On Evaluating and Improving the Efficiency of Deep Networks

Davis W. Blalock[1], Michael Carbin[1,2], Laura Florescu[1], Jonathan Frankle[1,2], Matthew L. Leavitt[1], Tyler Lee[1], Moin Nadeem[1], Jacob Portes[1]*, Naveen Rao[1], Landan Seguin[1], Cory Stephenson[1], Hanlin Tang[1], Abhinav Venigalla[1] †
[1]MosaicML, San Francisco, CA
[2]Massachusetts Institute of Technology, Cambridge, MA
{first_name}@mosaicml.com

Abstract

In this work we articulate our methodology for evaluating and composing algorithmic improvements in training efficiency of deep learning models. We first show that by using our library of efficiency methods, it is possible to achieve training speedups and cost reductions of 2.9X on ResNet-50 on ImageNet, 3.5X on ResNet-101 on ImageNet, and 1.7X on GPT-125 language models. Some of these methods are widely used, some were developed *de novo*, but many have not been evaluated in the context of efficiency. We frame our evaluation methodology for these methods by first defining the two quantities necessary to measure efficiency: model quality (e.g. accuracy) and resources (e.g. training time, monetary cost to train). Motivated by the observation that *quality and resources are typically interchangeable*, we emphasize the importance of evaluating methods for improving efficiency by examining tradeoffs between resource usage and model quality. From that lens, we define an *efficiency improvement* as either achieving the same model quality while utilizing fewer resources or achieving higher model quality while utilizing the same amount of resources, relative to a baseline model. While our results demonstrate that it is possible to find significant efficiency improvements, these improvements result from the combination of several efficiency methods, that jointly induce complex interactions between their behavior and the underlying hardware. We term the analysis and navigation of these complexities as the *composition problem*. Our results show that the broader machine learning community has developed a powerful set of methods that when methodically evaluated and composed offer the promise of dramatically reduced time, cost, and resources for deep learning.

1 Introduction

At MosaicML, our mission is to reduce the cost of training neural networks. We do so by modifying the training process at the algorithmic level: changing the details of the training recipe. We believe that the existing way of training neural networks is inefficient but that, by studying how neural networks behave in practice, we can leverage those findings to develop better training algorithms and significantly reduce the cost of training.

We develop methods that modify the training procedure to achieve better tradeoffs between the final model quality and the time or cost to train the model. The end result is that, to train a model to a particular level of quality, we can get there in less time or for fewer dollars. These methods change the training procedure in a variety of ways: for example, altering the data or the order in which they are presented to the model; tweaking the structure of the model; and changing the way forward propagation and back propagation take place.

[*] Work performed as part of the MosaicML Internship Program

[†] Authors listed in alphabetical order

Over the past few months, we have pursued this approach on common settings for computer vision—ResNets (He et al., 2016) on ImageNet (Deng et al., 2009), UNet Ronneberger et al. (2015) on BraTS (Baid et al., 2021)—and natural language processing—Transformer/GPT language models (Radford et al., 2019). We carefully evaluated 20 different methods and how they compose, and we implemented them in our Composer library. The compositions in this library have allowed us to achieve training speedups and cost reductions of 2.9X on ResNet-50 on ImageNet, 3.5X on ResNet-101 on ImageNet, and 1.7X on the GPT-125 language models (as compared to the optimized baselines on 8xA100s on AWS), all while achieving the same model quality as the baselines. To make sure that these results reflect fair comparisons, all of these data come from training on a fixed hardware configuration on publicly available clouds, and none of these methods increase the cost of inference.

These results demonstrate the enormous opportunity to accelerate neural network training across a wide variety of settings. In the coming months, we will continue to grow our open-source library of methods and work with customers to reduce the time and cost of their existing workloads and take on more ambitious machine learning problems. We are excited to share our ideas, tools, and infrastructure with the community, and we are eager for feedback and collaboration as we pursue our mission to make deep learning efficient for everyone. Don't hesitate to reach out if you're interested in getting involved as a contributor, collaborator, customer, or - especially - as a researcher, research engineer, or intern on our team.

2 The Science of Algorithmic Improvements to Training

Our goal in modifying the training algorithm is to improve the efficiency of training. We start with a baseline training algorithm and compare it to training with one or more methods. To evaluate the result, we typically ask one of the following questions:

- What is the fastest or cheapest way that we can train a particular model while maintaining the same quality as the baseline? For example, if the baseline ResNet-50 reaches 76.6% top-1 accuracy on ImageNet, can we reach this accuracy in less time or for less money than the baseline?
- What is the highest quality we can reach while maintaining the same time or cost budget? For example, if the baseline ResNet-50 takes four hours to reach 76.6% top-1 accuracy, can we reach a higher quality model while investing those same four hours?

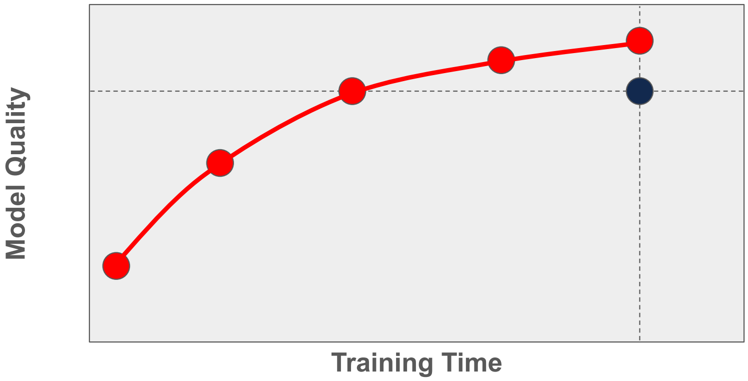

Figure 1: Improving efficiency can mean achieving the same model quality (y-axis; e.g. accuracy) with fewer resources (e.g. training time or money; x-axis), or achieving higher model quality with the same resources. The black point denotes a baseline model, while the red points denote a more efficient training scheme, enacted for different durations of training.

2.1 Measuring Success

We measure three quantities when developing improvements to training efficiency:

- *Quality:* Metrics that characterize the quality of the model. For example, accuracy in image classification and perplexity in language modeling.
- *Training Time:* The amount of time necessary to train the model on a given system configuration.
- *Training Cost:* The amount of money necessary to train the model on a given system configuration.

There are many other quantities that we might wish to measure. For example, it is important to understand the effects that methods have on the time and cost required for inference (using the model after it is trained). For simplicity in our top-line numbers, we only consider changes to the training procedure that have a negligible impact on inference time and cost. For example, we do not modify the model architecture in significant ways, nor do we report numbers here that reflect comparisons across different model sizes. These constraints make it possible to isolate and characterize how changes to the training procedure affect the three metrics above.

In addition, there are many other inputs that could affect the time and cost of training, for instance, varying the hardware platform. In our experience, using an NVIDIA T4 rather than an NVIDIA A100 significantly increases training times, but it can also reduce training costs because the cost per hour for a T4 is often significantly lower than the cost per hour for an A100 on public clouds. For simplicity in our top-line numbers, we only make comparisons between models trained on identical hardware. In doing so, all improvements in training time lead to equivalent improvements in training cost.

For ease of interpreting the top-line numbers we report here, we make the simplifying assumption that inference costs and hardware configurations are fixed. In the Explorer - which includes thousands of data points we have collected on different combinations of models, hardware platforms, and methods - you can go beyond these constraints. For example, you can make comparisons across different model sizes (e.g., ResNet-50 vs. ResNet-101) and hardware platforms (e.g., A100s vs. V100s vs. T4s and Amazon Web Services vs. Google Cloud Platform) in order to find the most cost-effective training configuration.

2.2 Evaluating Methods by Examining Tradeoffs

Any method can have two effects on training a model:

- It can affect model quality, either positively or negatively.
- It can affect training time, either positively or negatively. Training time can then be translated into training cost. Since we focus on fixed hardware here, all improvements to training speed lead to commensurate improvements in training cost.

If a method improves quality at the same or better training time (or improves training time at the same or better quality), then this method is strictly better than the baseline and using it is clearly worthwhile (Figure 2).

But what if, instead, a method improves training time but reduces quality, or improves quality but worsens training time (Figure 3)? How does one decide whether to use such a method?

To answer this question, we evaluate the tradeoff between model quality and the available training budget (as measured in training time or cost). Consider a particular model, say ResNet-50 on ImageNet. In its standard configuration, we train for 90 epochs and reach 76.6% top-1 accuracy. But who's to say that this is the correct configuration? If we train for 45 epochs, we will cut training time in half, but quality will decrease. If we train for 180 epochs, we will achieve higher quality, but training time will increase. No single one of these configurations is "right" per se, nor does any single data point accurately capture this time-quality tradeoff.

Instead, we describe tradeoffs in model training efficiency using a collection of points on a *tradeoff curve* that represent the quality attained for a given training budget (Figure 4). Ultimately, the researcher or engineer training the model must choose which point on a tradeoff curve best suits his or her goals, and only by showing the full collection of points may domain experts make informed decisions.

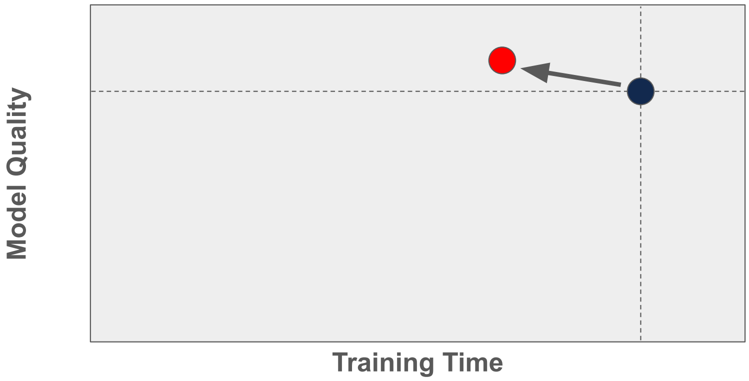

Figure 2: The model to which efficiency interventions have been applied (red) improves both training time (x-axis) and model quality (y-axis) compared to the baseline model (black).

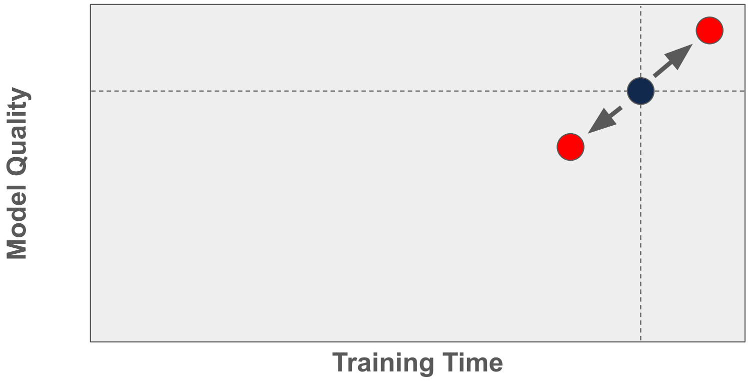

Figure 3: Applying a method to a model (red) could improve training time (x-axis) at the cost of model quality (y-axis), or improve model quality at the cost of training time compared to a baseline model (black). In either of these scenarios, determining whether the method constitutes and efficiency improvement requires examining a *tradeoff curve* (Figures 4 and 5).

Let's walk through an example tradeoff curve (Figure 4). On the x-axis, we plot the amount of time or money necessary to train a model. On the y-axis, we plot the quality of this model. For a given training configuration, we consider a wide range of training budgets. For example, on Resnet-50 on ImageNet, we train for 90, 72, 60, 54, 30, and 22.5 epochs (representing speedups and cost reductions of 1x, 1.25x, 1.5x, 2x, 3x, and 4x). When we scale the training time in this way, we correspondingly dilate the learning rate schedule. We then evaluate the resulting quality of these models and plot the points. The curve connecting these points characterizes the quality-speed tradeoff, shown in blue above.

Now, let's see the effect of applying a method to modify the training algorithm (Figure 5). When we modify the training configuration, we re-train the model for each of these budgets. This allows us to create a second tradeoff curve (red). To determine whether a method is beneficial, we compare the two tradeoff curves. If the curve moves upwards and to the left, it means that using this method attains a better overall tradeoff between quality and training time than the baseline. Note that, in the example above, the red points are all slower than the corresponding blue points because the method applied on its own costs extra time. However, the improvements in quality are such that the method still leads to a better overall tradeoff: to reach any particular quality, you can train for fewer steps, leading to an overall speedup despite the fact that each step is slightly more expensive.

This brings us to a key point in thinking about efficiency: *time and quality are typically interchangeable*. Most quality improvements can be converted into time improvements by training for

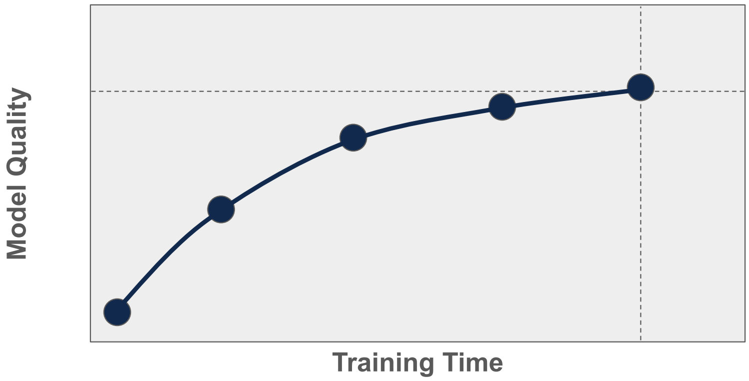

Figure 4: An example tradeoff curve plotting model quality (y-axis) against training time (x-axis).

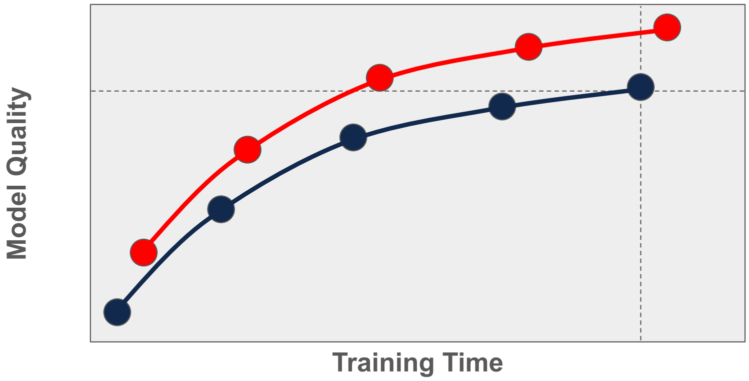

Figure 5: Example tradeoff curves for a baseline (black) and improved (red) model.

fewer steps, and most time improvements can be converted into quality improvements by training for more steps. In this way, regularization, which is typically thought of as a way to improve model quality, can also be used as a speedup method.

One final note: the example above is particularly clean: one curve is above and to the left of the other across the full range of training budgets we consider. This is not always the case; it is possible that different configurations may be optimal depending on the time or cost budget available for training. We have found that certain methods excel when budgets are very small, while others (especially regularization) thrive in larger-budget regimes.

In summary, we think about efficiency of training in terms of tradeoffs between quality and time or cost, and we consider a method beneficial when it makes it possible to reach a better tradeoff curve than the baseline.

2.3 The Costs of Neural Network Training

If one is trying to improve the efficiency of neural network training, it is helpful to understand the steps of the training process and the system resources they utilize. As mentioned in the previous section, methods typically have one or more of the following effects on the cost of training:

1. They reduce the number of optimization steps required for training to reach a particular quality.
2. They reduce the cost of each step in order to make it possible to train for the same number of steps in less time.

Each step of training comprises several operations:

1. Data is loaded off of the local disk or over the network and into main memory.
2. Data is pre-processed (e.g., JPEG decompression, data augmentation, tokenization). This can take place on CPU, an AI accelerator (e.g., a GPU), or some combination of the two.
3. Data is loaded onto the AI accelerator.
4. Data is forward propagated and backward propagated through the network, which involves several steps of loading data from accelerator memory, computing using the data, and writing data back to accelerator memory. This process may be bound by the amount of compute available on the accelerator or the amount of memory bandwidth on the accelerator, and different steps can be bound by one or the other.

These operations are often performed in a pipelined manner, with data loaded for the next step while forward and backward propagation take place for the current step. One of these components will always impose a bottleneck on the overall speed of training. In an ideal world, that bottleneck will be the amount of compute or memory bandwidth on the accelerator, since accelerators are typically the most expensive resources. However, if a system is not tuned properly, it is possible that other steps — for example loading data from disk — may become a bottleneck, hampering the ability to take full advantage of the accelerator and wasting money.

Although our methods typically focus on reducing the cost of forward and backward propagation (Operation 4), they often have side-effects on Operations 1-3 that create new bottlenecks or shift the location of the bottleneck. For example, while the main effect of RandAugment (Cubuk et al., 2019) is regularization that improves model quality and reduces the number of steps necessary to train the model, it requires additional data augmentation that has the side-effect of significantly increasing CPU usage. In fact, in some cases, we found that CPU usage became such a bottleneck that it slowed down training. Similarly, some of our combinations of methods were so successful at speeding up Operation 4 that JPEG decoding on the CPU became a bottleneck, restricting our ability to speed up the model further without developing a more efficient dataloader pipeline.

In summary, it is important to understand how methods improve the efficiency of training and which resource is currently imposing a bottleneck in order to target further interventions. This is a basic tenet of computer system optimization, and it is important not to lose sight of it in the midst of the many other considerations present in deep learning.

2.4 Composition: Combining Methods

Thus far, we have only considered the effects of applying a single method to improve training. But what happens when we compose multiple methods? In an ideal world, the effects of different methods would compose linearly or even synergistically, where the benefits of composition are at least as good as the combined benefits of the constituent parts. But composition is complex in practice, and methods interact nonlinearly.

Composition is the cornerstone of MosaicML's research program: it demonstrates how our diverse suite of methods and best practices may be combined together into a usable toolset. It is also the basis of our company name: we address the research problem of assembling the "mosaic" of individual methods proposed in the literature in order to see what the complete picture looks like. We have developed several strategies to reason about composition.

Central to our thinking about composition is Amdahl's Law, an idea for reasoning about efficiency in computer architectures. In our context, Amdahl's Law (Amdahl, 1967) states that optimizing a portion of a system responsible for N% of the costs can lead to, at most, an N% speedup. For example, assuming there are no other bottlenecks to the training process, back propagation consumes about 2/3 of the cost of training. Thus, any reductions to the cost of back propagation can reduce the cost of training by at most 2/3.

One consequence of Amdahl's law is that there are diminishing returns in optimizing the same part of a system more than once. For example, suppose a method reduces the cost of back propagation by 50%. This will reduce the overall cost of training by 1/3, assuming there are no other bottlenecks in the training process. Because back propagation now comprises less of the training cost than it did before, introducing a second method that also reduces the cost of back propagation by 50% will only reduce the overall cost of training by 1/6 (assuming it composes perfectly with the first method). And if each of these methods also reduces model quality, it is possible that composing these methods could actually lead to a worse tradeoff curve than using either method individually. Early in our research, we encountered this exact problem. We combined two methods that each resulted in improved tradeoffs individually, but when combined led to worse tradeoffs than the baseline model because they were optimizing the same part of the training process.

As a consequence of this experience, we have learned that composition tends to be most successful when the methods affect all parts of training in a balanced way. For example, it is more beneficial to compose a method that reduces the cost of forward propagation with another that reduces the cost of back propagation. Or, alternatively, to compose a method that increases the cost of back propagation but improves quality with another method that reduces the computational cost of each step back to the original level.

Our best results occur when we combine many methods. For example, on ResNet-101, they occur when we combine BlurPool, Channels Last, Label Smoothing, Mixup, Progressive Resizing, and Sharpness-Aware Minimization while training for fewer steps using Scale Schedule. You can use the Explorer to view these combinations

3 Setting Baselines

As the previous section illustrates, an essential part of improving the efficiency of neural network training is choosing appropriate baselines to compare against. One common failure mode in machine learning research is to optimize the configuration of a proposed method without optimizing the corresponding baseline, potentially overestimating the value of the method. At MosaicML, we take great care to ensure that our baselines are as strong as possible, presenting the most difficult possible scenario in which to improve training time, training cost, or model quality.

Our baselines reflect standard best practices for efficient training:

- We use mixed precision training.
- We have tuned the data loading pipeline to maximize throughput.
- We have performed hyperparameter search across batch size and learning rate.
- We use a step-wise (rather than epoch-wise) learning rate schedule, which we have found to improve quality across our range of settings.
- Rather than select a specific training budget, we produce tradeoff curves reflecting a range of possible training budgets to ensure that our baselines reflect the minimum possible budget to reach a particular level of quality.
- We use decoupled weight decay (Loshchilov & Hutter, 2017) to ensure that the level of weight decay is appropriate for a range of batch sizes and learning rates.
- On GPT models, we have been guided by scaling laws (Kaplan et al., 2020) in choosing our model sizes and hyperparameters, but we have explored around these hyperparameters to ensure that our choices are as strong as possible.

These baselines are challenging. Our ability to still reduce time, reduce cost, and improve quality in comparison to these baselines reflects the immense opportunity to improve the efficiency of training algorithmically.

These baselines reflect a significant amount of tuning. We recognize that many research and industrial models may not have been subjected to this level of tuning. In order to provide context for those scenarios, several of our benchmarks in the Explorer also include *unoptimized* baselines that use mixed precision training and tuned hyperparameters but no other enhancements.

4 Data Collection

To evaluate our methods, we collected data on thousands of combinations of models, datasets, methods, and hardware platforms. You can view these results in the Explorer. In this section, we provide the system details for how we collected this data so that you can replicate our results.

We collected all of our data on the publicly available Amazon (AWS) and Google (GCP) clouds. We determined the quality of each combination of model, dataset, and methods by training on a single hardware configuration (8xA100 on AWS - p4d.24xlarge). We determined the time and cost required for conducting this training run on all other hardware configuration by profiling the training throughput and extrapolating the time that would have been required for the full training run. For methods that behave uniformly throughout training, we profiled the initial epochs of training and extrapolated. For methods that change their behavior and throughput over the course of training according to a schedule, we profiled relevant points in training and extrapolated.

Performance and quality were obtained with our MosaicML Composer open-source library using the Composer Docker Image. The Docker Image is built with the following software dependencies: Ubuntu 18.04, Pytorch v1.9.0, Python 3.8.0. CUDA 11.1.1.

AWS Configuration

Image: EKS Optimized AMI Version 1.19 with GPU Support

/aws/service/eks/optimized-ami/1.19/amazon-linux-2-gpu/recommended/image_id

CUDA Driver:460.73.01

For more setup information, see here

GCP Configuration

Image: COS with ContainerD, Version 1.18.16-gke.502

CUDA Driver: 450.119.04

For more setup information, see here and here

References

Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 978-1-4503-7895-6. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560. event-place: Atlantic City, New Jersey.

Ujjwal Baid, Satyam Ghodasara, Suyash Mohan, Michel Bilello, Evan Calabrese, Errol Colak, Keyvan Farahani, Jayashree Kalpathy-Cramer, Felipe C. Kitamura, Sarthak Pati, Luciano M. Prevedello, Jeffrey D. Rudie, Chiharu Sako, Russell T. Shinohara, Timothy Bergquist, Rong Chai, James Eddy, Julia Elliott, Walter Reade, Thomas Schaffter, Thomas Yu, Jiaxin Zheng, Ahmed W. Moawad, Luiz Otavio Coelho, Olivia McDonnell, Elka Miller, Fanny E. Moron, Mark C. Oswood, Robert Y. Shih, Loizos Siakallis, Yulia Bronstein, James R. Mason, Anthony F. Miller, Gagandeep Choudhary, Aanchal Agarwal, Cristina H. Besada, Jamal J. Derakhshan, Mariana C. Diogo, Daniel D. Do-Dai, Luciano Farage, John L. Go, Mohiuddin Hadi, Virginia B. Hill, Michael Iv, David Joyner, Christie Lincoln, Eyal Lotan, Asako Miyakoshi, Mariana Sanchez-Montano, Jaya Nath, Xuan V. Nguyen, Manal Nicolas-Jilwan, Johanna Ortiz Jimenez, Kerem Ozturk, Bojan D. Petrovic, Chintan Shah, Lubdha M. Shah, Manas Sharma, Onur Simsek, Achint K. Singh, Salil Soman, Volodymyr Statsevych, Brent D. Weinberg, Robert J. Young, Ichiro Ikuta, Amit K. Agarwal, Sword C. Cambron, Richard Silbergleit, Alexandru Dusoi, Alida A. Postma, Laurent Letourneau-Guillon, Gloria J. Guzman Perez-Carrillo, Atin Saha, Neetu Soni, Greg Zaharchuk, Vahe M. Zohrabian, Yingming Chen, Milos M. Cekic, Akm Rahman, Juan E. Small, Varun Sethi, Christos Davatzikos, John Mongan, Christopher Hess, Soonmee Cha, Javier Villanueva-Meyer, John B. Freymann, Justin S. Kirby, Benedikt Wiestler, Priscila Crivellaro, Rivka R. Colen, Aikaterini Kotrotsou, Daniel Marcus, Mikhail Milchenko, Arash Nazeri, Hassan Fathallah-Shaykh, Roland Wiest, Andras Jakab, Marc-Andre Weber, Abhishek Mahajan, Bjoern Menze, Adam E.

Flanders, and Spyridon Bakas. The RSNA-ASNR-MICCAI BraTS 2021 Benchmark on Brain Tumor Segmentation and Radiogenomic Classification. *arXiv:2107.02314 [cs]*, September 2021. URL http://arxiv.org/abs/2107.02314. arXiv: 2107.02314.

Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. RandAugment: Practical automated data augmentation with a reduced search space. *arXiv:1909.13719 [cs]*, November 2019. URL http://arxiv.org/abs/1909.13719. arXiv: 1909.13719.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, June 2009. doi: 10.1109/CVPR.2009.5206848. ISSN: 1063-6919.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. January 2020. URL https://arxiv.org/abs/2001.08361v1.

Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. November 2017. URL https://arxiv.org/abs/1711.05101v3.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi (eds.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, Lecture Notes in Computer Science, pp. 234–241, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24574-4. doi: 10.1007/978-3-319-24574-4_28.