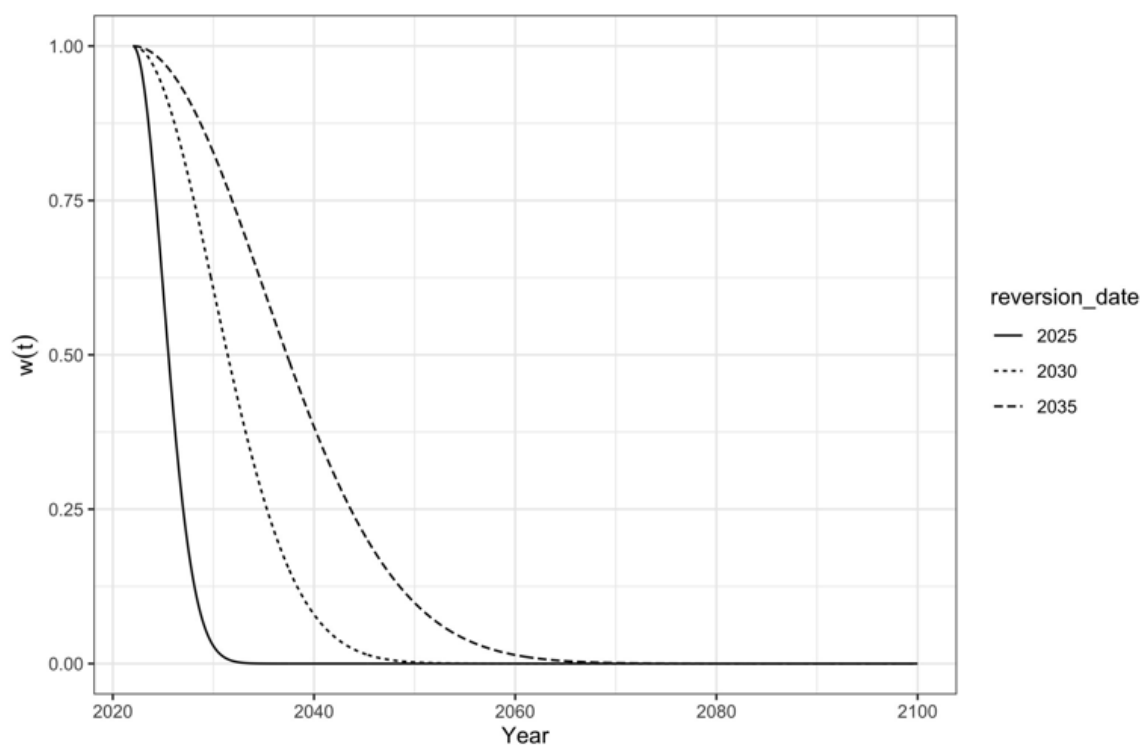
$$\tilde{g}_M = \sqrt{\hat{g}_M g_{20\text{-month}}},$$

where \hat{g}_M is the estimated growth rate during the Pre-DL era, and $g_{20\text{-month}}$ is the growth rate implied by a 20-month doubling period. The reason we take the geometric mean of the estimated growth rate, and the growth rate implied by a 20-month doubling period is because Moore's law is sufficiently well-established that the error bars around \hat{g}_M are too large relative to how well-established Moore's law is. We therefore artificially increase our precision of the growth rate associated with Moore's law by taking an average of our estimated value and the usual growth rate implied by an ~ 20 -month doubling-time.

Our weight function, $w(t)$, is constructed as follows:

$$w(t) = \exp\left(-\frac{(t - \text{reversion_date})^2}{2}\right)$$

Why? Well, it's a logistic-like function with a unit-interval range, which exceeds 1/2 when $t < \text{reversion date}$, equals 1/2 when $t = \text{reversion date}$, and is less than 1/2 otherwise. This is what it looks like:



We then simulate some path C^j as follows:

$$C_j = C(2022) e^{g_j^* t}, \text{ where, for any } j :$$

- \hat{g}_{DL} is estimated on our randomly sampled (with replacement) DL-Era Data,
- \hat{g}_M is estimated on our randomly sampled (with replacement) Pre-DL Era data, and
- $w(t)$ is set based on a randomly sampled reversion date from our prior over reversion dates.

1. \hat{C}

You can find the details of this analysis and a comparison to Carey's results [here](#).

Compute Trends — Comparison to OpenAI's AI and Compute

This is a slightly modified version of Appendix E from our paper "*Compute Trends Across Three Eras of Machine Learning*". You can find the summary [here](#) and the complete paper [here](#).

After sharing our updated compute trends analysis a common question was: "So how are your findings different from [OpenAI's previous compute analysis](#) by Amodai and Hernandez?". We try to answer this question in this short post.

Comparison to OpenAI's AI and Compute

[OpenAI's analysis](#) shows a 3.4 month doubling from 2012 to 2018. Our analysis suggests a 5.7 month doubling time from 2012 to 2022 (Table 1, *purple*). In this post, we investigate this difference. We use the same methods for estimating compute for the final training run. Our methods are described in detail in [Estimating training compute of Deep Learning models](#).

Our analysis differs in three points: (I) number of samples, (II) extended time period, and (III) the identification of a distinct large-scale trend. Of these, either the time period or the separation of the large-scale models is enough to explain the difference between our results.

To show this, we investigate the same period as in the OpenAI dataset. The period starts with AlexNet in September 2012 and ends with AlphaZero in December 2018.

As discussed, our work suggests that between 2015 and 2017 a new trend emerged — the Large-Scale Era. We discuss two scenarios: (1) assuming our distinction into two trends and (2) assuming there is a single trend (similar to OpenAI's analysis).

Period	Data	Scale (FLOPs)	Slope	Doubling time	R ²
AlexNet to AlphaZero 09-2012 to 12-2017	All models (n=31)	1e+16 / 1e+21	1.0 OOMs/year [0.6 ; 1.0 ; 1.3]	3.7 months [2.8 ; 3.7 ; 6.2]	0.48
	Regular scale (n=24)	2e+16 / 1e+20	0.8 OOMs/year [0.5 ; 0.8 ; 1.1]	4.5 months [3.2 ; 4.3 ; 7.8]	0.48

Period	Data	Scale (FLOPs)	Slope	Doubling time	R ²
AlphaGo Fan to AlphaZero 09-2015 to 12-2017	Large scale (n=7)	2e+17 / 3e+23	1.2 OOMs/year [1.0 ; 1.3 ; 1.8]	3.0 months [2.1 ; 2.9 ; 3.5]	0.95
AlphaZero to present 12-2017 to 02-2022	All models (n=62)	5e+19 / 1e+23	0.8 OOMs/year [0.5 ; 0.8 ; 1.1]	4.5 months [3.3 ; 4.4 ; 7.1]	0.36
	Regular scale (n=47)	2e+19 / 3e+22	0.9 OOMs/year [0.6 ; 0.9 ; 1.2]	4.2 months [3.1 ; 4.2 ; 6.0]	0.46
	Large scale (n=15)	1e+22 / 6e+23	0.4 OOMs/year [0.3 ; 0.4 ; 0.7]	8.7 months [5.4 ; 8.7 ; 14.6]	0.68
AlexNet to present 09-2012 to 02-2022	All models (n=93)	8e+16 / 7e+22	0.6 OOMs/year [0.5 ; 0.6 ; 0.7]	5.7 months [4.9 ; 5.7 ; 6.8]	0.60
	Regular scale (n=72)	4e+16 / 2e+22	0.6 OOMs/year [0.5 ; 0.6 ; 0.7]	5.7 months [5.0 ; 5.7 ; 6.8]	0.69
AlphaGo Fan to present 12-2017 to 02-2022	Large scale (n=19)	4e+21 / 6e+23	0.3 OOMs/year [0.1 ; 0.3 ; 0.5]	10.7 months [7.8 ; 10.7 ; 27.2]	0.66

Table 1: Trendline data over the same period as OpenAI's analysis, partitioned around the release of three landmark models: AlexNet, AlphaGo Fan, and AlphaZero.

We can interpret these results in two ways:

1. There is a single trend, which showed a 3.7 month doubling time between September 2012 and December 2017 (Table 1, *red*). Afterward, the trend slowed down to a 4.5 month doubling time (Table 1, *yellow*).
2. A new trend of large-scale models split off the main trend in late 2015. If we separate the large-scale models, we can see that the regular-scale trend had a similar doubling time before and after 2017 (4.5 and 4.2 months; Table 1, *green* and *blue*). OpenAI's result is different from ours because they are mixing the regular-scale and large-scale trends.

In the first interpretation, our result is different from OpenAI as we are grouping together the pre-2017 and post-2017 trends into a single analysis.

In the second interpretation, our result is different because we are analyzing the trend in large-scale and regular-scale models differently.

We currently favor the second explanation. This is because (a) the large-scale trend story seems to better predict developments after 2017, while [Lyzhov found](#) that the single-trend story does not extend past 2017, and (b) we think that the models in the large-scale trend are explained by a drastic departure in funding (see [Appendix F](#) for a discussion if large-scale models are a different category and various caveats of this interpretation).

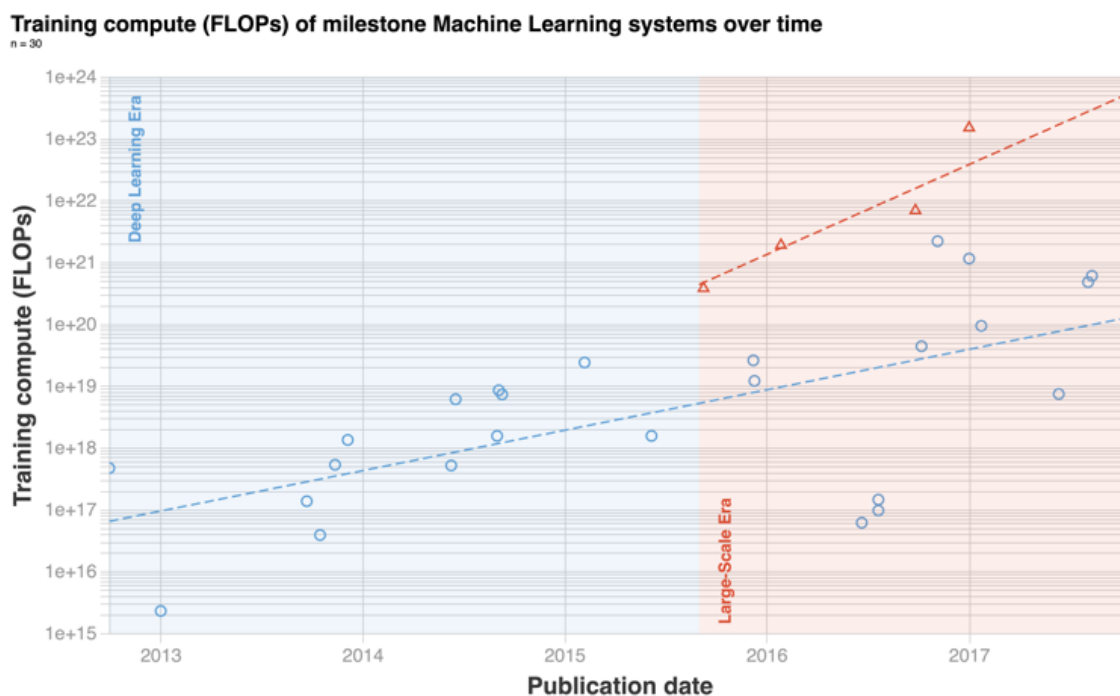


Figure 1: Visualization of our dataset with the two distinct trends in the same time period OpenAI's analysis.