

## A Experimental Details

### A.1 Model architecture

Across all of our experiments, we used four decoder-only transformer models [61] that share the same architecture and only differ in layer dimensions and tokenizer. We specify the exact layer dimensions and tokenizers used in Table 1.

We used a model architecture based on GPT-2 [51] with several small changes based on modern practices to improve model FLOPs utilization (MFU) [10], simplify implementation, and marginally improve model performance:

- Head dimension 128
  - We increased the head dimension on the smaller models from 64 to 128 to achieve a higher MFU. We trained each model on TPU v4, which has 128 x 128 multiply-accumulators. Using a head dimension any lower than 128 would prevent us from filling the multiply-accumulators, thereby lowering MFU. This decision follows the Gemma family of models optimized for TPU training [59].
- RMSNorm [74]
  - We wanted to include the Muon optimizer in our benchmarks, but Muon does not support training 1D layers. To get around this limitation, we simply omitted the use of any 1D layers by using RMSNorm with no bias.
- Untied embeddings
  - We do not tie the weights of the first and last layer. This increases number of trainable parameters but does not increase the number of active parameters or the cost of training.
- Rotary positional embeddings (RoPE) [58]
- QK-norm [11]
- Gelu [25]

We trained most models to follow Chinchilla scaling laws, i.e. using 20 tokens per active parameter. The only exception to this rule is the GPT-3 (1.3B) model, which we could only afford to train for 10B tokens, similar to the original GPT-2 (1.5B) model.

Model	49M	19M[69]	GPT-2 (124M)	GPT-3 (1.3B)
Dataset	Fineweb-Edu	C4	Fineweb	Fineweb
Tokenizer	GPT-2	T5	GPT-2	GPT-2
Vocabulary size	50257	32101	50257	50257
Model / embedding dimension	384	512	768	2048
Hidden dimension	1536	2048	3072	8192
Head dimension	128	128	128	128
Number of layers	6	6	12	24
Sequence length	512	512	1024	1024
Non-embedding parameters	11M	19M	85M	1.2B
Embedding parameters	2 x 19M	2 x 16M	2 x 39M	2 x 103M
Active parameters	30M	35M	124M	1.3B
Total trainable parameters	49M	52M	162M	1.4B
Training tokens	599M	705M	2.5B	10B

Table 1: Model dimensions and configurations across different architectures.

### A.2 Computational cost

We used a slice of 32 TPU v4 chips to train all models [30]. Table 2 shows the time for a single training run of each model using batch size  $\sim 4$  and higher, as well as the total computational budget for each model. Using batch sizes 1 and 2 resulted in roughly 30-70% drop in MFU, depending on the model size. In practice we do not recommend using a batch size so small that it severely degrades MFU. Instead we only used these smallest batch sizes to scientifically study the effect of batch size.

Model	49M	19M[69]	GPT2 (124M)	GPT3 (1.3B)
Accelerator	1 TPU-v4 chip	1 TPU-v4 chip	1 TPU-v4 chip	4 TPU-v4 chips
Training time	$\geq 23$ min	$\geq 29$ min	$\geq 5$ hours	$\geq 56$ hours
Number of training runs	4,059	133	17	6
Total cost (TPU v4 chip hours)	1556	64	85	1344

Table 2: Computational cost of training each model.

## B Additional Results

In Figure 7 we illustrate on a toy problem why momentum is required when the batch size is large but not required when the batch size is small. We run SGD on two variables  $(x, y)$  to minimize the value of a loss function defined as  $x + 10y^2$ . Notice that this loss function is much steeper in the vertical direction compared to the horizontal direction.

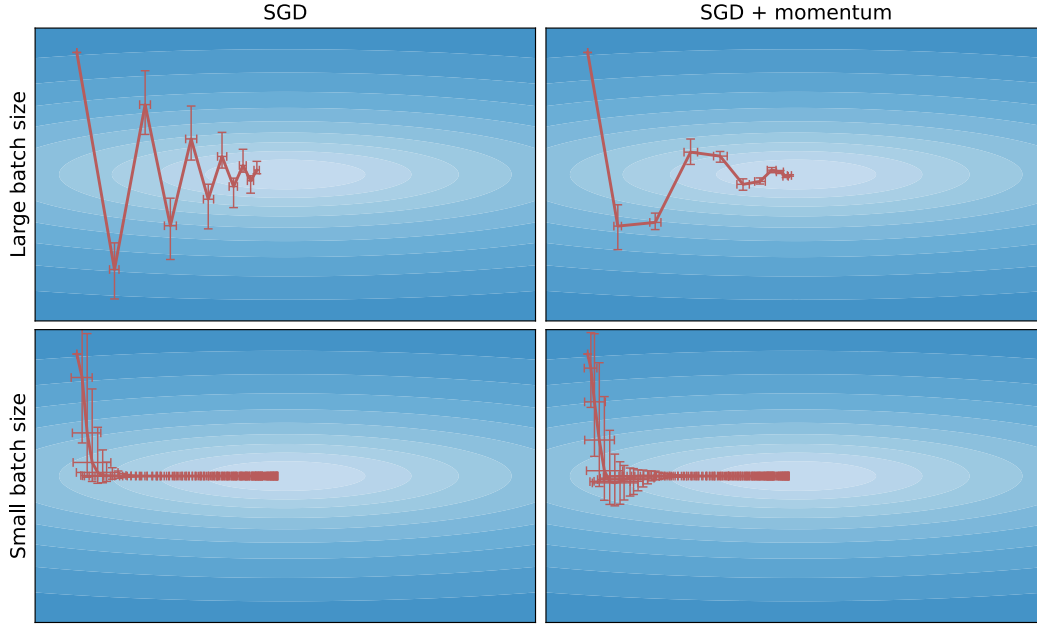


Figure 7: **Toy optimization problem.** We compare the use of SGD and SGD with momentum on a loss function defined as  $x + 10y^2$ . We simulate a small and a large batch size by adding Gaussian noise to the gradient estimate at each optimizer step. The error bars show a 50% interquartile range sampled across different random seeds for the minibatch gradient noise.

We simulate the use of a small and a large batch size by adding noise to the gradient estimate at every step. In particular, we define the minibatch gradient estimate at each step to be the true gradient multiplied by a random variable sampled from a normal distribution  $\mathcal{N}(1, \sigma^2)$ , where the noise scale  $\sigma$  controls the signal-to-noise ratio of the minibatch gradient estimate. We simulate a large batch size by using a high signal-to-ratio of 5, and a small batch size by using a small signal-to-noise ratio of only 0.3.

In our experiments training language models, we fixed the computational budget across all batch sizes by only training for a single epoch. Analogously in this toy example, we only run 10 optimization steps using the simulated large batch size, but we run 100 optimization steps using the small batch size. The idea is that if we use a small batch size, the minibatch gradient estimates at each step become more noisy but in turn we get to take more optimizer steps.

In the top row of Figure 7, we see that when a large batch size is used, we only get to take a small number of steps, and so in turn every step has to be large. Because the loss function is much steeper in the vertical direction compared to the horizontal direction, the large step size results in oscillations along the vertical direction. In this case, using momentum helps dampen the oscillations along the vertical direction and speeds up convergence along the horizontal direction. In fact, in the extreme

942 case of full-batch gradient descent (which has no gradient noise), using momentum probably improves  
943 the convergence rate of gradient descent for this toy problem [20].

944 We show the small batch size setting in the bottom row of Figure 7. Notice that since the batch size is  
945 small, every step becomes more noisy, but in turn we can take more steps and reduce the size of each  
946 step. As a result of decreasing the step size, the optimizer no longer overshoots along the vertical  
947 direction, obviating the need for momentum. There are no longer any oscillations to dampen, so  
948 using SGD with and without momentum results in similar performance.