arXiv:2102.12092v2 [cs.CV] 26 Feb 2021

Zero-Shot Text-to-Image Generation

Aditya Ramesh [1] **Mikhail Pavlov** [1] **Gabriel Goh** [1] **Scott Gray** [1]
Chelsea Voss [1] **Alec Radford** [1] **Mark Chen** [1] **Ilya Sutskever** [1]

Abstract

Text-to-image generation has traditionally focused on finding better modeling assumptions for training on a fixed dataset. These assumptions might involve complex architectures, auxiliary losses, or side information such as object part labels or segmentation masks supplied during training. We describe a simple approach for this task based on a transformer that autoregressively models the text and image tokens as a single stream of data. With sufficient data and scale, our approach is competitive with previous domain-specific models when evaluated in a zero-shot fashion.

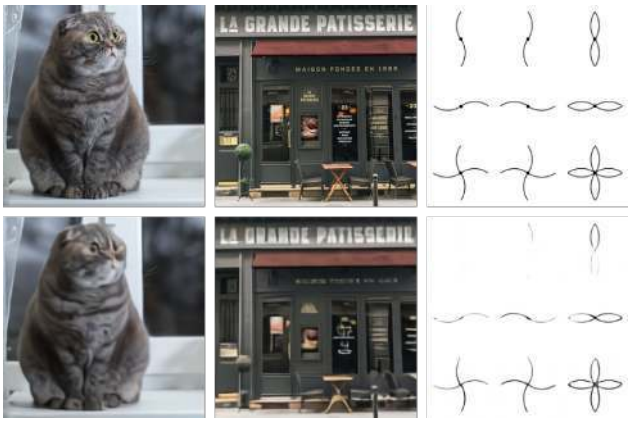


Figure 1. Comparison of original images (top) and reconstructions from the discrete VAE (bottom). The encoder downsamples the spatial resolution by a factor of 8. While details (e.g., the texture of the cat's fur, the writing on the storefront, and the thin lines in the illustration) are sometimes lost or distorted, the main features of the image are still typically recognizable. We use a large vocabulary size of 8192 to mitigate the loss of information.

1. Introduction

Modern machine learning approaches to text to image synthesis started with the work of Mansimov et al. (2015), who showed that the DRAW Gregor et al. (2015) generative model, when extended to condition on image captions, could also generate novel visual scenes. Reed et al. (2016b) later demonstrated that using a generative adversarial network (Goodfellow et al., 2014), rather than a recurrent variational auto-encoder, improved image fidelity. Reed et al. (2016b) showed that this system could not only generate objects with recognizable properties, but also could *zero-shot* generalize to held-out categories.

Over the next few years, progress continued using a combination of methods. These include improving the generative model architecture with modifications like multi-scale generators (Zhang et al., 2017; 2018), integrating attention and auxiliary losses (Xu et al., 2018), and leveraging additional sources of conditioning information beyond just text (Reed et al., 2016a; Li et al., 2019; Koh et al., 2021).

Separately, Nguyen et al. (2017) propose an energy-based framework for conditional image generation that obtained a large improvement in sample quality relative to contemporary methods. Their approach can incorporate pretrained discriminative models, and they show that it is capable of performing text-to-image generation when applied to a captioning model pretrained on MS-COCO. More recently, Cho et al. (2020) also propose a method that involves optimizing the input to a pretrained cross-modal masked language model. While significant increases in visual fidelity have occurred as a result of the work since Mansimov et al. (2015), samples can still suffer from severe artifacts such as object distortion, illogical object placement, or unnatural blending of foreground and background elements.

Recent advances fueled by large-scale generative models suggest a possible route for further improvements. Specifically, when compute, model size, and data are scaled carefully, autoregressive transformers (Vaswani et al., 2017) have achieved impressive results in several domains such as text (Radford et al., 2019), images (Chen et al., 2020), and audio (Dhariwal et al., 2020).

By comparison, text-to-image generation has typically been evaluated on relatively small datasets such as MS-COCO and CUB-200 (Welinder et al., 2010). Could dataset size and model size be the limiting factor of current approaches? In this work, we demonstrate that training a 12-billion parameter autoregressive transformer on 250 million image-text

[1]OpenAI, San Francisco, California, United States. Correspondence to: Aditya Ramesh <_@adityaramesh.com>.

(a) a tapir made of accordion. a tapir with the texture of an accordion.

(b) an illustration of a baby hedgehog in a christmas sweater walking a dog

(c) a neon sign that reads "backprop". a neon sign that reads "backprop". backprop neon sign

(d) the exact same cat on the top as a sketch on the bottom

Figure 2. With varying degrees of reliability, our model appears to be able to combine distinct concepts in plausible ways, create anthropomorphized versions of animals, render text, and perform some types of image-to-image translation.

pairs collected from the internet results in a flexible, high fidelity generative model of images controllable through natural language.

The resulting system achieves high quality image generation on the popular MS-COCO dataset *zero-shot*, without using any of the training labels. It is preferred over prior work trained on the dataset by human evaluators 90% of the time. We also find that it is able to perform complex tasks such as image-to-image translation at a rudimentary level. This previously required custom approaches (Isola et al., 2017), rather emerging as a capability of a single, large generative model.

2. Method

Our goal is to train a transformer (Vaswani et al., 2017) to autoregressively model the text and image tokens as a single stream of data. However, using pixels directly as image tokens would require an inordinate amount of memory for high-resolution images. Likelihood objectives tend to prioritize modeling short-range dependencies between pixels (Salimans et al., 2017), so much of the modeling capacity would be spent capturing high-frequency details instead of the low-frequency structure that makes objects visually recognizable to us.

We address these issues by using a two-stage training procedure, similar to (Oord et al., 2017; Razavi et al., 2019):

- **Stage 1.** We train a discrete variational autoencoder (dVAE)[1] to compress each 256×256 RGB image into a 32×32 grid of image tokens, each element of which can assume 8192 possible values. This reduces the context size of the transformer by a factor of 192 without a large degradation in visual quality (see Figure 1).
- **Stage 2.** We concatenate up to 256 BPE-encoded text tokens with the $32 \times 32 = 1024$ image tokens, and train an autoregressive transformer to model the joint distribution over the text and image tokens.

The overall procedure can be viewed as maximizing the evidence lower bound (ELB) (Kingma & Welling, 2013; Rezende et al., 2014) on the joint likelihood of the model distribution over images x, captions y, and the tokens z for the encoded RGB image. We model this distribution using the factorization $p_{\theta,\psi}(x, y, z) = p_\theta(x \mid y, z)p_\psi(y, z)$, which yields the lower bound

$$\ln p_{\theta,\psi}(x, y) \geqslant \mathop{\mathbb{E}}_{z \sim q_\phi(z \mid x)} \Big(\ln p_\theta(x \mid y, z) - \beta \, D_{\mathrm{KL}}(q_\phi(y, z \mid x), p_\psi(y, z)) \Big), \quad (1)$$

where:

- q_ϕ denotes the distribution over the 32×32 image tokens generated by the dVAE encoder given the RGB image x[2];
- p_θ denotes the distribution over the RGB images generated by the dVAE decoder given the image tokens; and
- p_ψ denotes the joint distribution over the text and image tokens modeled by the transformer.

Note that the bound only holds for $\beta = 1$, while in practice we find it helpful to use larger values (Higgins et al., 2016). The following subsections describe both stages in further detail.[3]

[1] https://github.com/openai/DALL-E

[2] We assume that y is conditionally independent of x given z.

[3] In preliminary experiments on ImageNet (Deng et al., 2009), we attempted to maximize the ELB with respect to ϕ, θ, and ψ jointly, but were unable to improve on two-stage training.

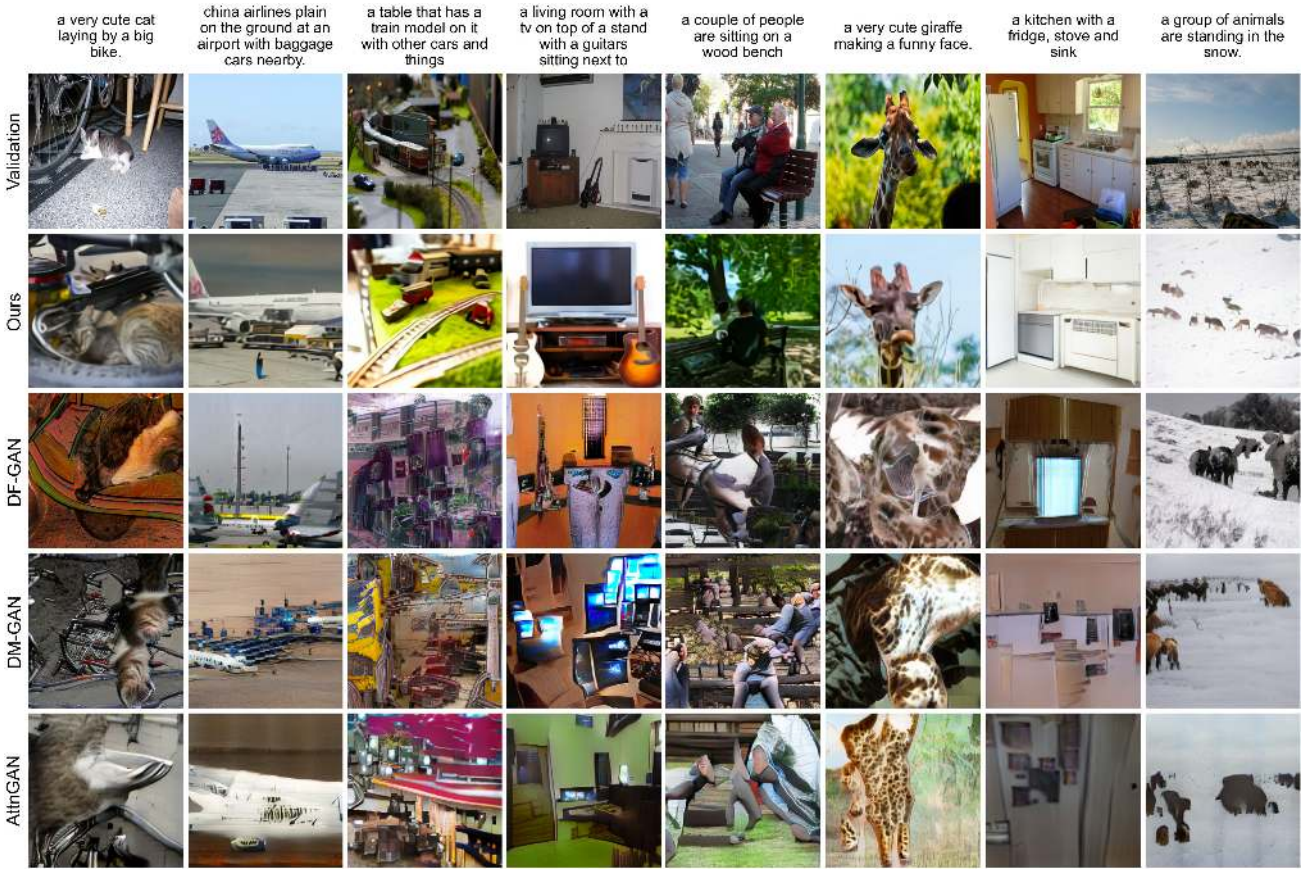


Figure 3. Comparison of samples from our model to those from prior approaches on captions from MS-COCO. Each of our model samples is the best of 512 as ranked by the contrastive model. We do not use any manual cherrypicking with the selection of either the captions or the samples from any of the models.

2.1. Stage One: Learning the Visual Codebook

In the first stage of training, we maximize the ELB with respect to ϕ and θ, which corresponds to training a dVAE on the images alone. We set the initial prior p_ψ to the uniform categorical distribution over the $K = 8192$ codebook vectors, and q_ϕ to be categorical distributions parameterized by the 8192 logits at the same spatial position in the 32×32 grid output by the encoder.

The ELB now becomes difficult to optimize: as q_ψ is a discrete distribution, and we cannot use the reparameterization gradient to maximize it. Oord et al. (2017); Razavi et al. (2019) address this using an online cluster assignment procedure coupled with the straight-through estimator (Bengio et al., 2013). We instead use the gumbel-softmax relaxation (Jang et al., 2016; Maddison et al., 2016), replacing the expectation over q_ϕ with one over q^τ_ϕ, where the relaxation becomes tight as the temperature $\tau \to 0$. The likelihood for p_θ is evaluated using the log-laplace distribution (see Appendix A.3 for a derivation).

The relaxed ELB is maximized using Adam (Kingma & Ba, 2014) with exponentially weighted iterate averaging. Appendix A.2 gives a complete description of the hyperparameters, but we found the following to be especially important for stable training:

- Specific annealing schedules for the relaxation temperature and step size. We found that annealing τ to $1/16$ was sufficient to close the gap between the relaxed validation ELB and the true validation ELB with q_ϕ intsead of q^τ_ϕ.
- The use of 1×1 convolutions at the end of the encoder and the beginning of the decoder. We found that reducing the receptive field size for the convolutions around the relaxation led to it generalizing better to the true ELB.
- Multiplication of the outgoing activations from the encoder and decoder resblocks by a small constant, to ensure stable training at initialization.

We also found that increasing the KL weight to $\beta = 6.6$ promotes better codebook usage and ultimately leads to a

smaller reconstruction error at the end of training.[4]

2.2. Stage Two: Learning the Prior

In the second stage, we fix ϕ and θ, and learn the prior distribution over the text and image tokens by maximizing the ELB with respect to ψ. Here, p_ψ is represented by a 12-billion parameter sparse transformer (Child et al., 2019).

Given a text-image pair, we BPE-encode (Sennrich et al., 2015) the lowercased caption using at most 256 tokens[5] with vocabulary size 16,384, and encode the image using $32 \times 32 = 1024$ tokens with vocabulary size 8192. The image tokens are obtained using argmax sampling from the dVAE encoder logits, without adding any gumbel noise.[6] Finally, the text and image tokens are concatenated and modeled autoregressively as a single stream of data.

The transformer is a decoder-only model in which each image token can attend to all text tokens in any one of its 64 self-attention layers. The full architecture is described in Appendix B.1. There are three different kinds of self-attention masks used in the model. The part of the attention masks corresponding to the text-to-text attention is the standard causal mask, and the part for the image-to-image attention uses either a row, column, or convolutional attention mask.[7]

We limit the length of a text caption to 256 tokens, though it is not totally clear what to do for the "padding" positions in between the last text token and the start-of-image token. One option is to set the logits for these tokens to $-\infty$ in the self-attention operations. Instead, we opt to learn a special padding token separately for each of the 256 text positions. This token is used only when no text token is available. In preliminary experiments on Conceptual Captions (Sharma et al., 2018), we found that this resulted in higher validation loss, but better performance on out-of-distribution captions.

We normalize the cross-entropy losses for the text and image tokens by the total number of each kind in a batch of data. Since we are primarily interested in image modeling, we multiply the cross-entropy loss for the text by $1/8$ and the cross-entropy loss for the image by $7/8$. The objective is optimized using Adam with exponentially weighted iterate averaging; Appendix B.2 describes the training procedure in more detail. We reserved about 606,000 images for validation, and found no signs of overfitting at convergence.

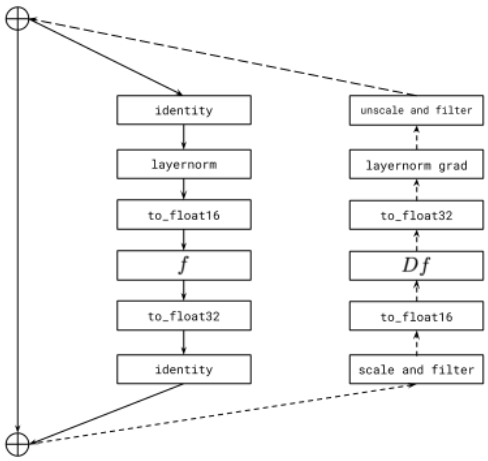


Figure 4. Illustration of per-resblock gradient scaling for a transformer resblock. The solid line indicates the sequence of operations for forward propagation, and the dashed line the sequence of operations for backpropagation. We scale the incoming gradient for each resblock by its gradient scale, and unscale the outgoing gradient before it is added to the sum of the gradients from the successive resblocks. The activations and gradients along the identity path are stored in 32-bit precision. The "filter" operation sets all Inf and NaN values in the activation gradient to zero. Without this, a nonfinite event in the current resblock would cause the gradient scales for all preceding resblocks to unnecessarily drop, thereby resulting in underflow.

2.3. Data Collection

Our preliminary experiments for models up to 1.2 billion parameters were carried out on Conceptual Captions, a dataset of 3.3 million text-image pairs that was developed as an extension to MS-COCO (Lin et al., 2014).

To scale up to 12-billion parameters, we created a dataset of a similar scale to JFT-300M (Sun et al., 2017) by collecting 250 million text-images pairs from the internet. This dataset does not include MS-COCO, but does include Conceptual Captions and a filtered subset of YFCC100M (Thomee et al., 2016). As MS-COCO was created from the latter, our training data includes a fraction of the MS-COCO validation images (but none of the captions). We control for this in the quantitative results presented in Section 3 and find that it has no appreciable bearing on the results. We provide further

[4]This is contrary to the usual tradeoff between the two terms. We speculate that for smaller values of β, the noise from the relaxation causes the optimizer to reduce codebook usage toward the beginning of training, resulting in worse ELB at convergence.

[5]During training, we apply 10% BPE dropout (Provilkov et al., 2019), whose use is common in the neural machine translation literature.

[6]Strictly speaking, Equation 1 requires us to sample from the categorical distribution specified by the dVAE encoder logits, rather than taking the argmax. In preliminary experiments on ImageNet, we found that this was a useful regularizer in the overparameterized regime, and allows the transformer to be trained using soft targets for the cross-entropy loss. We decided against this here since the model in consideration is in the underparameterized regime.

[7]We found using a single attention operation for all three interactions – "text attends to text", "image attends to text", and "image attends to image" – to perform better than using separate attention operations that are independently normalized.

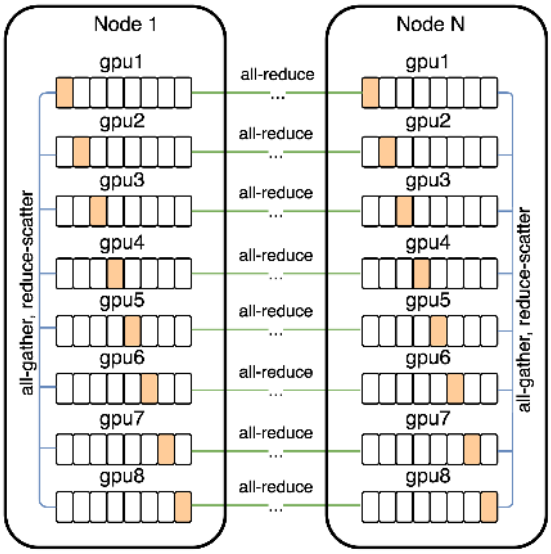


Figure 5. Communication patterns used for distributed training. Each parameter array in the model is sharded among the eight GPUs on each machine. During forward propagation, we prefetch the parameter shards for the next resblock (using all-gather) while computing the activations for the current resblock. To conserve memory, the parameter shards from the other GPUs are immediately discarded. Similarly, during backpropagation, we prefetch the parameter shards for the previous resblock while computing the activations and gradients for the current resblock. After all GPUs have computed the gradient with respect to an all-gathered parameter, the reduce-scatter operation leaves each GPU with only one slice – i.e., the gradient for its parameter shard, averaged over the eight GPUs.

details about the data collection process in Appendix C.

2.4. Mixed-Precision Training

To save GPU memory and increase throughput, most parameters, Adam moments, and activations are stored in 16-bit precision. We also use activation checkpointing and recompute the activations within the resblocks during the backward pass. Getting the model to train in 16-bit precision past one billion parameters, without diverging, was the most challenging part of this project.

We believe the root cause of this instability to be underflow in the 16-bit gradients. Appendix D presents a set of guidelines we developed to avoid underflow when training large-scale generative models. Here, we describe one of these guidelines: per-resblock gradient scaling.

Similar to prior work (Liu et al., 2020), we found that the norms of the activation gradients from the resblocks decrease monotonically as we move from the earlier resblocks to the later ones.[8] As the model is made deeper and wider, the true exponents of the activation gradients for later resblocks can fall below the minimum exponent of the 16-bit format. Consequently, they get rounded to zero, a phenomenon called *underflow*. We found that eliminating underflow allowed for stable training to convergence.

Effective Parameter Count	Compression Rank	Compression Rate
$2.8 \cdot 10^9$ ($d_{\text{model}} = 1920$)	512	$\approx 83\%$
$5.6 \cdot 10^9$ ($d_{\text{model}} = 2688$)	640	$\approx 85\%$
$12.0 \cdot 10^9$ ($d_{\text{model}} = 3968$)	896	$\approx 86\%$

Table 1. We show the relationship between model size and the minimum compression rank for the gradients (up to a multiple of 128) necessary to avoid a gap in the training loss during the first 10% of training. These results suggest that in our setting, we can achieve a compression rate of about 85%, independent of model size.

Standard loss scaling (Micikevicius et al., 2017) is able to avoid underflow when the range spanned by the smallest and largest activation gradients (in absolute value) fits within the exponent range of the 16-bit format. On NVIDIA V100 GPUs, this exponent range is specified by five bits. While this is sufficient for training vanilla language models of the same size, we found the range to be too small for the text-to-image model.

Our fix, which is shown in Figure 4, involves using a separate "gradient scale" for each resblock in the model. This can be seen as a practical alternative to a more general framework for mixed-precision training called Flexpoint (Köster et al., 2017), with the advantage that specialized GPU kernels are not required. We found that Sun et al. (2020) had independently developed similar procedure for training convolutional networks in 4-bit precision.

2.5. Distributed Optimization

Our 12-billion parameter model consumes about 24 GB of memory when stored in 16-bit precision, which exceeds the memory of a 16 GB NVIDIA V100 GPU. We address this using parameter sharding (Rajbhandari et al., 2019). As shown in Figure 5, parameter sharding allows us to almost completely hide the latency of the intra-machine communication by overlapping it with compute-intensive operations.

On the cluster used to train the model, the bandwidth between machines is much lower than the bandwidth among GPUs on the same machine. This makes the cost of the operation used to average the gradient among the machines (all-reduce) the main bottleneck during training. We were

[8] It is possible that better initialization schemes (Liu et al., 2020) might be able to avoid this, but we did not have success with alternative schemes in our experiments.

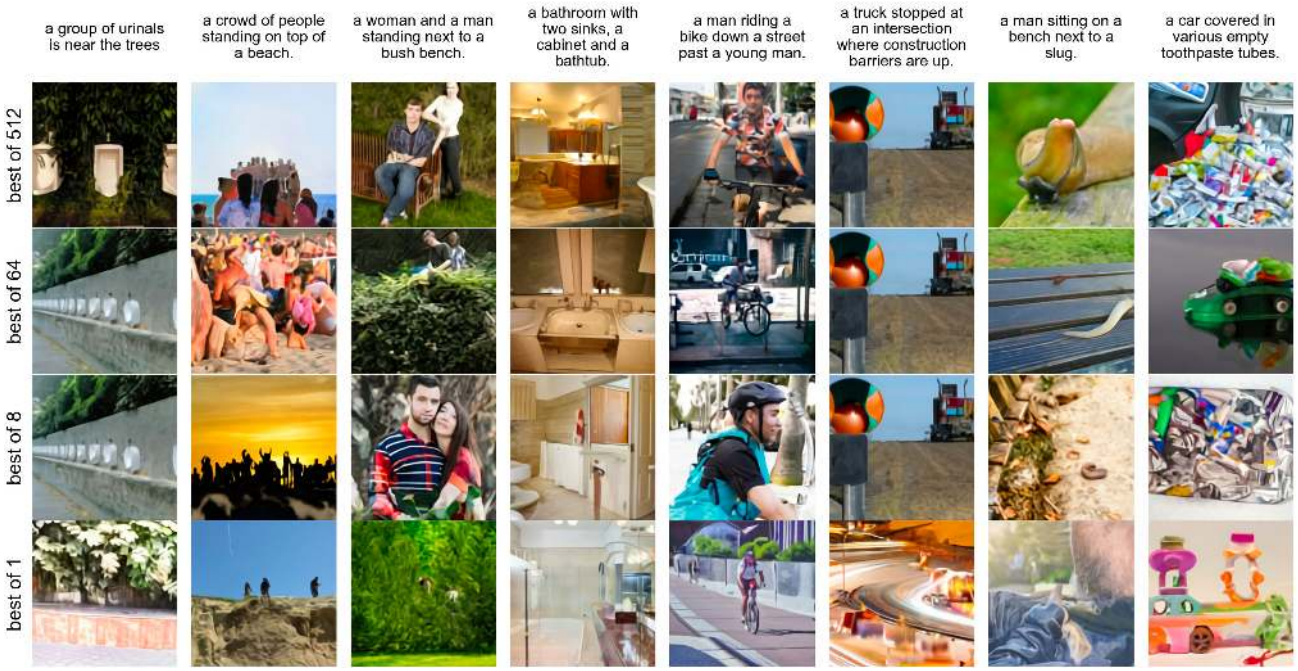


Figure 6. Effect of increasing the number of images for the contrastive reranking procedure on MS-COCO captions.

able to drastically reduce this cost by compressing the gradients using PowerSGD (Vogels et al., 2019).

In our implementation, each GPU in a machine computes the low-rank factors for its parameter shard gradients independently of its neighboring GPUs.[9] Once the low-rank factors are computed, each machine sets its error buffer to the residual between the uncompressed gradient averaged over its eight GPUs (obtained from reduce-scatter), and the decompressed gradient obtained from the low-rank factors.

PowerSGD replaces the large communication operation for an uncompressed parameter gradient with two, much smaller communication operations for its low-rank factors. For a given compression rank r and transformer activation size d_{model}, the compression rate is given by $1 - 5r/(8d_{\text{model}})$ (see Appendix E.1). Table 1 shows that we can achieve a compression rate of about 85%, independent of model size.

In Appendix E.2, we describe various details that were necessary to get PowerSGD to perform well at scale. These include:

- Saving memory by accumulating the gradient into the error buffers during backpropagation, rather than allocating separate buffers.
- Minimizing instances in which we zero out the error buffers (e.g., due to nonfinite values encountered during mixed-precision backpropagation, or when resuming training from a checkpoint).
- Improving numerical stability by using Householder orthogonalization instead of Gram-Schmidt, together with the addition of a small multiple of the identity matrix to the input.
- Avoiding underflow by using a custom 16-bit floating point format for the error buffers, their low-rank factors, and the all-reduce communication operations involving them.

We also found the warm-start procedure for the Q matrix described in Vogels et al. (2019) to be unnecessary: we were able to get equivalent results by fixing Q to a random gaussian matrix at the start of training, and never updating it.[10]

2.6. Sample Generation

Similar to Razavi et al. (2019), we rerank the samples drawn from the transformer using a pretrained contrastive model (Radford et al., 2021). Given a caption and a candidate image, the contrastive model assigns a score based on

[9] There is still intra-machine communication for other operations; what we mean is that the low-rank factors across the shards, when concatenated, are not regarded as collectively approximating the gradient for the full parameter matrix.

[10] We verified that the error in reconstructing the true gradient is higher when Q is fixed as opposed to being updated using warm-starting, so it is interesting that this does not affect the loss. By contrast, resampling Q at every update causes a large performance hit.

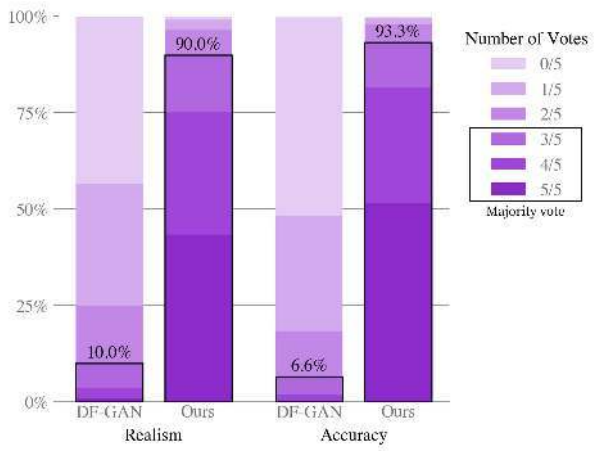


Figure 7. Human evaluation of our model (evaluated zero-shot without temperature reduction) vs prior work (DF-GAN) on captions from MS-COCO. In a best-of-five vote, our model's sample was chosen as the most realistic 90.0% of the time, and was chosen as the image best matching a shared caption 93.3% of the time.

Figure 8. Zero-shot samples from our model on the CUB dataset.

how well the image matches the caption. Figure 6 shows the effect of increasing the number of samples N from which we select the top k images. This process can be seen as a kind of language-guided search (Andreas et al., 2017), and is also similar to the auxiliary text-image matching loss proposed by Xu et al. (2018). Unless otherwise stated, all samples used for both qualitative and quantitative results are obtained without temperature reduction (i.e., using $t = 1$) (except for Figure 2) and use reranking with $N = 512$.

3. Experiments

3.1. Quantitative Results

We evaluate our model zero-shot by comparing it to three prior approaches: AttnGAN (Xu et al., 2018), DM-GAN (Zhu et al., 2019), and DF-GAN (Tao et al., 2020), the last of which reports the best Inception Score (Salimans et al., 2016) and Fréchet Inception Distance (Heusel et al., 2017) on MS-COCO. Figure 3 qualitatively compares samples from our model to those from prior work.

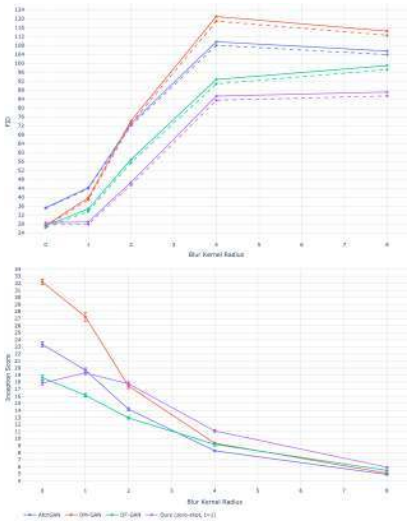
We also conduct a human evaluation similar to the one used in Koh et al. (2021) to compare our approach to DF-GAN, the results of which are shown in Figure 7. Given a caption, the sample from our model receives the majority vote for better matching the caption 93% of the time. It also receives the majority vote for being more realistic 90% of the time.

Figure 9(a) shows that our model also obtains an FID score on MS-COCO within 2 points of the best prior approach, despite having never been trained on the captions. Our training data incorporates a filtered subset of YFCC100M, and we found that it includes about 21% of the images in the MS-COCO validation set from a de-duplication procedure described in the next section. To isolate this effect, we compute the FID statistics for the validation set both with these images (solid lines) and without them (dashed lines), finding no significant change in the results.

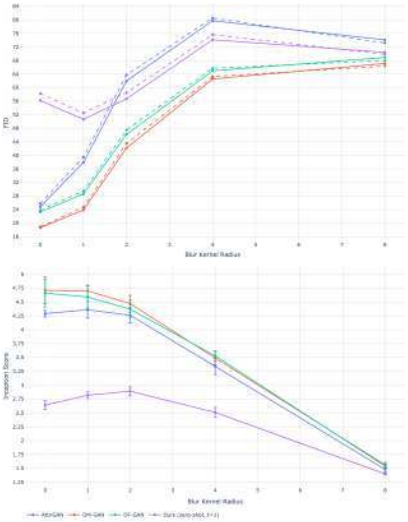
Training the transformer on the tokens from the dVAE encoder allows us to allocate its modeling capacity to the low-frequency information that makes images visually recognizable to us. However, it also disadvantages the model, since the heavy compression renders it unable to produce high-frequency details. To test the effect of this on the quantitative evaluations, we compute the FID and IS in Figure 9(a) after applying a Gaussian filter with varying radius to both the validation images and samples from the models. Our approach achieves the best FID by a margin of about 6 points with a slight blur of radius 1. The gap between our approach and others tends to widen as the blur radius is increased. We also obtain the highest IS when the blur radius is greater than or equal to two.

Our model fares significantly worse on the CUB dataset, for which there is a nearly 40-point gap in FID between our model and the leading prior approach (Figure 9(b)). We found an 12% overlap rate for this dataset, and again observed no significant difference in the results after removing these images. We speculate that our zero-shot approach is less likely to compare favorably on specialized distributions such as CUB. We believe that fine-tuning is a promising direction for improvement, and leave this investigation to future work. Samples from our model for captions in this dataset are shown in Figure 8.

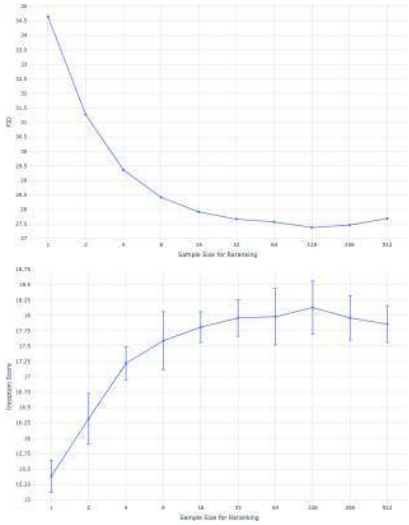
Finally, Figure 9(c) shows clear improvements in FID and IS for MS-COCO as the sample size used for reranking with the contrastive model is increased. This trend continues up to a sample size of 32, after which we observe diminishing



(a) FID and IS on MS-COCO as a function of blur radius.



(b) FID and IS on CUB as a function of blur radius.



(c) FID and IS on MS-COCO as a function of the sample size used for reranking.

Figure 9. Quantitative results on MS-COCO and CUB. Solid lines represent FID computed against the original validation sets, and dashed lines represent FID computed against validation sets with overlapping images removed (see Section 3.2). For MS-COCO, we evaluate all models on a subset of 30,000 captions sampled from the validation set. For CUB, we evaluate all models on all of the unique captions in the test set. We compute the FID and IS using the DM-GAN code, which is available at `https://github.com/MinfengZhu/DM-GAN`.

returns.

3.2. Data Overlap Analysis

We used the deduplication procedure described in Radford et al. (2021) to determine which images to remove. For each validation image, we find the closest image in the training data using a contrastive model specifically trained for this task. We then sort the images in descending order by closeness to their nearest matches in the training data. After inspecting the results by hand, we determine the images to remove by manually selecting a conservative threshold designed to minimize the false negative rate.

3.3. Qualitative Findings

We found that our model has the ability to generalize in ways that we did not originally anticipate. When given the caption "a tapir made of accordion..." (Figure 2a), the model appears to draw a tapir with an accordion for a body, or an accordion whose keyboard or bass are in the shape of a tapir's trunk or legs. This suggests that it has developed a rudimentary ability to compose unusual concepts at high levels of abstraction.

Our model also appears to be capable of combinatorial generalization, such as when rendering text (Figure 2b) or when probed on sentences like "an illustration of a baby hedgehog in a christmas sweater walking a dog" (Figure 2c). Prompts like the latter require the model to perform variable binding (Smolensky, 1990; Greff et al., 2020) – it is the hedgehog that is in the christmas sweater, not the dog. We note, however, that the model performs inconsistently on the task, sometimes drawing both animals with christmas sweaters, or drawing a hedgehog walking a smaller hedgehog.

To a limited degree of reliability, we also find our model to be capable of zero-shot image-to-image translation controllable by natural language (Figure 2d). When the model is given the caption "the exact same cat on the top as a sketch at the bottom" and the top 15×32 part of the image token grid for a photo of a cat, it is able to draw a sketch of a similar looking cat on the bottom.

This works with several other kinds of transformations, including image operations (e.g., changing the color of the image, converting it to grayscale, or flipping it upside-down) and style transfer (e.g., drawing the cat on a greeting card, a postage stamp, or a cell phone case). Some transformations, such as those that involve only changing the color of the animal, suggest that the model is capable of performing a rudimentary kind of object segmentation. We provide additional examples of zero-shot image-to-image translation in Section G.

4. Conclusion

We investigate a simple approach for text-to-image generation based on an autoregressive transformer, when it is executed at scale. We find that scale can lead to improved generalization, both in terms of zero-shot performance relative to previous domain-specific approaches, and in terms of the range of capabilities that emerge from a single generative model. Our findings suggest that improving generalization as a function of scale may be a useful driver for progress on this task.

Acknowledgements

We would like to thank Matthew Knight for reviewing the code release for this work, and Rewon Child, John Schulman, Heewoo Jun, and Prafulla Dhariwal for helpful early feedback on the paper. We would also like to thank Jong Wook Kim for writing the PyTorch package for the contrastive model described in Radford et al. (2019) that we used to rerank the samples from our model.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

Andreas, J., Klein, D., and Levine, S. Learning with latent language. *arXiv preprint arXiv:1711.00482*, 2017.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Bowman, S. R., Vilnis, L., Vinyals, O., Dai, A. M., Jozefowicz, R., and Bengio, S. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

Chen, M., Radford, A., Child, R., Wu, J., Jun, H., Luan, D., and Sutskever, I. Generative pretraining from pixels. In *International Conference on Machine Learning*, pp. 1691–1703. PMLR, 2020.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Cho, J., Lu, J., Schwenk, D., Hajishirzi, H., and Kembhavi, A. X-lxmert: Paint, caption and answer questions with multi-modal transformers. *arXiv preprint arXiv:2009.11278*, 2020.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Dhariwal, P., Jun, H., Payne, C., Kim, J. W., Radford, A., and Sutskever, I. Jukebox: A generative model for music. *arXiv preprint arXiv:2005.00341*, 2020.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.

Greff, K., van Steenkiste, S., and Schmidhuber, J. On the binding problem in artificial neural networks. *arXiv preprint arXiv:2012.05208*, 2020.

Gregor, K., Danihelka, I., Graves, A., Rezende, D., and Wierstra, D. Draw: A recurrent neural network for image generation. In *International Conference on Machine Learning*, pp. 1462–1471. PMLR, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *arXiv preprint arXiv:1706.08500*, 2017.

Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., Mohamed, S., and Lerchner, A. beta-vae: Learning basic visual concepts with a constrained variational framework. 2016.

Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1125–1134, 2017.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Koh, J. Y., Baldridge, J., Lee, H., and Yang, Y. Text-to-image generation grounded by fine-grained user attention. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 237–246, 2021.

Köster, U., Webb, T. J., Wang, X., Nassar, M., Bansal, A. K., Constable, W. H., Elibol, O. H., Gray, S., Hall, S., Hornof, L., et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. *arXiv preprint arXiv:1711.02213*, 2017.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Li, W., Zhang, P., Zhang, L., Huang, Q., He, X., Lyu, S., and Gao, J. Object-driven text-to-image synthesis via adversarial training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12174–12182, 2019.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *European conference on computer vision*, pp. 740–755. Springer, 2014.

Liu, L., Liu, X., Gao, J., Chen, W., and Han, J. Understanding the difficulty of training transformers. *arXiv preprint arXiv:2004.08249*, 2020.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Mansimov, E., Parisotto, E., Ba, J. L., and Salakhutdinov, R. Generating images from captions with attention. *arXiv preprint arXiv:1511.02793*, 2015.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Nguyen, A., Clune, J., Bengio, Y., Dosovitskiy, A., and Yosinski, J. Plug & play generative networks: Conditional iterative generation of images in latent space. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4467–4477, 2017.

Oord, A. v. d., Vinyals, O., and Kavukcuoglu, K. Neural discrete representation learning. *arXiv preprint arXiv:1711.00937*, 2017.

Provilkov, I., Emelianenko, D., and Voita, E. Bpe-dropout: Simple and effective subword regularization. *arXiv preprint arXiv:1910.13267*, 2019.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision. 2021.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

Razavi, A., Oord, A. v. d., and Vinyals, O. Generating diverse high-fidelity images with vq-vae-2. *arXiv preprint arXiv:1906.00446*, 2019.

Reed, S., Akata, Z., Mohan, S., Tenka, S., Schiele, B., and Lee, H. Learning what and where to draw. *arXiv preprint arXiv:1610.02454*, 2016a.

Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., and Lee, H. Generative adversarial text to image synthesis. In *International Conference on Machine Learning*, pp. 1060–1069. PMLR, 2016b.

Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pp. 1278–1286. PMLR, 2014.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. Improved techniques for training gans. *arXiv preprint arXiv:1606.03498*, 2016.

Salimans, T., Karpathy, A., Chen, X., and Kingma, D. P. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Sharma, P., Ding, N., Goodman, S., and Soricut, R. Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2556–2565, 2018.

Smolensky, P. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, 46(1-2):159–216, 1990.

Sun, C., Shrivastava, A., Singh, S., and Gupta, A. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, pp. 843–852, 2017.

Sun, X., Wang, N., Chen, C.-Y., Ni, J., Agrawal, A., Cui, X., Venkataramani, S., El Maghraoui, K., Srinivasan, V. V.,

and Gopalakrishnan, K. Ultra-low precision 4-bit training of deep neural networks. *Advances in Neural Information Processing Systems*, 33, 2020.

Tao, M., Tang, H., Wu, S., Sebe, N., Wu, F., and Jing, X.-Y. Df-gan: Deep fusion generative adversarial networks for text-to-image synthesis. *arXiv preprint arXiv:2008.05865*, 2020.

Thomee, B., Shamma, D. A., Friedland, G., Elizalde, B., Ni, K., Poland, D., Borth, D., and Li, L.-J. Yfcc100m: The new data in multimedia research. *Communications of the ACM*, 59(2):64–73, 2016.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Vogels, T., Karimireddy, S. P., and Jaggi, M. Powersgd: Practical low-rank gradient compression for distributed optimization. *arXiv preprint arXiv:1905.13727*, 2019.

Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., and Perona, P. Caltech-ucsd birds 200. 2010.

Xu, T., Zhang, P., Huang, Q., Zhang, H., Gan, Z., Huang, X., and He, X. Attngan: Fine-grained text to image generation with attentional generative adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1316–1324, 2018.

Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X., and Metaxas, D. N. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 5907–5915, 2017.

Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X., and Metaxas, D. N. Stackgan++: Realistic image synthesis with stacked generative adversarial networks. *IEEE transactions on pattern analysis and machine intelligence*, 41(8):1947–1962, 2018.

Zhu, M., Pan, P., Chen, W., and Yang, Y. Dm-gan: Dynamic memory generative adversarial networks for text-to-image synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5802–5810, 2019.

```
def preprocess_image(img, target_res):
    h, w  = tf.shape(img)[0], tf.shape(img)[1]
    s_min = tf.minimum(h, w)
    img   = tf.image.random_crop(img, 2 * [s_min] + [3])

    t_min = tf.minimum(s_min, round(9 / 8 * target_res))
    t_max = tf.minimum(s_min, round(12 / 8 * target_res))
    t     = tf.random.uniform([], t_min, t_max + 1, dtype=tf.int32)
    img   = tf.image.resize_images(img, [t, t], method=tf.image.ResizeMethod.AREA,
                align_corners=True)
    img   = tf.cast(tf.rint(tf.clip_by_value(img, 0, 255)), tf.uint8)
    img   = tf.image.random_crop(img, 2 * [target_res] + [channel_count])
    return tf.image.random_flip_left_right(img)
```

Listing 1. TensorFlow (Abadi et al., 2016) image preprocessing code for training dVAE. We use `target_res = 256` and `channel_count = 3`.

A. Details for Discrete VAE

A.1. Architecture

The dVAE encoder and decoder are convolutional (LeCun et al., 1998) ResNets (He et al., 2016) with bottleneck-style resblocks. The models primarily use 3×3 convolutions, with 1×1 convolutions along skip connections in which the number of feature maps changes between the input and output of a resblock. The first convolution of the encoder is 7×7, and the last convolution of the encoder (which produces the $32 \times 32 \times 8192$ output used as the logits for the categorical distributions for the image tokens) is 1×1. Both the first and last convolutions of the decoder are 1×1. The encoder uses max-pooling (which we found to yield better ELB than average-pooling) to downsample the feature maps, and the decoder uses nearest-neighbor upsampling. The precise details for the architectures are given in the files `dvae/encoder.py` and `dvae/decoder.py` of the code release.

A.2. Training

The dVAE is trained on the same dataset as the transformer, using the data augmentation code given in Listing 1. Several quantities are decayed during training, all of which use a cosine schedule:

1. The KL weight β is increased from 0 to 6.6 over the first 5000 updates. Bowman et al. (2015) use a similar schedule based on the sigmoid function.
2. The relaxation temperature τ is annealed from 1 to $1/16$ over the first 150,000 updates. Using a linear annealing schedule for this typically led to divergence.
3. The step size is annealed from $1 \cdot 10^{-4}$ to $1.25 \cdot 10^{-6}$ over 1,200,000 updates.

The decay schedules for the relaxation temperature and the step size are especially important for stability and successful optimization.

We update the parameters using AdamW (Loshchilov & Hutter, 2017) with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and weight decay multiplier 10^{-4}. We use exponentially weighted iterate averaging for the parameters with decay coefficient 0.999. The reconstruction term in the ELB is a joint distribution over the $256 \times 256 \times 3$ values for the image pixels, and the KL term is a joint distribution over the 32×32 positions in the spatial grid output by the encoder. We divide the overall loss by $256 \times 256 \times 3$, so that the weight of the KL term becomes $\beta/192$, where β is the KL weight. The model is trained in mixed-precision using standard (i.e., global) loss scaling on 64 16 GB NVIDIA V100 GPUs, with a per-GPU batch size of 8, resulting in a total batch size of 512. It is trained for a total of 3,000,000 updates.

A.3. The Logit-Laplace Distribution

The ℓ_1 and ℓ_2 reconstruction objectives are commonly used when training VAEs. These objectives correspond to using Laplace and Gaussian distributions for $\ln p_\theta(x \mid y, z)$ in Equation 1, respectively. There is a strange mismatch in this

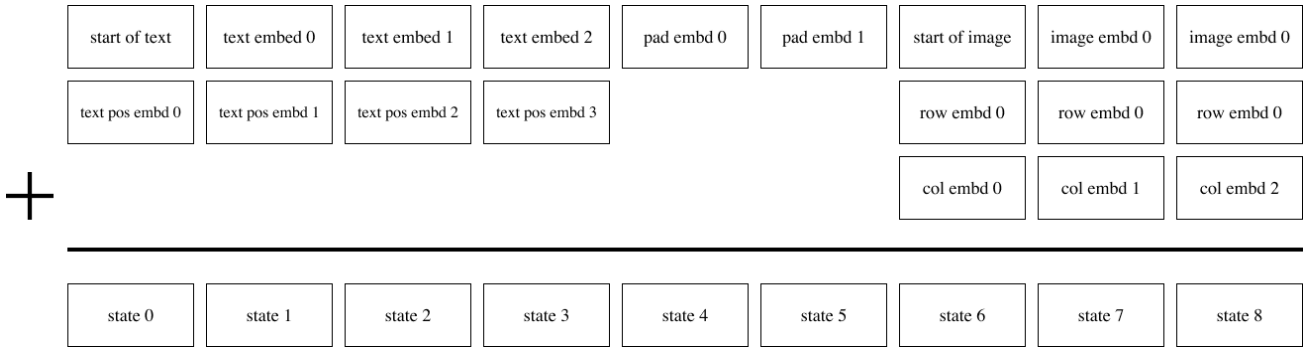


Figure 10. Illustration of the embedding scheme for a hypothetical version of our transformer with a maximum text length of 6 tokens. Each box denotes a vector of size $d_{\text{model}} = 3968$. In this illustration, the caption has a length of 4 tokens, so 2 padding tokens are used (as described in Section 2.2). Each image vocabulary embedding is summed with a row and column embedding.

modeling choice: pixel values lie within a bounded interval, but both of these distributions are supported by the entire real line. Hence, some amount of likelihood will be placed outside the admissible range of pixel values.

We present a variant of the Laplace distribution that is also supported by a bounded interval. This resolves the discrepancy between the range of the pixel values being modeled and the support of the distribution used to model them. We consider the pdf of the random variable obtained by applying the sigmoid function to a Laplace-distributed random variable. This pdf is defined on $(0, 1)$ and is given by

$$f(x \,|\, \mu, b) = \frac{1}{2bx(1-x)} \exp\left(-\frac{|\operatorname{logit}(x) - \mu|}{b}\right); \tag{2}$$

we call it the *logit-Laplace distribution.* We use the logarithm of the RHS of Equation 2 as the reconstruction term for the training objective of the dVAE.

The decoder of the dVAE produces six feature maps representing the sufficient statistics of the logit-Laplace distribution for the RGB channels of the image being reconstructed. The first three feature maps represent the μ parameter for the RGB channels, and the last three represent $\ln b$. Before feeding an image into the dVAE encoder, we transform its values using $\varphi : [0, 255] \to (\epsilon, 1 - \epsilon)$, which is given by

$$\varphi : x \mapsto \frac{1 - 2\epsilon}{255} x + \epsilon. \tag{3}$$

This restricts the range of the pixel values to be modeled by the dVAE decoder to $(\epsilon, 1-\epsilon)$, which avoids numerical problems arising from the $x(1 - x)$ in Equation 2. We use $\epsilon = 0.1$. To reconstruct an image for manual inspection or computing metrics, we ignore $\ln b$ and compute $\hat{x} = \varphi^{-1}(\text{sigmoid}(\mu))$, where μ is given by the first three feature maps output by the dVAE decoder.[11]

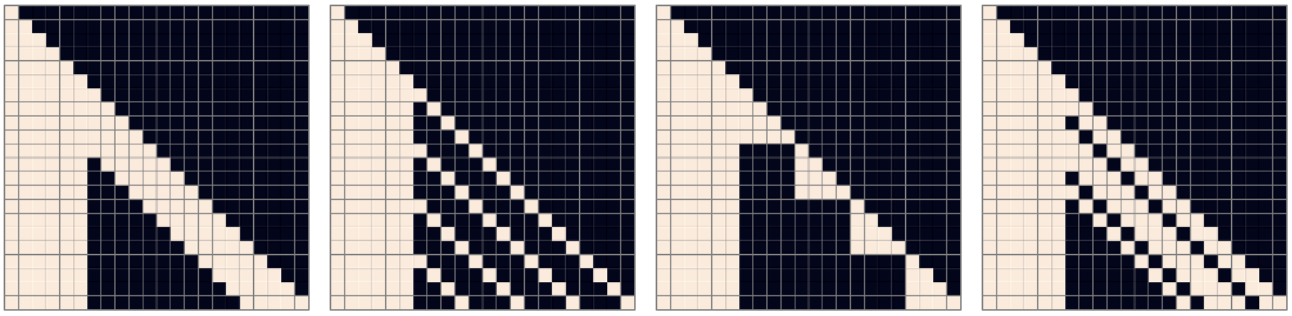
B. Details for Transformer

B.1. Architecture

Our model is a decoder-only sparse transformer of the same kind described in Child et al. (2019), with broadcasted row and column embeddings for the part of the context for the image tokens. A complete description of the embedding scheme used in our model is shown in Figure 10. We use 64 attention layers, each of which uses 62 attention heads with a per-head state size of 64.

The model uses three kinds of sparse attention masks, which we show in Figure 11. The convolutional attention mask (Figure 11(d)) is only used in the last self-attention layer. Otherwise, given the index i of a self-attention layer (with $i \in [1, 63]$), we use the column attention mask (Figure 11(c)) if $i - 2 \mod 4 = 0$, and row attention otherwise. E.g., the first four self-attention layers use "row, column, row, row", respectively. With the exception of the convolutional attention mask,

[11] See `notebooks/usage.ipynb` of the code release for an example.



(a) Row attention mask. (b) Column attention mask. (c) Column attention mask with transposed image states. (d) Convolutional attention mask.

Figure 11. Illustration of the three types of attention masks for a hypothetical version of our transformer with a maximum text length of 6 tokens and image length of 16 tokens (i.e., corresponding to a 4×4 grid). Mask (a) corresponds to row attention in which each image token attends to the previous 5 image tokens in raster order. The extent is chosen to be 5, so that the last token being attended to is the one in the same column of the previous row. To obtain better GPU utilization, we transpose the row and column dimensions of the image states when applying column attention, so that we can use mask (c) instead of mask (b). Mask (d) corresponds to a causal convolutional attention pattern with wraparound behavior (similar to the row attention) and a 3×3 kernel. Our model uses a mask corresponding to an 11×11 kernel.

which we found to provide a small boost in performance over the row and dense causal attention masks when used in the final self-attention layer, this is the same configuration used in Child et al. (2019).

B.2. Training

When training the transformer, we apply data augmentation to the images before encoding them using the dVAE encoder. We use slightly different augmentations from the ones used to train the dVAE; the code used for this is given in Listing 2. We also apply 10% BPE dropout when BPE-encoding the captions for training. The model is trained using per-resblock scaling (see Section 2.4) and gradient compression (see Section 2.5) with total compression rank 896 (so that each GPU uses a compression rank of 112 for its parameter shards). As shown in Table 1, this results in a compression rate of about 86%, which we analyze in Section E.1.

We update the parameters using AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.96$, $\epsilon = 10^{-8}$, and weight decay multiplier $4.5 \cdot 10^{-2}$. We clip the decompressed gradients by norm using a threshold of 4, prior to applying the Adam update. Gradient clipping is only triggered during the warm-up phase at the start of training. To conserve memory, most Adam moments (see Section D for details) are stored in 16-bit formats, with a 1-6-9 format for the running mean (i.e., 1 bit for the sign, 6 bits for the exponent, and 9 bits for the significand), and a 0-6-10 format for the running variance. We clip the estimate for running variance by value to 5 before it is used to update the parameters or moments. Finally, we apply exponentially weighted iterate averaging by asynchronously copying the model parameters from the GPU to the CPU once every 25 updates, using a decay coefficient of 0.99.

We trained the model using 1024, 16 GB NVIDIA V100 GPUs and a total batch size of 1024, for a total of 430,000 updates. At the start of training, we use a linear schedule to ramp up the step size to $4.5 \cdot 10^{-4}$ over 5000 updates, and halved the step size each time the training loss appeared to plateau. We did this a total of five times, ending training with a final step size that was 32 times smaller than the initial one. We reserved about 606,000 images for validation, and did not observe overfitting at any point during training.

C. Details for Data Collection

In order to train the 12-billion parameter transformer, we created a dataset of a similar scale to JFT-300M by collecting 250 million text-image pairs from the internet. As described in Section 2.3, this dataset incorporates Conceptual Captions, the text-image pairs from Wikipedia, and a filtered subset of YFCC100M. We use a subset of the text, image, and joint text and image filters described in Sharma et al. (2018) to construct this dataset. These filters include discarding instances whose captions are too short, are classified as non-English by the Python package `cld3`, or that consist primarily of boilerplate

```
def preprocess_image(img, target_res):
    h, w  = tf.shape(img)[0], tf.shape(img)[1]
    s_min = tf.minimum(h, w)

    off_h = tf.random.uniform([], 3 * (h - s_min) // 8,
        tf.maximum(3 * (h - s_min) // 8 + 1, 5 * (h - s_min) // 8),
        dtype=tf.int32)
    off_w = tf.random.uniform([], 3 * (w - s_min) // 8,
        tf.maximum(3 * (w - s_min) // 8 + 1, 5 * (w - s_min) // 8),
        dtype=tf.int32)

    # Random full square crop.
    img   = tf.image.crop_to_bounding_box(img, off_h, off_w, s_min, s_min)
    t_max = tf.minimum(s_min, round(9 / 8 * target_res))
    t     = tf.random.uniform([], target_res, t_max + 1, dtype=tf.int32)
    img   = tf.image.resize_images(img, [t, t], method=tf.image.ResizeMethod.AREA,
                align_corners=True)
    img   = tf.cast(tf.rint(tf.clip_by_value(img, 0, 255)), tf.uint8)

    # We don't use hflip aug since the image may contain text.
    return tf.image.random_crop(img, 2 * [target_res] + [channel_count])
```

Listing 2. TensorFlow (Abadi et al., 2016) image preprocessing code for training the transformer. We use `target_res = 256` and `channel_count = 3`.

phrases such as "photographed on <date>", where <date> matches various formats for dates that we found in the data. We also discard instances whose images have aspect ratios not in $[1/2, 2]$. If we were to use to very tall or wide images, then the square crops used during training would likely exclude objects mentioned in the caption.

D. Guidelines for Mixed-Precision Training

The most challenging part of this project was getting the model to train in 16-bit precision past one billion parameters. We were able to do this after detecting for underflow in various parts of training, and revising the code to eliminate it. We developed a set of guidelines as a result of this process that we present here.[12]

1. **Use per-resblock gradient scaling (Figure 4) instead of standard loss scaling.** Our model uses 128 gradient scales, one for each of its resblocks. All of the gradient scales are initialized to $M \cdot 2^{13}$, where M is the number of data-parallel replicas (i.e., the number of GPUs). In our setup, each grad scale is multiplied by $2^{1/1000}$ at every parameter update when there are no nonfinite values for any parameter gradient in that resblock. Otherwise, we divide the grad scale by $\sqrt{2}$ and skip the update. We also disallow consecutive divisions of the same grad scale within a window of 125 updates. All grad scales are clamped to the range $[M \cdot 2^7, M \cdot 2^{24}]$ after being updated. Figure 12 shows the gradient scales in the early phase of training for a 2.8-billion parameter model.

2. **Only use 16-bit precision where it is really necessary for performance.** In particular, store all gains, biases, embeddings, and unembeddings in 32-bit precision, with 32-bit gradients (including for remote communication) and 32-bit Adam moments. We disable gradient compression for these parameters (though PowerSGD would not make sense for 1D parameters like gains and biases). The logits for the text and image tokens are computed and stored in 32-bit precision. We found that storing the embeddings in 16-bit precision sometimes caused divergence early in optimization, and using 16-bit logits resulted in a small shift in the training curve, so we switched to use 32-bit precision out of an abundance of caution.

3. **Avoid underflow when dividing the gradient.** For data-parallel training, we need to divide the gradients by the total number of data-parallel workers M. One way to do this is to divide the loss by the per-machine batch size, and then divide the parameter gradients by M before summing them over the machines (using all-reduce). To save time and space, the gradients are usually computed and stored in 16-bit precision. When M is large, this division could result in

[12] Fewer of these guidelines may be necessary on hardware like the TPU that has native support for the bfloat16 format, since the larger 8-bit exponent range makes underflow less likely to occur.

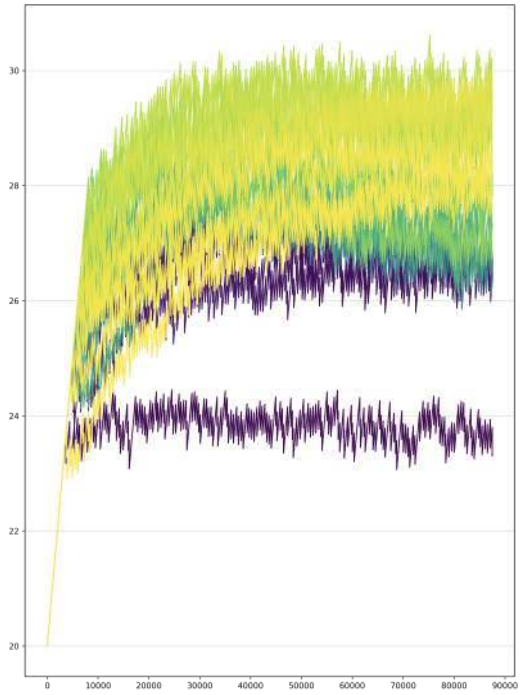


Figure 12. Plot of per-resblock gradient scales for a 2.8-billion parameter text-to-image transformer trained without gradient compression. The x-axis is parameter updates, and the y-axis is the base-2 logarithm of the gradient scale. Darkest violet corresponds to the first resblock, and brightest yellow corresponds to the last (of which there are 128 total). The gradient scale for the second MLP resblock hovers at around 2^{24}, while the others stay within a 4-bit range. The extent of this range increases as the model is made larger.

underflow before the gradients are summed. On the other hand, if we attempt to sum the gradients first and then divide them later, we could encounter overflow in the all-reduce.

Our solution for this problem attempts to minimize the loss of information in the division prior to the all-reduce, without danger of overflow. To do this, we divide the loss by the overall batch size (which includes M as a factor) rather than the per-machine batch size, and multiply the gradient scales by M to compensate, as described in (1). Then, prior to the all-reduce operation, we divide the gradients by a constant that was tuned by hand to avoid both underflow and overflow. This was done by inspecting histograms of the exponents (i.e., base-2 logarithms) of the absolute values of the scalar components of the per-parameter gradients. Since the gradient scaling keeps the gradients close to right end of the exponent range of the 16-bit format, we found that the same constant worked well for all parameters in the model with 16-bit gradients. When using PowerSGD, we chose different constants for the P and Q matrices.

E. Details for Distributed Optimization

We use PowerSGD (Vogels et al., 2019) to compress the gradients with respect to all parameters except the embeddings, unembeddings, gains, and biases. In Section E.1, we derive an expression for the reduction in the amount of data communicated as a function of the compression rank and model size. In Section E.2, we present a detailed overview of our adaptation of PowerSGD, and the modifications we had to make in order to fix performance regressions, some of which only manifest at billion-parameter scale.

E.1. Bandwidth Analysis

Gradient compression uses the factorization $G \approx PQ^t$, where P and Q both have rank r. Instead of using a single all-reduce to transmit G, we use two, smaller all-reduces to transmit both P and Q^t in succession. Hence, the compression ratio is the sum of the sizes of the P and Q matrices divided by the sum of the sizes of the G matrices. We shard along axis 1 for all parameters except for the second MLP matrix. The derivation of the compression ratio in our setup is given in Table 2. We note that the choice of shard axis changes the compression ratio for the MLP matrices. Finally, this analysis excludes the

Parameter Names	Parameter Shard Gradient Shape (No Compression)	P shape	Q shape
qkv and post-attention matrices	$d \times (d/m)$	$d \times (r/m)$	$(r/m) \times (d/m)$
First MLP matrix	$d \times (4d/m)$	$d \times (r/m)$	$(r/m) \times (4d/m)$
Second MLP matrix	$(4d/m) \times d$	$(4d/m) \times (r/m)$	$(r/m) \times d$
Total size	$12d^2/m$	$(5drm + 4dr)/m^2$	$(drm + 8dr)/m^2$

Table 2. We analyze the amount of data sent from each GPU on a given machine to GPUs on other machines, in the case where we shard the parameters among the m GPUs on each machine. Here, r denotes the rank used for compression, and d the transformer hidden size. The compression ratio is given by the sum of the last two columns of the last row, divided by the first column of the last row. This comes out to $r(m+2)/(2dm)$, which for $m = 8$ is $5r/8d$.

embeddings, unembeddings, gains, and biases, for which we do not use compression. The total fraction of the bandwidth used by these parameters becomes smaller as the model size is increased.

E.2. Implementation Details

We describe the steps in our implementation of PowerSGD in detail, since these details were crucial in getting it to work efficiently and reliably at billion-parameter scale.

1. Our training setup uses a combination of parameter sharding and gradient compression, as described in Section 2.5. During backpropagation, while recomputing the activations and computing the gradients for the current resblock, we prefetch the parameters for the preceding resblock using all-gather. Once each GPU has computed the gradient with respect to a full parameter matrix, we compute the average of the slice of the gradient corresponding to the GPU's parameter shard, and discard the full gradient immediately to conserve memory. This average is taken over all of the GPUs on a machine using reduce-scatter.

2. If there are no nonfinite values in the result of the reduce-scatter (which could be caused by overflow in backpropagation or the reduce-scatter), we divide the result by the resblock's gradient scale, and add it to the error buffer (i.e., the buffer used for error correction). Otherwise, we do nothing and proceed with backpropagation; a single nonfinite value in the gradient means that the entire update will be skipped, which happens about 5% of the time. The error buffer uses the same 1-6-9 format used for the Adam mean, which we describe in Section B.2; the larger exponent range ensures that this division does not result in underflow. Adding the gradients directly to the error buffers avoids redundantly allocating another set of buffers of size equal to the parameter shard gradients.

3. Once the reduce-scatter operations for the resblock have finished, we schedule the operations to compute the P matrices from the errors buffers and the Q matrices, whose values are fixed at the start of training (see Section 2.5). Both the P and Q matrices are stored in 1-6-9 format and have their values scaled by predetermined constants, as discussed in Section D.

4. Once each GPU has computed the P matrices for the parameter shards in a resblock, they are averaged with the P matrices from the GPUs with the same ordinal on all other machines, using a single, grouped all-reduce operation. This all-reduce is carried out in the 1-6-9 format, using a custom kernel. The grouping results in better bandwidth utilization, since it avoids scheduling many all-reduce calls for smaller, individual parameters, each of which carries some overhead. We clamp any infinities in the results of the all-reduce to the maximum value of the 1-6-9 format (which is slightly less than 16), retaining the sign. With our choice of scaling factors for the P and Q matrices, this clamping happens very rarely.

5. Once the all-reduce operation for the P matrices for a resblock have finished, we orthogonalize the columns of the resulting matrices. We use a custom Householder orthogonalization kernel rather than Gram-Schmidt, as we found the latter to be numerically unstable. We also add $\epsilon I_{m \times r}$ to P in order to ensure that the result is not near rank-deficient, where $\epsilon = 10^{-6}$. Here, $I_{m \times r}$ is a rectangular matrix of the same size as the P matrix to which it is added; it contains the $r \times r$ identity matrix and has zeros elsewhere. The orthogonalizalied P matrices are stored in 1-6-9 format, but without scaling.

6. Once the P matrices for a resblock have been orthogonalized, we schedule the operations to compute the new Q matrices from the error buffers and the P matrices.

7. Once the new Q matrices for a resblock have been computed, we schedule another grouped all-reduce, similar to what we did for the P matrices. As in step (4), we clamp all infinities in the results of the all-reduce to the maximum value of the 1-6-9 format, retaining the sign. The error buffers for the resblock have now been decomposed into low-rank factors P and Q^t.

8. The gradients for all parameters that are not compressed are grouped together into a single, 32-bit precision all-reduce. Section D explains why we use 32-bit precision for these parameters and their gradients.

9. Once all GPUs on a machine have finished steps (7) and (8) for every resblock in the model, the values of the P and Q matrices for the same parameter shard on all machines will be identical. We then compute the global gradient norm, which is the sum of two quantities: (a) the sum of the squared Frobenius norms of the Q matrices over all of the parameter shards on a machine, and (b) the sum of the squared norms of the gradients for the parameter shards that do not use compression, taken over all such parameter shards on a machine. We need to compute this value for gradient clipping (see Section B.2).

10. While computing the global norm, we also synchronize the information from step (2) about which parameter shard gradients contained nonfinite values after the reduce-scatter. After doing this, we have two pieces of information for each parameter shard: (a) whether its error buffer from step (2) contains nonfinite values on the current GPU, and (b) whether P or Q contains nonfinite values. We cannot rely on the values of the P and Q matrices to determine (b), since we clamp infinities as described in step (4). If we find that the gradient with respect to any parameter shard on the machine contains nonfinite values, then we set the global norm to infinity.

11. Once all of the all-reduces have finished and the global norm has been computed, we can apply the parameter updates. Like backpropagation, the parameter updates proceed resblock-by-resblock. The first step is to compute the decompressed gradients by forming the product PQ^t for all parameters in a given resblock. To avoid overflow, these products are computed in 32-bit precision. We can then apply the Adam update to the parameters using the decompressed gradients and the global norm computed in step (9). If the global norm is not finite, then the update to the parameters and Adam moments is skipped. We note that the decompressed gradient must be divided by the scale of the Q matrix (the P matrix is stored without scaling after orthogonalization).

12. The second step is the update to the error buffers. First, we use the results from step (10) to check if the P and Q matrices for a given parameter shard contain only finite values. If this is the case, then we divide the decompressed gradient by the total number of machines, and subtract it from the current value for the error buffer. This sets the error buffer to the difference between the "local" gradient averaged over the GPUs on the machine using reduce-scatter, and the "remote" decompressed gradient (i.e., the "error"). If either P or Q contains nonfinite values, then we check if the error buffer computed in step (2) contains only finite values. If it does, then we preserve its value and do nothing. If it does not, then we set it to zero. The purpose of this tedious logic is to set an error buffer to zero only when we must do so, because it has been contaminated with nonfinite values. We found that error buffers getting set to zero too frequently by gradient scaling events leads to performance regressions.

13. The parameter shards whose gradients are not compressed are updated separately.

We also note the following important optimizations:

1. There are several opportunities for overlap between compute and communication in the above steps. For example, while we are running step (2) for resblock i, we can proceed to steps (3)–(8) for all resblocks $j > i$. Exploiting opportunities for overlap is necessary to achieve good performance.

2. We throttle specific operations that are liable to exhaust all available memory. For example, we only prefetch the parameters from the preceding resblock when the reduce-scatter operations have finished for the current one. Otherwise, we risk running out of memory by holding on to the full parameters. We also throttle the Adam updates, so that we do not decompress all of the gradients at once.

3. There are two places in the implementation where the transposition matters: (a) the choice of shard axis for the MLP matrices and (b) whether we compute the low-rank factorization for a gradient or its transpose. The former influences the bandwidth analysis, which we present in Section E.1. The latter influences the cost of the orthogonalization. Suppose

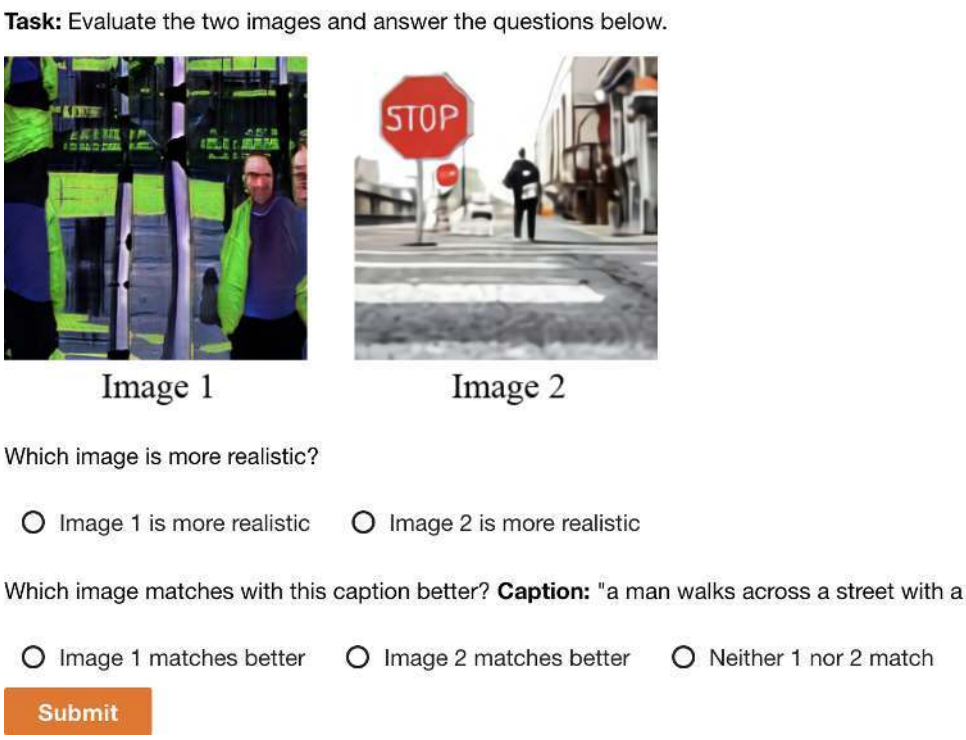


Figure 13. Example task interface shown to workers.

that the gradient G is $m \times n$ and its low-rank factors P and Q^t are $m \times r$ and $r \times n$, respectively, with $r \ll m, n$. To make orthogonalization cheaper, we transpose G appropriately so that $m \leqslant n$.

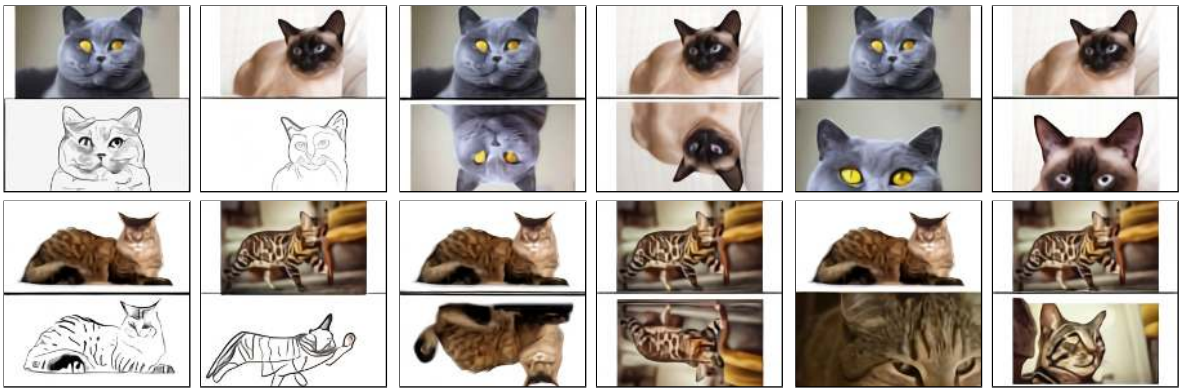
At first glance, it may seem like a limitation that the NCCL all-gather and reduce-scatter primitives shard along axis 0 only. We may need to transpose some matrices before and after communication operations because of (a) and (b), which would require additional time and potentially special care to avoid out-of-memory errors. In fact, we never actually needed to do this. This is because we stored some of the parameters in their transposed formats and exploited the `transpose_a` and `transpose_b` parameters of the matrix multiplication kernels used in forward propagation, backpropagation, and steps (1)–(13) above. This allowed us to avoid explicit transposition while retaining the freedom to choose how to handle (a) and (b).

4. In step (12) above, we note that setting the error buffers to zero too often can cause performance regressions. We wanted to avoid doing this when resuming training from a checkpoint, which happens more frequently for larger jobs as it is likely that a machine will periodically fail. Naively, this would require uploading the error buffers from all of the machines along with the model checkpoints. Since we use a total of 128 machines for training, this would lead to 128 times greater storage usage, which is extremely wasteful.

   Fortunately, this is unnecessary, as error correction depends only on the sum of the error buffers. This property follows from linearity and the sequence of operations used by PowerSGD. Hence, it suffices to store the sums of the errors buffers taken across all GPUs with the same ordinal. When resuming from a checkpoint, we can divide the error buffers by the total number of machines and broadcast them.

F. Details for Human Evaluation Experiments

We start with a list of 1000 captions and generate one sample image per model per caption. Captions and sample images are then used to create 1000 image comparison tasks per experiment, which we submitted to Amazon's Mechanical Turk. Each task was answered by five distinct workers. Workers were asked to compare two images and answer two questions about them: (1) which image is most realistic, and (2) which image best matches the shared caption. The experimental setup provided to workers is shown in Figure 13. One worker's answers were disqualified due to a high rate of disagreement



(a) “the exact same cat on the top as a sketch on the bottom”

(b) “the exact same photo on the top reflected upside-down on the bottom”

(c) “2 panel image of the exact same cat. on the top, a photo of the cat. on the bottom, an extreme close-up view of the cat in the photo.”

(d) “the exact same cat on the top colored red on the bottom”

(e) “2 panel image of the exact same cat. on the top, a photo of the cat. on the bottom, the cat with sunglasses.”

(f) “the exact same cat on the top as a postage stamp on the bottom”

Figure 14. Further examples of zero-shot image-to-image translation.

with other workers combined with a fast answer velocity (with many submission times under 4 seconds); all other worker answers were kept.

G. Zero-Shot Image-to-Image Translation

Figure 14 shows further examples of zero-shot image-to-image translation, which we discussed in Section 3.3. We did not anticipate that this capability would emerge, and made no modifications to the training procedure to encourage it.