Transformers can be slow
The paper proposes to run 2 TFs at once for faster results

DeepMind

2023-2-3

Accelerating Large Language Model Decoding with Speculative Sampling

↳ the process of generating the text

Charlie Chen[1], **Sebastian Borgeaud**[1], **Geoffrey Irving**[1], **Jean-Baptiste Lespiau**[1], **Laurent Sifre**[1] **and John Jumper**[1]

[1]All authors from DeepMind

We present speculative sampling, an algorithm for accelerating transformer decoding by enabling the generation of multiple tokens from each transformer call. Our algorithm relies on the observation that the latency of parallel scoring of short continuations, generated by a faster but less powerful draft model, is comparable to that of sampling a single token from the larger target model. This is combined with a novel modified rejection sampling scheme which preserves the distribution of the target model within hardware numerics. We benchmark speculative sampling with Chinchilla, a 70 billion parameter language model, achieving a 2–2.5× decoding speedup in a distributed setup, without compromising the sample quality or making modifications to the model itself.

for forward pass → have to do 70 billion operations for each new token generated

Introduction

Scaling transformer models to 500B+ parameters has led to large performance improvements on many natural language, computer vision and reinforcement learning tasks (Arnab et al., 2021; Brown et al., 2020; Chowdhery et al., 2022; Dosovitskiy et al., 2020; Hoffmann et al., 2022; Rae et al., 2021). However, transformer decoding remains a highly costly and inefficient process in this regime.

Transformer sampling is typically memory bandwidth bound (Shazeer, 2019), so for a given set of hardware, the time to generate a single token in transformer models is proportional to a first order approximation to the size of parameters and the size of the transformer memory. The size of language models also necessitates serving with model parallelism – adding communication overheads (Pope et al., 2022) and multiplying resource requirements. Since each new token depends on the past, many such transformer calls are required to sample a new sequence.

→ generating text based on a probability distribution

We present an algorithm to accelerate transformer sampling for latency critical applications, which we call speculative sampling (SpS). This is achieved by:

2 models: smaller draft model

1. Generating a short draft of length K. This can be attained with either a parallel model (Stern et al., 2018) or by calling a faster, auto-regressive model K times. We shall refer to this model as the **draft model**, and focus on the case where it is auto-regressive.
2. Scoring the draft using the larger, more powerful model from we wish to sample from. We shall refer to this model as the **target model**.
3. Using a modified rejection sampling scheme, accept a subset of the K draft tokens from left to right, recovering the distribution of the target model in the process.

to preserve og accuracy

Intuitively, there are often sequences where the next token might be "obvious". Therefore, if there is strong agreement between the draft and target model's distributions on a given token or sub-sequence of tokens, this setup permits the generation of *multiple tokens each time the target model is called*.

use smaller model to generate "easier" tokens

We show that the expected acceptance rate of draft tokens is sufficient to offset the overhead of the

Corresponding author(s): ccharlie@deepmind.com

drafting process for large language models, resulting in an effective and practical method for reducing sampling latency without the need for modifying the target model or biasing the sample distribution. Depending on the evaluation domain, SpS leads to a 2–2.5× speedup when sampling from Chinchilla (Hoffmann et al., 2022). Notably, the mean tokens per second with SpS often exceeds the idealised ceiling on auto-regressive sampling speed imposed by the memory bandwidth.

Related Work

There has been a substantial body of work focused on improving sampling latency of large transformers and other auto-regressive models.

Since sampling performance is heavily coupled with the model size in memory, quantisation to `int8` or even `int4` (Dettmers et al., 2022; Yao et al., 2022) and distillation (Jiao et al., 2020; Sanh et al., 2019) of transformers are effective techniques for reducing sampling latency with little to no performance penalty. The observation that model size contributes less to the final performance than expected (Hoffmann et al., 2022) has also encouraged smaller language models in general.

During sampling, a cache of the keys and values is maintained for every attention layer, and could become a memory bandwidth bottleneck as the batch size increases. Methods such as multi-query attention (Shazeer, 2019) aims to improve sampling performance by shrinking this cache. However these techniques are most effective at maximising throughout (at larger batch sizes) instead of latency, especially for larger models where the majority of the memory bandwidth budget is consumed by the parameters.

Using a combination of the above techniques, in addition to a number of low-level optimisations to TPUs, Pope et al. (2022) have greatly improved the serving latency and efficiency of PaLM 540B.

There is an existing body of similar work exploiting the efficiency of transformers and sequence models operating in parallel. This includes block parallel sampling (Stern et al., 2018), aggressive decoding (Ge et al., 2022), in addition to some work in parallelizing autoregressive models in the image domain (Song et al., 2021; Wiggers and Hoogeboom, 2020). These methods have yet to be adapted to typical language model use-cases since they either only work with greedy sampling, bias the results or are focused on other modalities. Further, to our knowledge none of these techniques have been scaled to distributed setups, which is necessary for the most expensive decoders with the tens or hundreds of billions of parameters.

Coincidentally, the work in this manuscript was undertaken concurrently and independently of the work on speculative decoding from Leviathan et al. (2022). We focus more heavily the distributed serving setting for large models and offer some incremental optimisations, but otherwise the core underlying idea is the same.

Auto-regressive Sampling

Whilst transformers can be trained efficiently and in parallel on TPUs and GPUs, samples are typically drawn auto-regressively (See algorithm 1). For most applications, auto-regressive sampling (ArS) is highly memory bandwidth bound and thus cannot make effective use of modern accelerator hardware (Shazeer, 2019). A memory bound model call only generates a single token for every sequence in the batch, hence generating multiple tokens introduces a large amount of latency in any system which makes use of it.

This is especially problematic as the number of parameters in the model increases. Since all the model parameters need to pass through at least one accelerator chip, the model size divided by the total memory bandwidth across all chips gives us a hard ceiling on the maximum auto-regressive sampling speed. Larger models also require serving on multiple accelerators, introducing a further source of latency due to inter-device communication overheads.

Algorithm 1 Auto-regressive (ArS) with Auto-Regressive Models

Given auto-regressive target model $q(.|.)$ and initial prompt sequence $x_1, \ldots, x_t$ and target sequence length T.
Initialise $n \leftarrow t$.
while $n < T$ **do**
  Sample $x_{n+1} \sim q(x|x_1, \ldots, x_n)$
  $n \leftarrow n + 1$
end while

Algorithm 2 Speculative Sampling (SpS) with Auto-Regressive Target and Draft Models

Given lookahead K and minimum target sequence length T.
Given auto-regressive target model $q(.|.)$, and auto-regressive draft model $p(.|.)$, initial prompt sequence $x_0, \ldots, x_t$.
Initialise $n \leftarrow t$.
while $n < T$ **do**
  for $t = 1 : K$ **do**
    Sample draft auto-regressively $\tilde{x}_t \sim p(x|, x_1, \ldots, x_n, \tilde{x}_1, \ldots, \tilde{x}_{t-1})$
  end for
  In parallel, compute $K + 1$ sets of logits from drafts $\tilde{x}_1, \ldots, \tilde{x}_K$:

$$q(x|, x_1, \ldots, x_n), \; q(x|, x_1, \ldots, x_n, \tilde{x}_1), \; \ldots, \; q(x|, x_1, \ldots, x_n, \tilde{x}_1, \ldots, \tilde{x}_K)$$

  for $t = 1 : K$ **do**
    Sample $r \sim U[0, 1]$ from a uniform distribution.
    if $r < \min\left(1, \frac{q(x|x_1,\ldots,x_{n+t-1})}{p(x|x_1,\ldots,x_{n+t-1})}\right)$, **then**
      Set $x_{n+t} \leftarrow \tilde{x}_t$ and $n \leftarrow n + 1$.
    else
      sample $x_{n+t} \sim (q(x|x_1, \ldots, x_{n+t-1}) - p(x|x_1, \ldots, x_{n+t-1}))_+$ and exit for loop.
    end if
  end for
  If all tokens $x_{n+1}, \ldots, x_{n+K}$ are accepted, sample extra token $x_{n+K+1} \sim q(x|, x_1, \ldots, x_n, x_{n+K})$ and set $n \leftarrow n + 1$.
end while

Speculative Sampling

Conditional Scoring

For speculative sampling (See algorithm 2), we first make the observation that computing the logits of a short continuation of K tokens in parallel has a very similar latency to that of sampling a single

token. We focus our attention on large transformers, sharded in the Megatron style (Shoeybi et al., 2019). For these models the majority of sampling time can be attributed to three components:

1. **Linear Layers:** For small batch sizes, each linear layer only processes a small number of embeddings. This causes the dense matrix multiplies in the feed-forward layers, queries, keys, values computations and the final attention projection to become memory bound. For small K, this will continue to be memory bound and therefore take a similar amount of time.
2. **The Attention Mechanism:** The attention mechanism is also memory bound. During sampling, we maintain a cache of all the keys and values of the previous tokens in the sequence to avoid re-computation. These KV-caches are large, and accounts for the majority of the memory bandwidth utilisation for the attention mechanism. However, since the KV-cache size does not change as we increase K, there is little to no delta in this component.
3. **All-reduces:** As models grow in size, its parameters need to be divided across multiple accelerators, leading to communication overheads. With Megatron, this manifests itself as an all-reduce after every feed-forward and attention layer. Since only the activations for a small number of tokens are transmitted, this operation is typically latency bound instead of throughput bound for both sampling and scoring (for small K). Again, this results in a similar amount of time spent in the two cases.

Other sources of overhead may exist, depending on the exact transformer implementation. Therefore it is still possible that the choice of positioning encoding, decoding method (e.g. a sort might be required for nucleus sampling), hardware limitations etc. can introduce some deltas between scoring and sampling. However, if the conditions are met such that the above components dominate then scoring should not be significantly slower for small K.

Modified Rejection Sampling

We require a method to recover the distribution of the target model from samples from the draft model, and logits of said tokens from both models.

To achieve this, we introduce the following rejection sampling scheme of the drafted tokens. Given a sequence of tokens $x_1, \ldots, x_n$, and K draft tokens $\tilde{x}_{n+1}, \ldots, \tilde{x}_{n+K}$ generated from $p(.|.)$, we accept $\tilde{x}_{n+1}$ with probability:

$$\min\left(1, \frac{q(\tilde{x}_{n+1}|x_1, \ldots, x_n)}{p(\tilde{x}_{n+1}|x_1, \ldots, x_n)}\right)$$

Where $q(\tilde{x}_{n+1}|x_1, \ldots, x_n)$ and $p(\tilde{x}_{n+1}|x_1, \ldots, x_n)$ are the probability of $\tilde{x}_{n+1}$ according to the target and draft models respectively, conditioned on the context so far.

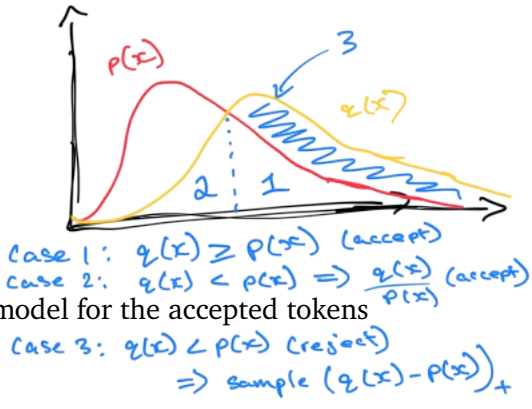
If the token is accepted, we set $x_{n+1} \leftarrow \tilde{x}_{n+1}$ and repeat the process for $\tilde{x}_{n+2}$ until either a token is rejected or all tokens have been accepted.

If $\tilde{x}_{n+1}$ is rejected, we resample x_{n+1} from the following distribution:

$$x_{n+1} \sim (q(x|x_1, \ldots, x_n) - p(x|x_1, \ldots, x_n))_+$$

Where $(.)_+$ denotes:

$$(f(x))_+ = \frac{\max(0, f(x))}{\sum_x \max(0, f(x))}$$



By applying this sequentially, we recover the distribution of the target model for the accepted tokens (see proof in Theorem 1) within hardware numerics. Note that:

Token distribution ends up being exactly q(x)
∴ no loss in accuracy

Best case : generates K+1 tokens
Worst case : generates 1 token

- At least one token will always be generated from a draft-accept loop – if the first token is rejected, a valid token is resampled.
- Since the final token of the draft gives us the logits for the next token, if every drafted token is accepted, we can sample from it normally. This gives us a maximum of $K + 1$ tokens per loop, over the naive implementation which would only return K tokens.

With standard sampling methods such as nucleus, top-k sampling and adjusting temperature, we can modify the probabilities accordingly before applying this rejection sampling scheme. We have observed that the overall acceptance rate is robust to the exact parameters used.

Because we do not interact with the body of the transformer itself, this method can be used in conjunction many other techniques for accelerating or optimising the memory use of sampling, such as quantisation and multi-query attention.

Choice of Draft Models

Since the acceptance criterion guarantees the distribution of the target model in our samples, we are free to choose the method for drafting a continuation as long as it exposes logits, and there is a high enough acceptance rate and/or low enough latency to break-even. There exist several approaches here:

- Incorporating draft generation into the target model, and train the model from the start. This is the strategy used by Stern et al. (2018), which adds multiple heads into the transformer to generate multiple tokens.
- Using sequence level distillation (Kim and Rush, 2016) to generate a second model which predicts K tokens in parallel. This strategy was employed by Ge et al. (2022).
- Set a portion of the activations of the target model as an input to the draft model, and train the draft model with this input.

Although these methods will likely yield powerful drafts, they require a large number of data generated from the target model or changes to the target model. Sequence level distillation in particular would require a large compute budget. This makes them less practical for large scale applications.

Whilst large language models produce better samples, intuitively there are "easier" tokens to predict for which smaller models may be sufficient. Therefore we may simply use a smaller version of the target language model as the draft and obtain high acceptance rates. This would also be convenient from an engineering and workflow perspective, since robust tooling for such models should already exist to train the target model in the first place.

Results

We train a 4 billion parameter draft model optimised for sampling latency on 16 TPU v4s – the same hardware that is typically used to serve Chinchilla for research purposes. This model was trained with the same tokeniser and dataset as Chinchilla, with a slightly smaller width and with only 8 layers. The relatively few number of layers allows it to achieve a sampling speed of `1.8ms/token` compared to `14.1ms/token` for Chinchilla. For details, please refer to the hyperparameters in Table 2.

For distributed setups it is insufficient to naively choose a small model as the draft, since different models have different optimal inference setups. For example, it is typical to serve Chinchilla 70B

Table 1 | **Chinchilla performance and speed on XSum and HumanEval with naive and speculative sampling at batch size 1 and** $K = 4$. XSum was executed with nucleus parameter $p = 0.8$, and HumanEval with $p = 0.95$ and temperature 0.8.

Sampling Method	Benchmark	Result	Mean Token Time	Speed Up
ArS (Nucleus)	XSum (ROUGE-2)	0.112	14.1ms/Token	1×
SpS (Nucleus)		0.114	7.52ms/Token	1.92×
ArS (Greedy)	XSum (ROUGE-2)	0.157	14.1ms/Token	1×
SpS (Greedy)		0.156	7.00ms/Token	2.01×
ArS (Nucleus)	HumanEval (100 Shot)	45.1%	14.1ms/Token	1×
SpS (Nucleus)		47.0%	5.73ms/Token	2.46×

on 16 TPU v4s (where it achieves the aforementioned `14.1ms/token`), whereas a chinchilla-optimal 7B achieves its lowest sampling latency on 4 TPU v4s (where it achieves `5ms/token`). For smaller models, the additional memory bandwidth and flops are insufficient to offset the additional communication overhead between more devices – serving a 7B on 16 TPUs actually *increases* the latency. This means the 7B would provide only a modest speedup if used as a draft with its optimal topology, and we will not make full utilisation of the hardware during drafting.

We can sidestep this issue by training a wider model with a relatively few number of layers in order to minimise communication overhead. It has been observed that the performance of language models is relatively robust to changes in model aspect ratio (Levine et al., 2020), so this allows us to serve a powerful draft model which can be sampled rapidly on the same hardware as the target model.

Evaluation on XSum and HumanEval

We evaluate speculative sampling with Chinchilla on two tasks and summarize the results in Table 1:

- The XSum (Narayan et al., 2018) benchmark. This is a natural language summarisation task using a 1-shot prompt where we sample a total of 11,305 sequences with a maximum sequence length 128.
- The 100-shot HumanEval task (Chen et al., 2021). This is a code generation task involves the generation of 16,400 samples with a maximum sequence length of 512.

Even with greedy sampling, a single token deviating due to numerics could result in two sequences diverging wildly. Since pseudo-random seeds are processed differently between ArS and SpS, and because the different computation graphs lead to different numerics, we cannot not expect identical outputs. However, we expect the samples to come from the same distribution within numerics and we empirically verify this by evaluating these benchmarks.

We run the tasks at batch size 1 with SpS and ArS. The time taken per SpS/ArS loop has low variance, and we can measure it directly from TPU profiles. To obtain the average speedup, standard deviations and other metrics, we log the amount of tokens generated for each speculative loop. In Table 1 we show the performance on the XSum and HumanEval benchmarks for naive and speculative sampling with Chinchilla.

We obtain a substantial speedup in both tasks, with HumanEval reaching speedups of almost 2.5×. Yet, we have parity in the benchmark metrics – the underlying samples distribution is provably the same up to numerics, and this verifies that the draft model is not biasing the results empirically. In the case of HumanEval and greedy XSum, this speedup *exceeded the theoretical memory bandwidth limit of the hardware* for autoregressive sampling (model size divided by the total memory bandwidth).

Acceptance rate changes per domain

It is apparent that the acceptance rate is dependent on the application and the decoding method. HumanEval achieves a significantly larger speedup — We hypothesize that this is due to a combination of code containing a lot of common sub-sequences (e.g. `for i in range(len(arr)):` would be relatively easy for a draft model to guess), is often decomposed into a smaller set of shorter tokens and the temperature value sharpening both the draft and target logits.

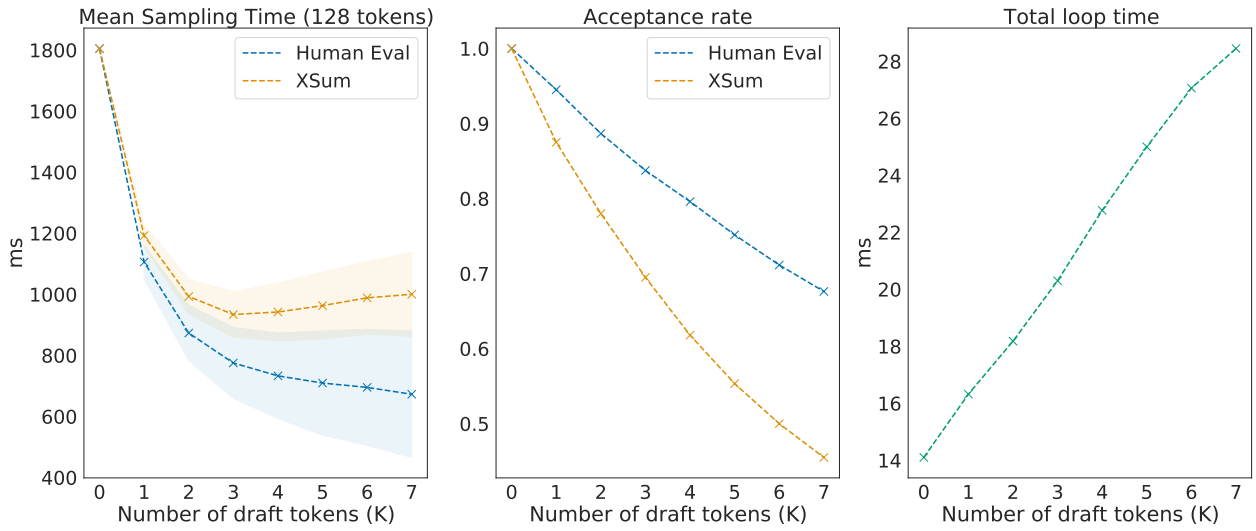


Figure 1 | **Left:** The average time to generate 128 tokens, with standard deviation. Note that as K increases, the overall speedup plateaus or even regresses, with XSum being optimal at $K = 3$. The variance consistently increases with K. **Middle:** The average number of tokens accepted divided by $K + 1$ – this serves as a measure of the overall efficiency of the modified rejection scheme, which decreases with the lookahead. **Right:** Average time per loop increases approximately linearly with K due to the increased number of model calls. Note that the gradient is slightly higher than the sampling speed of the draft model, due to additional overheads in nucleus decoding.

Trade off between longer drafts and more frequent scoring

We visualise the trade-off of increasing K, the number of tokens sampled by the draft model in Figure 1. As K increases, we need fewer scoring calls from the large models to generate the same sequence length, potentially giving us a larger speedup. However, the total loop time increases approximately linearly with the larger number of draft model calls and small increases in the scoring time. The overall efficiency of the proportion of accepted tokens decreases as K increases, since later tokens depend on the acceptance of previous tokens. This results in the average speedup plateauing or even degrading with a larger K (for example, XSum with nucleus's latency is minimised at $K = 3$), depending on the domain.

Further, even though larger values of K may yield marginally greater mean speedups in certain circumstances, it also increases variance of the time to generate a full sequence. This could be problematic for settings where the P90, P99 latencies of concern.

Conclusion

In this work, we demonstrate a new algorithm and workflow for accelerating the decoding of language models. Speculative sampling does not require making any modifications to the target language model's parameters or architecture, is provably lossless within numerics, scales well with the appropriate draft model and complements many existing techniques for reducing latency in the small batch size setting.

We optimise and scale the technique to Chinchilla 70B using a draft model which was easy to train with existing infrastructure, demonstrating that it yields a large speedup across benchmark tasks and common decoding methods in the process. We verify that it is indeed lossless empirically in its downstream tasks.

References

A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lucic, and C. Schmid. Vivit: A video vision transformer. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6816–6826. IEEE Computer Society, 2021.

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

T. Ge, H. Xia, X. Sun, S. Chen, and F. Wei. Lossless acceleration for seq2seq generation with aggressive decoding. *ArXiv*, abs/2205.10350, 2022.

J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, Online, Nov. 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.372. URL https://aclanthology.org/2020.findings-emnlp.372.

Y. Kim and A. M. Rush. Sequence-level knowledge distillation. *CoRR*, abs/1606.07947, 2016. URL http://arxiv.org/abs/1606.07947.

Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. *ArXiv*, abs/2211.17192, 2022.

Y. Levine, N. Wies, O. Sharir, H. Bata, and A. Shashua. The depth-to-width interplay in self-attention. *arXiv preprint arXiv:2006.12467*, 2020.

S. Narayan, S. B. Cohen, and M. Lapata. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1797–1807, Brussels, Belgium, Oct.-Nov. 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1206. URL https://aclanthology.org/D18-1206.

R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.

J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

N. Shazeer. Fast transformer decoding: One write-head is all you need. *CoRR*, abs/1911.02150, 2019. URL http://arxiv.org/abs/1911.02150.

M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Y. Song, C. Meng, R. Liao, and S. Ermon. Accelerating feedforward computation via parallel nonlinear equation solving. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9791–9800. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/song21a.html.

M. Stern, N. Shazeer, and J. Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *CoRR*, abs/1811.03115, 2018. URL http://arxiv.org/abs/1811.03115.

A. Wiggers and E. Hoogeboom. Predictive sampling with forecasting autoregressive models. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10260–10269. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/wiggers20a.html.

Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.

Supplementary Materials

Author Contributions

- **Initial proposal:** Charlie Chen, John Jumper and Geoffrey Irving
- **Initial Implementation, Optimisation and Scaling:** Charlie Chen
- **Modified Rejection Sampling Scheme:** John Jumper
- **Engineering Improvements:** Jean-Baptiste Lespiau and Charlie Chen
- **Experiments:** Charlie Chen, Sebastian Borgeaud and Laurent Sifre
- **Draft of Manuscript:** Charlie Chen and Sebastian Borgeaud
- **Manuscript Feedback:** Laurent Sifre, Geoffrey Irving and John Jumper

Acknowledgements

We'd like to thank Oriol Vinyals and Koray Kavukcuoglu for your kind advice and leadership. We'd also like to thank Evan Senter for your additional feedback on the manuscript and Amelia Glaese for your support in navigating the publishing process. Finally, we'd like to thank Blake Hechtman, Berkin Ilbeyi for your valuable advice on XLA and Nikolai Grigoriev for our discussions on the various tricks that can be applied to the transformer architecture.

Hyperparams

Table 2 | **Hyperparameters** for the draft model

Model	d_{model}	Heads	Layers	Params
Target (Chinchilla)	8192	64	80	70B
Draft	6144	48	8	4B

Proofs

Theorem 1 (Modified Rejection Sampling recovers the target distribution)**.** *Given discrete distributions q, p and a single draft sample $\tilde{x} \sim p$, let X be the final resulting sample. For $X = x$ to be true, we must either sample $\tilde{x} = x$ and then accept it, or resample it after $\tilde{x}$ (of any value) is rejected. Hence:*

$$\mathbb{P}(X = x)$$

$$= \mathbb{P}(\tilde{x} = x)\mathbb{P}(\tilde{x} \text{ accepted} | \tilde{x} = x) + \mathbb{P}(\tilde{x} \text{ rejected})\mathbb{P}(X = x | \tilde{x} \text{ rejected})$$

For the first term, we apply the acceptance rule:

$$\mathbb{P}(\tilde{x} = x)\mathbb{P}(\tilde{x} \text{ accepted} | \tilde{x} = x)$$

$$= p(x) \min\left(1, \frac{q(x)}{p(x)}\right)$$

$$= \min\left(p(x), q(x)\right)$$

For the second conditional term, we apply the resampling rule:

$$\mathbb{P}(X = x | \tilde{x} \textit{ rejected}) = (q(x) - p(x))_+$$

Where $(.)_+$ *denotes:*

$$(f(x))_+ = \frac{\max(0, f(x))}{\sum_x \max(0, f(x))}$$

Finally, we calculate the probability of rejection:

$$\begin{aligned}
\mathbb{P}(\tilde{x} \textit{ rejected}) &= 1 - \mathbb{P}(\tilde{x} \textit{ accepted}) \\
&= 1 - \sum_{x'} \mathbb{P}(X = x', \tilde{x} \textit{ accepted}) \\
&= 1 - \sum_{x'} \min(p(x'), q(x')) \\
&= \sum_{x'} \max(0, q(x') - p(x')) \\
&= \sum_{x'} q(x') - \min(p(x'), q(x')) \\
&= \sum_{x'} \max(0, q(x') - p(x'))
\end{aligned}$$

This is equal to the denominator of $(q(x) - p(x))_+$*, so:*

$$\mathbb{P}(\tilde{x} \textit{ rejected}) \mathbb{P}(X = x | \tilde{x} \textit{ rejected}) = \max(0, q(x) - p(x))$$

Hence:

$$\begin{aligned}
&\mathbb{P}(X = x) \\
&= \min(p(x), q(x)) + \max(0, q(x) - p(x)) \\
&= q(x)
\end{aligned}$$

and we have recovered the desired target.