LayerSkip: Enabling Early Exit Inference and Self-Speculative Decoding

Mostafa Elhoushi[1*†], Akshat Shrivastava[1*†], Diana Liskovich[2†], Basil Hosmer[1], Bram Wasti[2], Liangzhen Lai[3], Anas Mahmoud[4], Bilge Acun[1], Saurabh Agrawal[6], Ahmed Roman[7], Ahmed A Aly[3], Beidi Chen[1,5], Carole Jean-Wu[1],

[1]FAIR at Meta [2]GenAI at Meta [3]Reality Labs at Meta [4]University of Toronto [5]Carnegie Mellon University [6]University of Wisconsin-Madison [7]Dana-Farber Cancer Institute

m.elhoushi@ieee.org akshats@meta.com

Abstract

We present LayerSkip, an end-to-end solution to speed-up inference of large language models (LLMs). First, during training we apply layer dropout, with low dropout rates for earlier layers and higher dropout rates for later layers, and an early exit loss where all transformer layers share the same exit. Second, during inference, we show that this training recipe increases the accuracy of early exit at earlier layers, without adding any auxiliary layers or modules to the model. Third, we present a novel self-speculative decoding solution where we exit at early layers and verify and correct with remaining layers of the model. Our proposed self-speculative decoding approach has less memory footprint than other speculative decoding approaches and benefits from shared compute and activations of the draft and verification stages. We run experiments on different Llama model sizes on different types of training: pretraining from scratch, continual pretraining, finetuning on specific data domain, and finetuning on specific task. We implement our inference solution and show speedups of up to 2.16× on summarization for CNN/DM documents, 1.82× on coding, and 2.0× on TOPv2 semantic parsing task. We open source code at https://github.com/facebookresearch/LayerSkip.

1 Introduction

Large Language Models (LLMs) have been deployed to many applications, yet their high compute and memory requirements lead to high financial and energy costs when deployed to GPU servers Samsi et al. (2023). Acceleration solutions do exist to deploy to commodity GPUs on laptops but they suffer from significant drop in accuracy Zhu et al. (2023). Accelerating LLMs further to mobile or edge devices is still an active research area Çöplü et al. (2023); Liu et al. (2024). While a large portion of LLM acceleration approaches reduce number of non-zero weights Xia et al. (2023) (a.k.a. sparsity), number of bits per weight Xiao et al. (2023) (a.k.a. quantization), number of heads per layer Shim et al. (2021) (a.k.a. head pruning), a smaller portion of approaches focus on reducing number of layers Fan et al. (2020); Elbayad et al. (2020). In this paper, we explore reducing the number of layers required for each token by exiting early during inference. Unlike quantization or sparsity, acceleration by reducing number of layers does not require specialized hardware or software kernels.

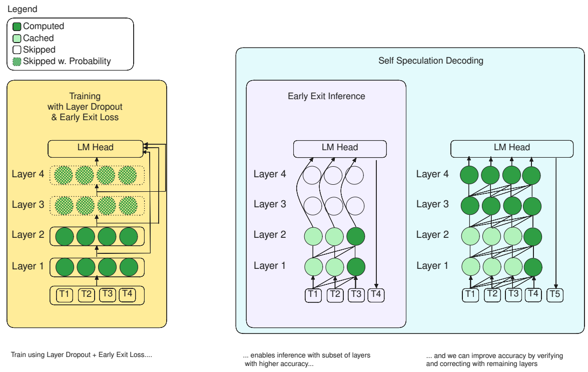


Figure 1: Overview of our end-to-end solution, LayerSkip, showing its 3 components.

Moreover, a popular research trend in LLM acceleration is speculative decoding Leviathan et al. (2023); Chen et al. (2023) that has no drop in accuracy, where a large model, referred to as the *main* model, is accompanied with a faster model, referred to as the *draft* model. The advantage of speculative decoding is that it leads to faster inference compared to the main model, but requires a larger memory footprint and complexity in implementation to maintain key-value (KV) cache in two different models. In addition to exiting early, this paper also proposes combining exiting early with

[*]Equal Contribution.
[†]Core Contributors.

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 12622–12642
August 11-16, 2024 ©2024 Association for Computational Linguistics

speculative decoding to propose a *self-speculative decoding approach* that does not require an additional model or auxiliary layers.

The contribution of this paper is an end-to-end solution:

- a training recipe that combines layer dropout and early exit loss, that leads to,
- inference that is more robust to exiting at earlier layers of the model, essentially creating different sized sub-models within the same model, and
- a self-speculative decoding solution that decodes with earlier layers and verifies and corrects with later layers.

The solution achieves speedups between 1.34× and 2.16× depending on the task. We provide an overview of the solution in Figure 1.

2 Motivation

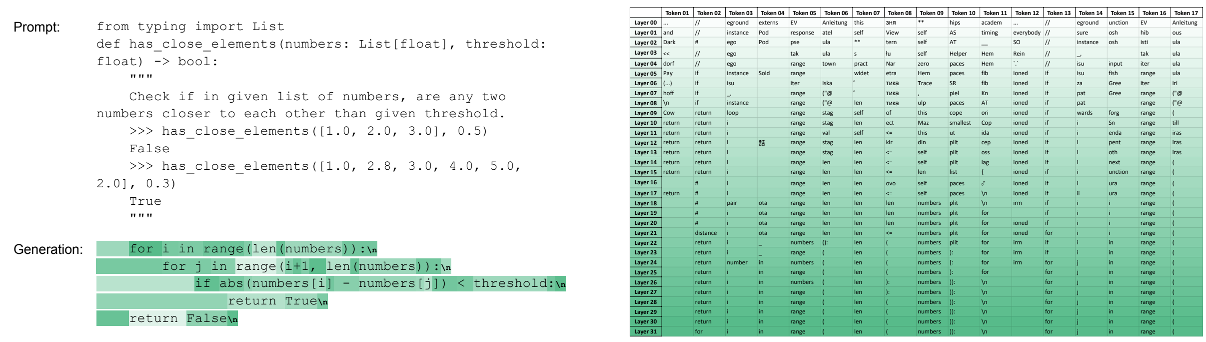
2.1 Exiting Earlier in LLMs

To motivate our approach, we investigate, with an example prompt, what happens in each layer in a LLM. In Figure 2a, we provide the first prompt from the HumanEval coding dataset Chen et al. (2021) to a pretrained Llama1 7B model Touvron et al. (2023a). The prompt consists of a Python function header and a docstring, and the model autocompletes it by defining the function body. When generating each token, we probe each transformer layer in the LLM by projecting its output embeddings on the language model (LM) head (that consists of the model's final layer normalization and linear layer), applying softmax, and then obtaining the index of the output element with highest value. The resulting index corresponds to the predicted token at this layer. This operation is referred to in some literature as the unembedding operation Phuong and Hutter (2022); Cancedda (2024), as it converts an embedding to an index. Unembedding at each layer is equivalent to early-exit at that layer, i.e., it is equivalent to skipping the remaining transformer layers to the model's LM head.

The token predictions across layers in Figure 2b illustrate the evolution of embeddings from an input token fed to the model to the predicted next token by the model. When analyzing the token prediction in each layer in Figure 2b, we make a few observations. First, token predictions in earlier layers appear to be irrelevant as they correspond to the previous token projected on the model's embedding layer's weights, which are different from the weights of the LM head. In later layers, token predictions converge to the final prediction. Second, we do not always need all the layers to predict the correct token. In fact, most of the time, the final token prediction is predicted fewer layers before the end. We also notice that intermediate layers are sometimes hesitant and "change their minds", e.g., for Token 05, the model was predicting "range" as early as Layer 07, but changed its mind between Layer 22 and Layer 26, before settling again on "range".

Similar analysis was done in Geva et al. (2022) on a GPT2 model Radford et al. (2019) as it developed predictors to estimate when prediction saturates to exit early. For the particular example we present in Figure 2, we find, on average, a token requires 23.45 layers out of the model's 32 layers. Hence, even if we have a perfect predictor that has zero compute overhead, we can only save up to 26% of computation. Therefore, there is a need to make LLM models require fewer layers to predict each token, and spend less compute being hesitant or "changing its mind". By default, deep learning models are not motivated to predict their final output early and instead spread their compute across all layers Voita et al. (2019, 2023). We see in Figure 2b, that tokens we would consider easy or straightforward to predict, e.g., Token 02 that starts a for-loop, required all 32 layers to predict "for". We would like our model to be less reliant on later layers and only use later layers for harder tokens. We would like our models to be more reliant on earlier layers than later layers. To do that, we propose skipping layers during training, which we refer to as layer dropout. However, we use higher dropout rates for later layers and lower dropout rates for earlier layers, to make the model less reliant on later layers.

Moreover, LM heads in LLMs are trained to unembed embeddings from the last transformer layer. They were not trained to unembed from earlier layers. Therefore, our solution also adds a loss function during training to make LM heads better "understand" embeddings of earlier layers. While most papers that explored early exit Schuster et al. (2022); Elbayad et al. (2020) trained a dedicated LM head for each transformer layer, and some have introduced additional modules for each early exit Zhang et al. (2019), we chose to have a shared LM head for all transformer layers in the model. This makes training faster, require



(a) (b)

Figure 2: (a) A prompt from the HumanEval dataset Chen et al. (2021) and corresponding text generated by Llama1 7B. The color of each generated token corresponds to the earliest layer in the model that predicted it. (b) Token prediction at each layer in Llama1 7B.

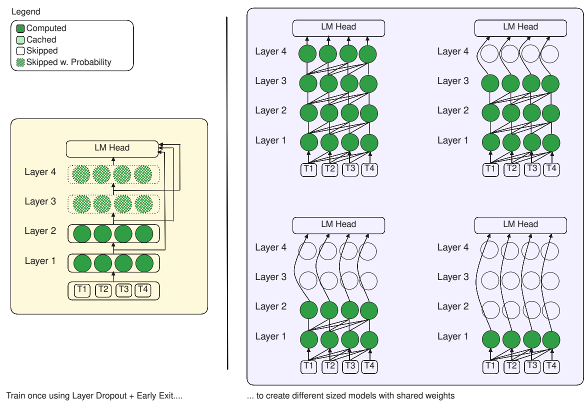


Figure 3: We propose using layer dropout and early exit loss during training to create a model that is equivalent to an ensemble of models of various depths.

less memory consumption for both training and inference, and eases deployment and maintenance. Hence, as shown in Figure 3, we train a deep learning model that is equivalent to an ensemble of models of various depths, capable of skipping from different transfomer layers to the LM head.

2.2 Correcting if we Exit Too Early

Regardless if we use heuristics or predictors (as Schuster et al. (2022); Geva et al. (2022)) to exit early, or if we modify the training procedure to make models predict early (as Elbayad et al. (2020); Zhang et al. (2019) and this paper as well), it is likely that exiting early during inference will lead to a reduction in accuracy. It will be ideal if there is a way to verify if an early prediction is accurate, and correct it by executing remaining layers. Some approaches like Zhang et al. (2019) proposed a confidence heuristic to decide after executing an early exit if the remaining layers are needed. Here, we leverage speculative decoding techniques to verify the early exit prediction and correct it. Speculative decoding benefits from the fact that verifying the prediction of a group of tokens is faster than generating each token auto-regressively. Hence, we present a *self-speculative decoding* approach where we use early exit to generate each token auto-regressively, and use the remaining layers to verify a group of tokens in parallel, and correct them.

3 Related Work

Dropout Dropout was first introduced by Srivastava et al. (2014) and involved stochastically replacing a portion of output elements of fully-connected layers with zeros during training. We refer to this variant of dropout as *unstructured dropout*. It presented a regularization effect for training, with the purpose of reducing over-fitting. Unstructured dropout was commonly used in convolutional neural networks (CNNs) before batch normalization Ioffe and Szegedy (2015) replaced it as a means to improve generalization. However, the introduction of transformers brought it back to light as Vaswani et al. (2017) used a dropout rate of 0.1. However, dropout faded again when pretraining dataset sizes increased, e.g., large scale models like Llama Touvron et al. (2023a) and GPT3 Brown et al. (2020) do not mention dropout in their papers.

Layer Dropout Skipping layers stochastically during training is referred to in literature with different terms such as *stochastic depth* or *layer dropout*. It was first explored in ResNets by Huang et al. (2016) and is used to train ConvNext Liu et al.

(2022).In language models, LayerDrop Fan et al. (2020) applied dropout to every other transformer layer, which increased its robustness to pruning layers at inference time. Zhang and He (2020) increased the pretraining speed of BERT by applying a dropout rate that progressively increased every iteration as well as every layer. To the best of our knowledge, layer dropout for training decoder-only models, or scaling language models to large model sizes or large datasets has not been explored. Moreover, our paper is the first to propose using layer dropout to improve early exit inference.

Early Exit Exiting early in deep learning has first been explored in CNNs Panda et al. (2016); Teerapittayanon et al. (2017). They added branch modules at different exit points in a deep learning network and introduced additional loss functions during training to improve the accuracies of those early exits.

In language models Elbayad et al. (2020) added a dedicated LM head for each decoder layer in an encoder-decoder translation model.CALM Schuster et al. (2022) built upon that and started with a model pretrained with early exit losses, and focused on finding optimal criteria to decide which layer to exit at during inference. Din et al. (2023) started with pretrained models and finetuned auxiliary fully-connected layers to map the embeddings outputted by earlier layers to later layers. In our proposed solution, we do not introduce any additional modules or linear layers for early exit, and instead used a shared exit for all layers.

Speculative Decoding Speculative decoding Leviathan et al. (2023); Chen et al. (2023) is a popular acceleration technique for language models. It is based on the fact that auto-regressive decoding of decoder models are slow as they generate one token a time, while measuring the likelihood of a group of generated tokens in parallel is faster. It uses a fast, less accurate model, referred to as the *draft* model, to generate multiple tokens auto-regressively, and a large, slower, more accurate *main* model, to verify the tokens in parallel, and correct them when needed. The draft model could have the same or different architecture as the main model, or could be a compressed version of the model. Zhang et al. (2023) recently proposed a self-speculative decoding approach where the draft model is the same as the main model, but with a group of intermediate attention and feed forward network (FFN) layers skipped. The advantage of our proposed solution compared to Zhang et al. (2023) is that verification and correction stages can reuse the activation and KV cache from the draft stage as both stages execute the same early layers in the same order, while Zhang et al. (2023) can not reuse them as it skips intermediate layers. Hooper et al. (2024) used shared transformer layer groups and a shared LM head to exit each token at a different layer and execute different layer groups in a pipeline fashion.

4 Proposed Solution

Our approach has three different stages:

1. Training using Layer Dropout & Early Exit Loss
2. Inference using Early Exit
3. Verification and Correction using Speculative Decoding

We explain each stage in the following subsections.

4.1 Training using Layer Dropout & Early Exit Loss

We denote the input tokens to a transformer model as X and its output as Y, with an embedding layer that maps the token indices to token embeddings, x_0, and a transformer model with L transformer layers, where transformer layer l evolves embeddings outputted from its previous layer, $x_{l+1} = x_l + f_l(x_l)$, and a final LM head that maps the embedding outputs of the last layer, x_L to logits, $e_L = g(x_L)$. We denote the cross entropy loss function that is usually used to train language models as $J_{\text{CE}}(e_L, Y)$.

4.1.1 Layer Dropout

The first modification we apply to common training recipes, is to apply layer dropout. Hence the transformer layer operation at layer l and training iteration t changes to:

$$x_{l+1,t} = x_{l,t} + M(p_{l,t})f_l(x_{l,t}) \tag{1}$$

where $p_{l,t}$ is the dropout rate of layer l at iteration t, $M(p)$ is a Bernoulli function that returns 0 with probability p and returns 1 with probability $1 - p$. We apply the dropout operation on each sample separately within a batch. We remove the dropped samples from a batch, apply the transformer operation f_l on the remaining samples, and then concatenate the output with the dropped samples. To ensure higher speedup during training, we seed the random number generator for each GPU with the same seed, so that each transformer layer at each iteration will drop the same number of samples.

The dropout rate can be different at each layer l and training iteration t, $p_{l,t}$:

$$p_{l,t} = S(t)D(l)p_{max} \tag{2}$$

where p_{max} is a hyperparameter that sets the maximum dropout rate in the model during training, $D(l)$ is a per-layer scaling function, and $S(t)$ is a per-time step scaling function. We found that the best per-layer scaling is to increase dropout rate exponentially across layers from 0.0 in layer 0, to 1.0 in last layer, $L-1$:

$$D(l) = e^{\frac{l \ln 2}{L-1}} - 1 \tag{3}$$

For scaling across time, $S(t)$, we found that if we start with a pre-trained model and perform continual pre-training or finetuning, it is best to not scale across time and hence set $S(t) = 1$. However, for pretraining from scratch, we found that an exponential curriculum, $S_{exp}(t)$, lead to best accuracies for T training steps:

$$S_{exp}(t) = e^{\frac{t \ln 2}{T-1}} - 1 \tag{4}$$

4.1.2 Early Exit Loss

To boost prediction accuracy of lower layers, we need to ensure that the model's LM head, g, is capable of unembedding outputs of different layers. Hence, during training, we augment layer dropout with early exit loss at each layer. During training we supervise the model directly to connect the early exit layers to the LM head, this enables us to directly supervise the lower layers for the language modeling task. The total loss of the model at iteration t is:

$$J(X, Y, t) = \sum_{l=0}^{l=L-1} \tilde{e}(t, l) J_{\text{CE}}(g(x_{l+1}), Y) \tag{5}$$

Where $\tilde{e}(t, l)$ is a normalized per-layer loss scale, whose sum across all layers is equal to 1:

$$\tilde{e}(t, l) = \frac{C(t, l)e(l)}{\sum_{i=0}^{i=L-1} C(t, i)e(i)} \tag{6}$$

$C(t, l)$ is a binary curriculum function that determines if we enable early exit of layer l at iteration t. We build upon Elbayad et al. (2020) and set a scale that increases across layers, such as the scale at one layer is proportional to the sum of the scales of all previous layers:

$$e(l) = \begin{cases} e_{scale} \sum_{i=0}^{i=l} i, & \text{if } 0 \leq l < L-1 \\ L - 1 + e_{scale} \sum_{i=0}^{i=L-2} i, & \text{if } l = L-1 \end{cases}$$

This way, we penalize later layers with quadratically higher weight, as predicting in later layers is easier. $0 \leq e_{scale} \leq 1$ is a hyperparameter that controls the scale of early exit loss.

Note that we do not add additional LM heads as proposed in other early exit papers Elbayad et al. (2020); Schuster et al. (2022), as we essentially use the same LM head for all layers.

Early Exit Loss Curriculum We find that adding early exit loss of all layers at all iterations during training slows down training and reduces accuracy of the last layer. To overcome this, we introduce a curriculum, $C(t, l)$. We have explored 2 different curricula. First, we explored a *rotational early exit curriculum*, $C_{\text{rot},R}$, where we enable early exit at every R layers, and perform circular rotation at each iteration. This way, early exit at each layer is enabled once every R iterations. Hence, at each training iteration, only $\lceil L/R \rceil$ unembedding operations are applied. Second, we explored a *gradual early exit curriculum*, C_{grad}, where we gradually enable early exit loss from layers $L-1$ to 0, one layer at a time every $T/2L$ iterations.

4.2 Inference using Early Exit

When generating each token during autoregressive decoding, we run the first E transformer layers in a model, and skip to the model's LM head, i.e., the model's final output becomes $g(x_E)$. We explore with different values of E and provide the accuracies in the Results section.

4.3 Inference using Self-Speculative Decoding

With layer dropout and early exit loss in training, we show it is possible to speedup autoregressive generation by exiting early, but this comes at an accuracy cost compared to using the full model. Speculative decoding Leviathan et al. (2023); Chen et al. (2023) is able to leverage a faster yet less accurate model to speedup generation without accuracy cost. However, this requires storing and training 2 models.

We introduce a novel self-speculative decoding algorithm built on top of early exit, enabling us to

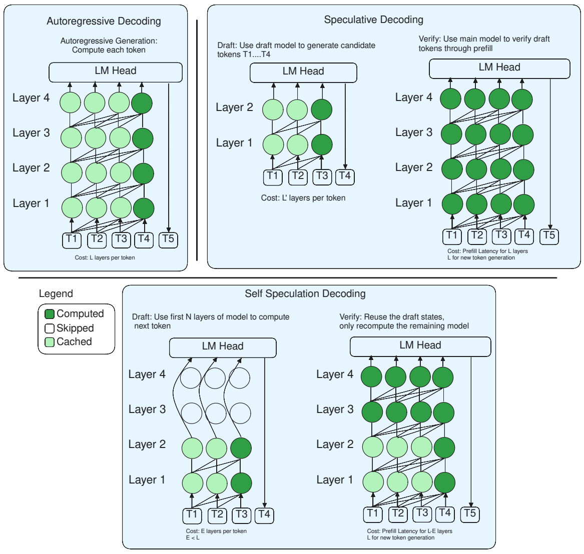


Figure 4: Comparison between autoregressive decoding, speculative decoding, and our proposed self-speculative decoding.

reduce memory through the use of a single model and latency of traditional speculative decoding through re-using hidden states in draft and verify steps. As shown in Figure 4, our self-speculation algorithm consists of 2 key steps (1) *Self-Drafting*, using the early exit to draft tokens from the same model (2) *Self-Verification*, using the remaining layers to validate the prediction. To enable re-use in (1) and (2), we develop a novel *Cache Reuse* technique that unifies the KV cache and storing the exit query. We provide a high level description of the algorithm in sections §4.3.1 and 4.3.2 and provide pseudo code in A.6.

4.3.1 Self-Drafting

The first step in speculative decoding is to define a set of draft tokens $D_{0...d-1}$. In our algorithm, we compute the first d draft tokens through early exit. We refer to d as the number of speculations. We leverage a subset of the LLM and conduct auto-regressive inference exiting at layer E.

Our training recipe enabled us to train the model once to get an ensemble of different candidate draft models at each layer depth. We can evaluate exiting at different layers and observe a trade off between latency and accuracy.

4.3.2 Self-Verification

The next step in speculative decoding is verification. Verification leverages the full LLM to predict the next token for each draft token in a single forward pass. We then assess to see where the draft tokens and verified tokens agree. All the draft tokens up till the disagreement point are added to the output along with the next verified token and the process continues from the draft stage.

In our self-speculative decoding algorithm, the self-verification stage critically only requires computing the remaining layers of the model that were not used in the draft stage. For a model with L layers, the number of verification layers is $L - E$. In order to re-use the first E layers from the draft stage we employ some modifications to the KV cache as we show in the subsequent subsection.

4.3.3 Reusing the Cache

In autoregressive transformers, KV cache is a critical component of efficient generation, allowing us to avoid recomputing prior KV pairs in each layer.

As our draft stage uses the first E layers of the model and the verification stage uses the remaining $L - E$ layers, we are able to re-use a significant amount of compute between the 2 stages:

- **Single KV Cache** As draft and verification stages operate on the same model using the same order of layers, the first E layers are shared in both steps. Hence, in the draft stage, the KV cache in the first E layers are already computed, so we are able to effectively maintain a single KV cache for the draft and verify steps, reducing memory and latency.
- **Exit Query Cache**: To further reduce computation of the first E layers, we introduce an *exit query cache* that saves the query vector of exit layer $E - 1$ for verification to directly continue from layer E to last layer L. Critically note that we need to save only the query for the exit layer. We term the union of the KV cache and the exit query as KVQ cache.

5 Experiments

We would like to evaluate our training recipe on different types of training, whether pretraining from scratch or finetuning. To verify our approach, we run different types of training experiments:

- **Continual Pretraining**: start with a pretrained model and continue pretraining on 52B tokens from a corpus of diverse data containing natural language text and code. We experiment using pretrained Llama2 7B (32 layers), with $p_{\max} = 0.1$, $e_{scale} = 0.2$, $C_{\text{rot},R=8}$, and Llama2 13B (40 layers), with $p_{\max} = 0.1$, $e_{scale} = 0.1$, $C_{\text{rot},R=39}$.

- **Pretraining from Scratch**: start with randomly initialized model and pretrain on 26B tokens from a corpus of diverse data containing natural language text and code. We experiment with Llama2 1.5B (a custom small Llama-like model with 24 layers) (see A.3.1 for architecture details) with $p_{\max} = 0.1$, $e_{scale} = 0.2$, $C_{\text{rot},R=23}$ and Llama2 7B (32 layers) with $p_{\max} = 0.2$, $e_{scale} = 0.2$, $C_{\text{rot},R=31}$. Following Srivastava et al. (2014) we use higher learning rates when layer dropout is greater than 0.0.
- **Finetuning on Code Data**: see §A.2 for details and §A.4 for results.
- **Finetuning on Task-Specific Dataset**: see §A.2 for details and §A.4 for results.

We try different variants of LayerSkip: layer dropout only (LD), early exit loss only (EE), and both layer dropout and early exit loss (LD+EE). We provide more details about training hyperparameters in Appendix A.3.

6 Results

6.1 Early Exit Inference Results

After training each model configuration, we evaluate accuracy of exiting early at different layers.

Continual Pretraining In Figure 5, we present our results for Llama2 7B and 13B on a diverse set of evaluation tasks (see § A.3.2 for task details) and compare with the baseline model from Touvron et al. (2023b). In Table A4 we zoom in and show the specific values of accuracies for the last layer and middle layer of each model. In Figure A1 we show sample text generations for exiting at earlier layers for both models with and without continual pretraining with LayerSkip. Overall, for earlier layers, LayerSkip is clearly better than the baseline. For last layer accuracy, LayerSkip has minimal drop in accuracy compared to baseline.

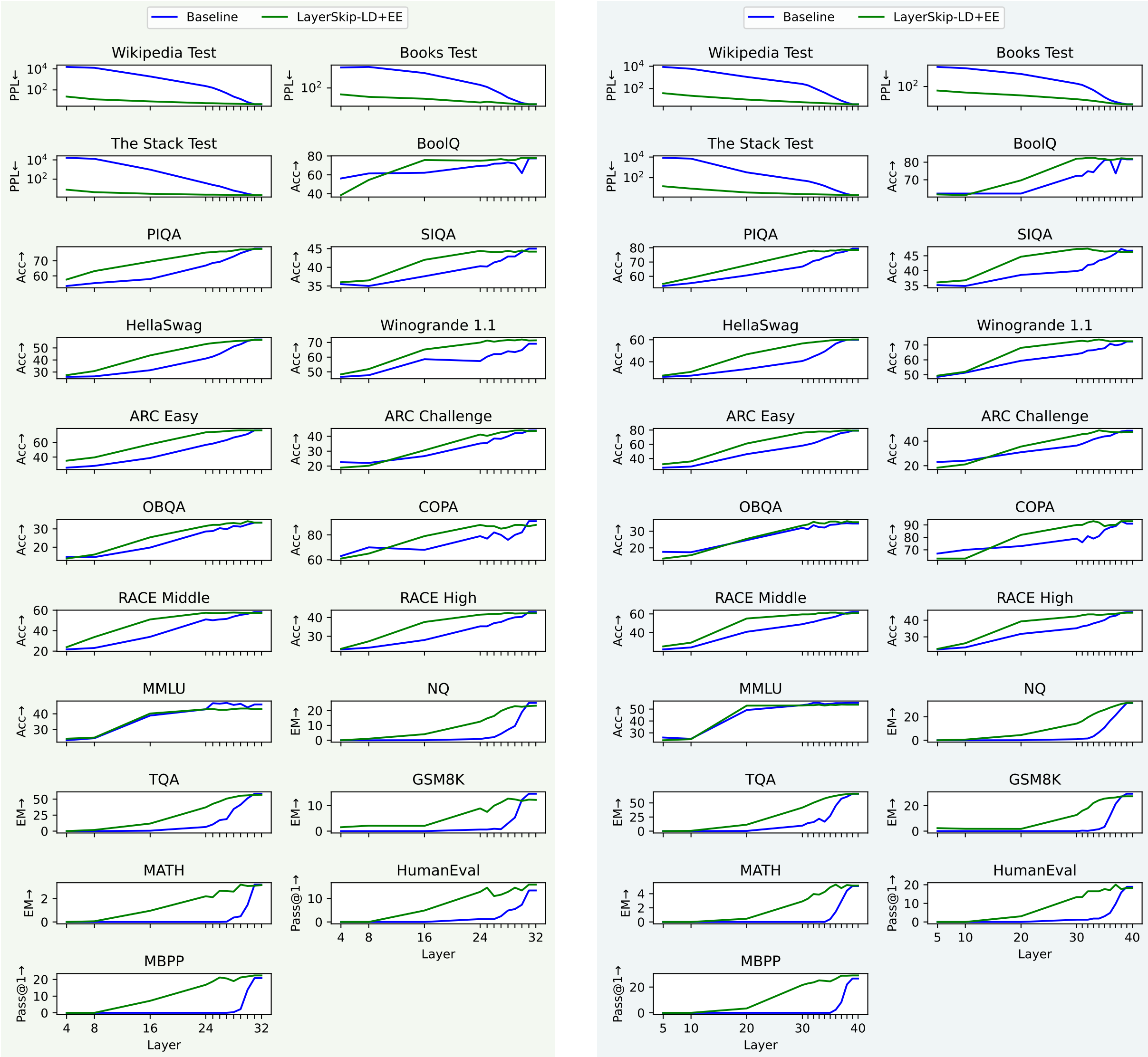
Pretraining from Scratch In Figure A2, we present our results for Llama2 1.5B and 7B pretrained from scratch on 26B tokens using LayerSkip on a diverse set of evaluation tasks (see § A.3.2 for task details) and compare with the same models pretrained on the same number of tokens from scratch without LayerSkip. In Figure A3 we show sample text generations for exiting at earlier layers. Results show that introducing our proposed training recipe leads to higher accuracy than the baseline on earlier layers. On the last layer, we do see a slight drop in accuracy in some downstream tasks, while in other tasks we see LayerSkip leading to higher accuracy.

6.2 Self-Speculative Decoding Results

We evaluate the self-speculative decoding algorithm introduced in §4.3 on different trained models. We report quality metrics, EM (exact match) and ROUGE-2 Ganesan (2018), token acceptance rate for the self speculation algorithm (how often verification accepts each of the draft tokens), throughput measured as tokens per second averaged over the sampled dataset, and speed up compared to autoregressive decoding. For our early exit and our self-speculative decoding experiments, we denote layer we exit at as E. We compare with Draft & Verify Zhang et al. (2023) on common models and tasks evaluated in both papers. All experiments were performed with greedy decoding and generated a maximum of 512 tokens for each sample. Following Zhang et al. (2023), speedup is calculated as acceleration of average inference time per token compared to "Autoregressive" baseline. "Autoregressive" experiments use baseline models that were pretrained or finetuned without LayerSkip, while "Early Exit" and "Self Speculative" experiments use our models trained or finetuned with LayerSkip. Our implementation leverages HuggingFace Wolf et al. (2020).

Continual Pretraining In Table 1, we evaluate the continual pre-training of Llama2 7B and 13B with and without LayerSkip on various tasks: CNN/DM Nallapati et al. (2016), XSUM Narayan et al. (2018) abstractive summarization tasks, and HumanEval Chen et al. (2021) coding task. The experiments were performed on NVIDIA H100 GPUs. The number of speculations, i.e., the number of tokens generated in the draft stage, is denoted d. We obtain speedups between 1.34× and 2.16× depending on model or task. In general, we observe higher speedups for the smaller 7B compared to the larger 13B model. Comparing with Draft & Verify, we are significantly faster on CNN/DM (1.81× vs. 1.5×) and slightly slower on XSUM (1.34× vs. 1.48×).

Pretraining from Scratch Experiments were performed on H100 GPUs and results presented in Table 2. We found an opposite trend to continual pretraining: bigger model has a bigger speedup, reaching 2.16× speedup.



(a) Llama2 7B

(b) Llama2 13B

Figure 5: Early exit evaluation of continual pretraining

7 Ablation Studies

Many ablation studies are in the Appendix, but we summarize some here.

Scaling with Pretraining Tokens Figure A5, shows that without LayerSkip pretraining increases perplexity of earlier layers by orders of magnitude.

KV Cache in Self-Speculation Table A7 shows that our proposed re-use of KV cache consistently saves us 9-20 ms per token depending on the task.

Selecting Parameters for Self Speculation Self speculation relies on 2 core parameters (1) early exit layer and (2) number of speculations. There exists a tradeoff where selecting too low of an exit point and too many tokens are rejected, too high and the latency cost of the exit layer reduces the benefits of speculation. We find that these parameters are task dependent. Figure 6 shows how range of decoding parameters varies for different tasks.

8 Conclusion

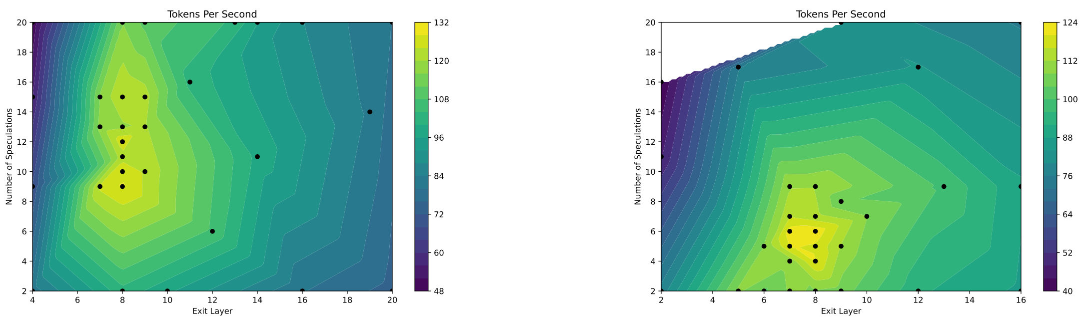
We show that combining layer dropout & early exit loss with curriculum, improves accuracy of early exit during inference, and developed a novel self-speculative decoding solution that led upto 1.86× speedup. We hope this encourages researchers to adopt the proposed recipe in pretraining and fine-tuning. In the future, we can increase accuracy of earlier layers to obtain better speedups for self-speculative decoding, e.g., by combining with dynamic conditions (like Schuster et al. (2022)).

	Llama2 7B						Llama2 13B					
Generation	*E*	*d*	ROUGE-2	Token Acc.	Tokens per Sec.	Speedup	*E*	*d*	ROUGE-2	Token Acc.	Tokens per Sec.	Speedup
CNN-DM One-Shot Abstractive Summarization												
Autoregressive	-	-	0.079	-	62.7	1.00×	-	-	0.098	-	37.2	1.00×
Early Exit	8	-	0.012	-	232.4	-	15	-	0.016	-	105.5	-
Self Speculative	8	12	0.078	68.9%	127.9	**1.86**×	15	12	0.098	74.5%	70.2	**1.81**×
Draft and Verify	n/a	n/a	n/a	n/a	n/a	n/a	-	-	0.107	n/a	n/a	1.56×
XSUM Abstractive Summarization												
Autoregressive	-	-	0.073	-	63.4	1.00×	-	-	0.124	-	43.8	1.00×
Early Exit	8	-	0.002	-	228.0	-	15	-	0.009	-	110.6	-
Self Speculative	8	12	0.073	54.6%	104.7	**1.54**×	15	4	0.124	67.7%	60.5	1.34×
Draft and Verify	n/a	n/a	n/a	n/a	n/a	n/a	-	-	0.126	n/a	n/a	**1.48**×
HumanEval Coding												
Autoregressive	-	-	0.041	-	62.9	1.00×	-	-	0.055	-	48.9	1.00×
Early Exit	8	-	0.003	-	225.4	-	15	-	0.0005	-	244.3	-
Self Speculative	8	6	0.042	67.1%	122.8	**1.83**×	7	4	0.055	57.0%	84.2	**1.66**×

Table 1: Generation results for Llama2 continually pretrained with and without LayerSkip.

	Llama2 1.5B - 26B Tokens					Llama2 7B - 26B Tokens				
Generation	*E*	ROUGE-2	Token Acc.	Tokens per Sec.	Speedup	*E*	ROUGE-2	Token Acc.	Tokens per Sec.	Speedup
CNN-DM One-Shot Abstractive Summarization										
Autoregressive	-	0.063	-	91.6	1.00×	-	0.060	-	64.5	1.00×
Self Speculative	8	0.063	77.4%	167.4	**1.76**×	8	0.067	77.8%	145.6	**2.16**×

Table 2: Generation results for Llama2 pretrained from scratch on 26B tokens with and without LayerSkip.



(a) Llama2 7B CNN-DM Self-Speculation

(b) Llama2 7B HumanEval Self-Speculation

Figure 6: Self Speculation Decoding Parameters Sweep.

9 Limitations

- Our self-speculative decoding solution requires finetuning a model or pretraining it with our recipe, while the self-speculative decoding approach propoposed in Zhang et al. (2023) does not require changing a model's weights.
- The introduced hyperparameters, p_{max} for layer dropout, e_{scale} and R for early exit, requires tuning in order to avoid a drop in last layer accuracy.
- When pretraining with layer dropout from scratch, increasing the learning rate is required to maintain accuracy, and tuning learning rate to get optimal accuracy could be tricky and time consuming.

Acknowledgements

We would like to thank:

- Volker Seeker, Artem Korenev, and Ilia Kulikov for logistic support,
- Fabian Gloeckle, Andrey Gromov, Francisco Massa, Daniel Haziza, Aaditya Singh, Karen Hambardzumyan, Nicola Cancedda, for discussions,
- FAIR's clusters' support team members, especially, Henry Estela, Hongsheng Song, Shubho Sengupta, and Nabib Ahmed, for their help in maintaing our clusters,
- Kamila Benzina, Carolyn Krol, Helen Klein, and Philippe Brunet for legal support.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Nicola Cancedda. 2024. Spectral filters, dark signals, and attention sinks.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, L. Sifre, and John M. Jumper. 2023. Accelerating large language model decoding with speculative sampling. *ArXiv*, abs/2302.01318.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Xilun Chen, Asish Ghoshal, Yashar Mehdad, Luke Zettlemoyer, and Sonal Gupta. 2020. Low-resource domain adaptation for compositional task-oriented semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5090–5100, Online. Association for Computational Linguistics.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *NAACL*.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.

Alexander Yom Din, Taelin Karidi, Leshem Choshen, and Mor Geva. 2023. Jump to conclusions: Short-cutting transformers with linear transformations.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. 2020. Depth-adaptive transformer. In *In Proc. of ICLR*.

Angela Fan, Edouard Grave, and Armand Joulin. 2020. Reducing transformer depth on demand with structured dropout. In *International Conference on Learning Representations*.

Kavita Ganesan. 2018. Rouge 2.0: Updated and improved measures for evaluation of summarization tasks.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Mor Geva, Avi Caciularu, Kevin Wang, and Yoav Goldberg. 2022. Transformer feed-forward layers build predictions by promoting concepts in the vocabulary space. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 30–45, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021a. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021b. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Hasan Genc, Kurt Keutzer, Amir Gholami, and Sophia Shao. 2024. Speed: Speculative pipelined execution for efficient decoding.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. 2016. Deep networks with stochastic depth.

Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Aniruddha Kembhavi, Minjoon Seo, Dustin Schwenk, Jonghyun Choi, Ali Farhadi, and Hannaneh Hajishirzi. 2017. Are you smarter than a sixth grader? textbook question answering for multimodal machine comprehension. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5376–5384.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association of Computational Linguistics*.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. 2017. RACE: Large-scale ReAding comprehension dataset from examinations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 785–794, Copenhagen, Denmark. Association for Computational Linguistics.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. 2024. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. A convnet for the 2020s. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Conference on Empirical Methods in Natural Language Processing*.

Ramesh Nallapati, Bowen Zhou, Cicero Nogueira dos santos, Caglar Gulcehre, and Bing Xiang. 2016. Abstractive text summarization using sequence-to-sequence rnns and beyond.

Shashi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. 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*, Brussels, Belgium.

Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. 2016. Conditional deep learning for energy-efficient and enhanced pattern recognition.

Mary Phuong and Marcus Hutter. 2022. Formal algorithms for transformers. *ArXiv*, abs/2207.09238.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S. Gordon. 2011. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011*. AAAI.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. Winogrande: An adversarial winograd schema challenge at scale.

Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. *arXiv e-prints*, page arXiv:2310.03003.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. 2019. Social IQa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4463–4473, Hong Kong, China. Association for Computational Linguistics.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. 2022. Confident adaptive language modeling. In *Advances in Neural Information Processing Systems*.

Kyuhong Shim, Iksoo Choi, Wonyong Sung, and Jungwook Choi. 2021. Layer-wise pruning of transformer attention heads for efficient language modeling. In *2021 18th International SoC Design Conference (ISOCC)*, pages 357–358.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

PyTorch Team. 2024. gpt-fast. `https://github.com/mostafaelhoushi/gpt-fast`.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. Branchynet: Fast inference via early exiting from deep neural networks.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Elena Voita, Javier Ferrando, and Christoforos Nalmpantis. 2023. Neurons in large language models: Dead, n-gram, positional.

Elena Voita, Rico Sennrich, and Ivan Titov. 2019. The bottom-up evolution of representations in the transformer: A study with machine translation and language modeling objectives.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, Florence, Italy. Association for Computational Linguistics.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2023. Draft & verify: Lossless large language model acceleration via self-speculative decoding.

Linfeng Zhang, Zhanhong Tan, Jiebo Song, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. 2019. Scan: A scalable neural networks framework towards compact and efficient models. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Minjia Zhang and Yuxiong He. 2020. Accelerating training of transformer-based language models with progressive layer dropping. In *Advances in Neural Information Processing Systems*, volume 33, pages 14011–14023. Curran Associates, Inc.

Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. 2023. A survey on model compression for large language models.

Tolga Çöplü, Marc Loedi, Arto Bendiken, Mykhailo Makohin, Joshua J. Bouw, and Stephen Cobb. 2023. A performance evaluation of a quantized large language model on various smartphones.

A Appendix

A.1 LayerSkip Hyperparameters

The hyperparameters of LayerSkip training recipe:

- Layer Dropout:
  - p_{max}: maximum dropout rate of last layer of the model,
  - $S(t)$: layer dropout curriculum. We use either no curriculum $S(t) = 1$ for fine-tuning or continual pretraining, or an exponential curriculum, $S(t) = S_{exp}(t)$ for pretraining from scratch,
- Early Exit Loss:
  - e_{scale}: scalar scale of loss of earlier layers,
  - $C(t, l)$: early exit loss curriculum, either rotational, $C_{\text{rot},R}(t, l)$, or gradual, $C_{\text{grad}}(t, l)$
    - R: is a dilation across layers for rotational early exit loss curriculum

The hyperparameters of LayerSkip self-speculative decoding inference:

- E: layer to exit at during draft stage,
- d: number of speculations, i.e., number of tokens generated during the draft stage autoregressively, that are then verified in parallel during the verification stage by the remaining layers.

A.2 Additional Experiments

In addition to pretraining from scratch and continual pretraining, we evaluate LayerSkip on finetuning on specific domain data in further experiments:

- **Finetuning on Code Data**: start with pretrained Llama1 7B model Touvron et al. (2023a) and finetune on 5.2B tokens of CodeLlama Rozière et al. (2023) data mix. We use $p_{\text{max}} = 0.1$, $e_{scale} = 1.0$, $C_{\text{rot},R=16}$.

- **Finetuning on Task-Specific Dataset**: start with a pretrained Llama 1.5B (24 layers) and finetune on TOPv2 Chen et al. (2020), a multi-domain task-oriented compositional semantic parsing dataset. We post processed the dataset into a JSON format to be more aligned with code pre-training. We report our results on the TOPv2 evaluation set. We use $p_{\text{max}} = 0.2$, $e_{scale} = 1.0$, C_{grad}.

A.3 Experiment Details

We provide details of training configuration and hyperparameters for each of our experiments in Table A1.

When pretraining from scratch, layer dropout leads to higher accuracy when trained on higher learning rate Srivastava et al. (2014). Therefore, we show learning rates of each experiment with and without layer dropout separately in Table A2.

A.3.1 Model Architectures

We provide details of architectures of different models in Table A3.

A.3.2 Evaluation Tasks

We have evaluated our language models on a wide range of tasks. For the sake of discussions in § 6.1, we categorize the tasks into:

- **"Classification" Tasks:** where model responds with one out of pre-defined answers, e.g., multiple-choice questions, or questions whose answers are either "True" or "False":
  - **Common Sense Reasoning Tasks**
    - **BoolQ** Clark et al. (2019)
    - **PIQA** (Physical Interaction Question Answering) Bisk et al. (2020)
    - **SIQA** (Social Interaction Question Answering) Sap et al. (2019)
    - **HellaSwag** Zellers et al. (2019)
    - **Winogrande 1.1** Sakaguchi et al. (2019)
    - **ARC** (Abstraction and Reasoning Corpus) Clark et al. (2018)
      - ARC Challenge
      - ARC Easy
    - **OBQA** (Open Book Question Answers) Mihaylov et al. (2018)
    - **COPA** (Choice Of Plausible Alternatives) Roemmele et al. (2011)
  - **RACE** (ReAding Comprehension dataset from Examinations) Lai et al. (2017)
    - RACE Middle
    - RACE High
  - **MMLU** (Massive Multitask Language Understanding) Hendrycks et al. (2021a)
- **"Generation" Tasks:** where model responds with an open-ended sequence of tokens and we evaluate either exact match of the tokens with a reference answer, or, in case of code, build or execute.

Experiment	Model	Batch Size	Steps	GPUs
Continual Pretraining	Llama2 7B	4	50×10^3	64 A100 80 GB
	Llama2 13B	4	50×10^3	64 A100 80 GB
Pretraining from Scratch	Llama 1.5B	4	50×10^3	32 A100 30 GB
	Llama2 7B	4	50×10^3	32 A100 30 GB
Finetuning on Code Data	Llama1 7B	4	10×10^3	32 A100 80 GB
Finetuning on Task-Specific Dataset	Llama 1.5B	32	5.8×10^3	8 A100 80 GB

Table A1: Training Hyperparameters and Configuration of Experiments

Experiment	Model	Dropout	Initial Learning Rate
Continual Pretraining	Llama2 7B	✓	3×10^{-5}
	Llama2 13B	✓	2×10^{-5}
Pretraining from Scratch	Llama 1.5B		4×10^{-4}
	Llama 1.5B	✓	8×10^{-4}
	Llama2 7B		3×10^{-4}
	Llama2 7B	✓	8×10^{-4}
Finetuning on Code Data	Llama1 7B		1×10^{-4}
	Llama1 7B	✓	1×10^{-4}
Finetuning on Task-Specific Dataset	Llama 1.5B		1×10^{-4}
	Llama 1.5B	✓	1×10^{-4}

Table A2: Learning Rates of Experiments

Model	Dim	Heads	Layers	Context
Llama 1.5B	2048	16	24	4096
Llama1 7B Touvron et al. (2023a)	4096	16	32	2048
Llama2 7B Touvron et al. (2023a)	4096	16	32	4096
Llama2 13B Touvron et al. (2023b)	5120	40	40	4096

Table A3: Model Architectures

- **Question Answering**
  - **NQ** (Natural Questions) Kwiatkowski et al. (2019)
  - **TQA** (Textbook Question Answering) Kembhavi et al. (2017)
- **Mathematics**
  - **MATH** Hendrycks et al. (2021b)
  - **GSM8K** Cobbe et al. (2021)
- **Code Generation**
  - **HumanEval** Chen et al. (2021)
  - **MBPP** (Mostly Basic Python Problems Dataset) Austin et al. (2021)

We also evaluate perplexity on held out test sets on the following datasets:

- **The Stack**, a coding dataset Kocetkov et al. (2022)
- **Books** Gao et al. (2020)
- **Wikipedia**

A.4 Additional Results

A.4.1 Early Exit Results

Continual Pretraining Table A4 zooms into the accuracies of middle and last layers of Llama2 7B and Llama2 13B continual pretraining experiments that are shown in Figure 5. It is noteworthy that some "classification" tasks, i.e., multiple choice question or true/false question tasks, maintain relatively decent accuracy on earlier layers on the baseline model, while open-ended "generation" tasks drop drastically. Surprisingly, MMLU Hendrycks et al. (2021a) which is considered a challenging task, only drops from 55.2% to 49.2% on Llama2 13B baseline from the last to the middle layer. This could be because classification tasks are evaluated on generating one token only while generation tasks are evaluated on the accuracy of many tokens, and an error in one token may have a compounding effect when generating later tokens. Moreover, classification tasks evaluate a token out of 4 or 2 possible outcomes, while generation tasks evaluate each token out of thousands of possible entries in the LLM's dictionary. We observe LayerSkip's significant importance on generation tasks, e.g., NaturalQuestions Kwiatkowski et al. (2019) drops from 25.1% to 0% when exiting in middle layers of Llama2 7B, but jump to 4% when using LayerSkip.

Figure A1 shows sample generations exiting at different layers for Llama2 7B and Llama2 13B

	Llama2 7B				Llama2 13B			
	Last Layer (Layer 32)		Middle Layer (Layer 16)		Last Layer (Layer 40)		Middle Layer (Layer 20)	
	Baseline	LayerSkip	Baseline	LayerSkip	Baseline	LayerSkip	Baseline	LayerSkip
Eval Perplexity ↓								
Wikipedia	4.32	**4.3**	1900	**8.12**	**3.97**	3.98	507	**10.5**
Selected Books	1.60	**1.06**	4390	**6.53**	**1.40**	**1.40**	1170	**11.9**
The Stack	2.15	**2.14**	968	**2.99**	**2.05**	2.06	65.8	**3.71**
Common Sense Reasoning Tasks ↑ (Multiple Choice Questions / True False Questions)								
BoolQ	77.4	**77.8**	62.2	**75.7**	81.6	**82.0**	62.2	**69.7**
PIQA	**78.0**	77.9	57.9	**69.5**	**79.3**	78.5	62.8	**67.8**
SIQA	**44.7**	44.2	37.8	**42.0**	**46.7**	46.3	40.7	**44.7**
HellaSwag	**57.0**	56.6	31.5	**43.8**	60.1	**60.3**	35.6	**46.8**
WinoGrande	69.8	**71.4**	58.6	**65.2**	72.3	**72.5**	59.4	**68.1**
ARC-e	**76.5**	**76.5**	38.6	**57.5**	**79.4**	79.2	48.8	**61.1**
ARC-c	**43.8**	43.6	26.8	**30.6**	**48.3**	47.3	31.9	**35.6**
OBQA	**33.4**	**33.4**	19.6	**25.4**	34.4	**35.4**	23.8	**25.4**
COPA	**90**	88	68	**79**	91	**93**	73	**82**
Reading Comprehension ↑ (Multiple Choice Questions)								
RACE Middle	**58.2**	57.4	34.0	**51.1**	**62.0**	60.7	40.9	**55.1**
RACE High	**42.9**	42.2	28.0	**37.6**	**44.9**	44.5	31.8	**39.3**
MMLU ↑ (Multiple Choice Questions)								
MMLU	**46.0**	43.1	38.9	**40.2**	**55.2**	53.7	49.2	**52.9**
Question Answering ↑ (Open Ended Answers)								
NaturalQuestions	**25.1**	23.2	0.0554	**4.07**	31.5	**31.8**	0.609	**4.43**
TriviaQA	**58.5**	56.8	0.619	**11.8**	66.2	**66.3**	4.36	**11.4**
Mathematics ↑ (Open Ended Answers)								
GSM8K	**14.3**	12.2	0	**2.05**	**29.3**	27.4	0.0758	**1.74**
MATH	**3.22**	3.16	0	**0.96**	5.06	**5.16**	0	**0.46**
Code Generation ↑ (Open Ended Answers)								
HumanEval	13.4	**15.9**	0	**4.88**	**18.9**	18.3	0	**3.05**
MBPP	21.0	**22.4**	0	**7.20**	26.4	**29.0**	0	**3.40**

Table A4: Evaluation of continual pretraining of Llama2 7B and Llama2 13B.

with and without continual pretraining with LayerSkip.

Pretraining from Scratch Figure A3 shows sample generations exiting at different layers for Llama2 7B pretrained with and without LayerSkip.

Finetuning on Code Data In Figure A4a, we present our results on 2 coding tasks and compare accuracy to Llama1 7B finetuned on the same number of code tokens without LayerSkip. For earlier layers, LayerSkip is clearly better than the baseline, with layer dropout combined with early exit loss showing a big improvement on one of the 2 tasks. For last layer accuracy, LayerSkip with both layer dropout and early exit loss has almost the same accuracy as baseline. Note that since this experiment finetuned on specific domain data, we were able to increase e_{scale} to 1.0 (as opposed to $e_{scale} = 0.1$ or 0.2 in the previous two configurations).

Finetuning on Task-Specific Dataset In Figure A4b, we compare results of fine-tuning our Llama 1.5B model on TOPv2 training set with and without LayerSkip. In semantic parsing, correctness requires an exact match (EM) between generated sequence and annotated parse. We find when removing layers from the baseline model,

Baseline

The capital of Egypt is Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian Egyptian
Egyptian Egyptian Egyptian Egyptian Egyptian

```
from typing import List
def add_two_lists(A: List[float], B: List[float]) -> List:
hw genuineershipomy twitch channelpдчингdompairing桥�
bridgebridge bridge bridge bridge bridge bridge bridge bridge
bridge bridge bridgebridge bridge bridge bridge bridgebridge
bridge bridge bridge bridge bridgebridge bridge bridgebridge
bridge bridgebridgebridge bridge bridge bridgebridge bridge
bridge bridge bridgebridge bridge bridgebridgebridge bridge
bridgebridge bridge bridge bridge bridgebridgebridge bridge
bridgebridge bridgebridge bridgebridge bridge bridge
bridgebridge bridgebridge bridgebridge bridge bridge bridge
bridge bridge bridgebridgebridge bridgebridge bridge
bridgebridge bridge bridgebridge bridge bridge
bridgebridgebridge bridgebridge bridge bridgebridgebridge
bridge bridge bridge bridgebridgebridge bridgebridge bridge
bridge bridgebridgebridge bridgebridgebridge bridgebridge
bridge bridgebridgebridge Bridge bridge bridgebridge bridge
bridge bridge bridge bridge bridge bridge bridge
bridgebridgebridgebridge bridgebridge bridgebridge bridge
bridgebridgebridge bridge bridge bridgebridge bridgebridge
bridgebridge Bridgebridge bridge bridge bridgebridge
bridgebridge bridgebridge bridgebridge bridgebridge
bridgebridgebridgebridge bridge bridgebridge Bridge
```

LayerSkip-LD+EE

The capital of Egypt is a big city with a large number of tourist attractions. Growing to be one of the most famous cities in the world, it attracts more than 10 million people a year. At the heart of the city is a complex of ancient monuments, buildings, and wonders. The Egyptian capital is a wonderful city where the history of Egypt is spread out like a palette of colors.
The most famous of all Egyptian monuments is the Pyramid of Giza, which is located at the end of the Giza boulder. It is the only one of three great pyramids that survived the centuries. It is the oldest of the three pyramids and the only one that has three tiers.
Many tourists visit Egypt to see the pyramids of Giza. They are not the only attractions in the city, however. The oldest of the three great pyramids of Giza, which is the oldest of the three great pyramids, has three tiers. Many tourists visit the pyramids of Giza to see the pyramids of Giza. They are not the only attractions in the city, however.
The oldest of the three great pyramids of

```
from typing import List
def add_two_lists(A: List[float], B: List[float]) -> List:
    A_sorted = sorted(A)
    B_sorted = sorted(B)
    new_list = []
    for A_item, B_item in zip(A_sorted, B_sorted):
        new_list.append(A_item + B_item)
    return new_list


def main():
    listA = [1, 2, 3, 4, 5, 6]
    listB = [7, 8, 9, 10, 11, 12]
    listC = add_two_lists(listA, listB)
    print(listC)


if __name__ == "__main__":
    main()
```

(a) Llama2 7B (that has 32 layers) exiting at layer 20.

Baseline

The capital of New York State is Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb
Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb
Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb Alb
Alb

```
def bubble_sort(list_a):
konnichiya gentlemen ladies gentlemen ladies welcome welcome
welcome welcome welcome welcome welcome welcome welcome welcome
welcome welcome welcome welcome welcome welcome welcome welcome
welcome welcome welcome welcome welcome welcome welcome welcome
```

LayerSkip-LD+EE

The capital of New York State is Albany. It is also the capital of Albany County. It was founded on Albany Albany River. The city is situated at the Albany Albany River and is surrounded by Albany Albany River, the Albany Albany River and the Albany Albany River. It is also surrounded by Albany Albany State Forest and Albany Albany State Park. The city has 24 neighborhoods, and the population is about 45,000. There are many clubs, clubs and clubs. There are many clubs, clubs and clubs. There are many clubs, clubs and clubs.

```
def bubble_sort(list_a):
    i = len(list_a)
    for i-1:
        for i-1:
            if list_a[i] > list_a[i-1]:
                swap = list_a[i]
                list_a[i] = list_a[i-1]
                list_a[i-1] = swap
                i = i - 1
    return list_a
```

(b) Llama2 13B (that has 40 layers) exiting at layer 24.

Figure A1: Early exit text generation examples for models continually pretrained with LayerSkip. Blue: The prompt fed into the model. Red: incorrect phrases or words generated by the model (whether factually or grammatically wrong, or hallucinations). With self-speculative decoding, we fix those incorrect phrases by verifying with remaining layers.

the model is not able to generate any complete or accurate. However, with LayerSkip, early exit inference improves to 77% at layer 12. We notice a regression in the final layer reducing accuracy by 3%. Again, as this configuration finetuned data on a specific task, we were able to set $e_{scale} = 1.0$.

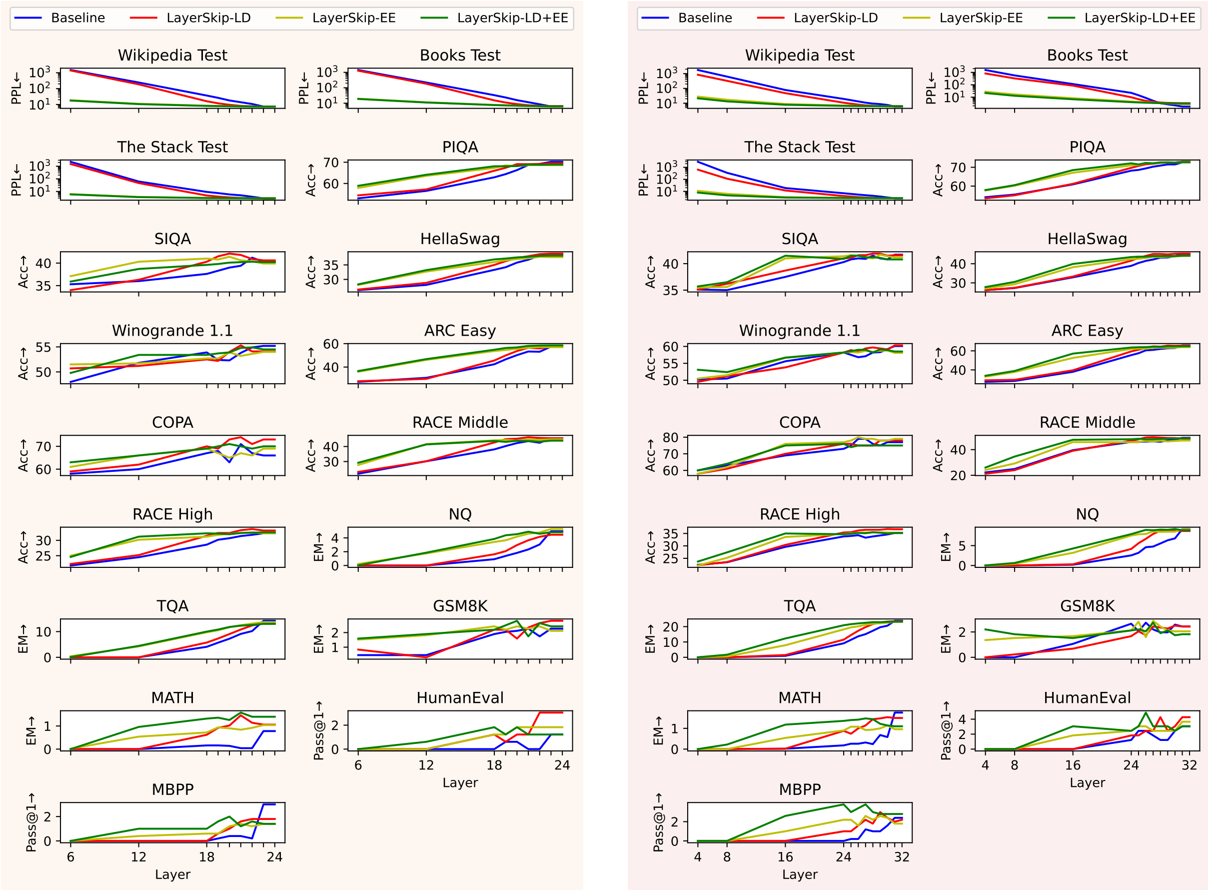
A.4.2 Self-Speculative Decoding Results

Finetuning on Code Data In Table A5, we evaluate our code-finetuned Llama1 7B on HumanEval using 12 speculations, and exit at layer 6 for self speculation & early exit. The experiments were performed on NVIDIA A100 GPUs. We show speedup of upto 1.82× with no accuracy drop.

Finetuning on Task-Specific Dataset In Table A6 we show results for Llama 1.5B finetuned on TOPv2's training dataset and evaluated on TOPv2 test set. The experiments were performed on NVIDIA H100 GPUs. We present the EM (exact match) on the fully TOPv2 test set, further we sample 1000 samples for latency experiments where we leverage 8 speculations, and generate the next 80 tokens with greedy decoding. With self-speculation, the model was able to achieve high token acceptance rate, ($E = 6$: 76.0%, $E = 12$: 97.2%, $E = 18$: 98.9%) reaching 2.0× speedup.

Generation	E	ROUGE-2	Token Acc.	Tokens per Sec.	Speedup
Autoregressive	-	0.0513	-	34	1.0×
Early Exit	6	0.0035	-	170	-
Self Speculative	6	0.0513	45%	62	**1.82**×

Table A5: Generation results on HumanEval for Llama 7B finetuned on code



(a) Llama2 1.5B - 26B tokens

(b) Llama2 7B - 26B tokens

Figure A2: Early exit evaluation of pretraining from scratch on 26B tokens.

Baseline

The capital of Egypt is Egypt capital city is

LayerSkip-LD

The capital of Egypt is Egypt City, which is located on the banks of river Egypt. This city is situated on the banks of River Egypt, which is one of the largest rivers in the world. The city has a population of 1.3 million people and it is also known as the second largest city in Egypt after the capital city of Egypt, Cairo.
The city was built on the banks of River Egypt in 1958. The city is known for its historical monuments and for its museums and museums. The city also has a lot of facilities for tourism. It has several hotels and resorts which are popular among tourists. The city also has a lot of clubs and bars which are popular among tourists. The city also has many shopping centres which are popular among tourists. The city also has a lot of shopping centres which are popular among tourists. The city also has a lot of shopping centres which are popular among tourists. The city also has a lot of shopping centres which are popular among tourists. The city also has a lot of shopping centres which are popular among tourists. The city also has a lot of shopping centres which are popular among tourists. The city

LayerSkip-EE

The capital of Egypt is a country which has a lot to offer to visitors from all over the world. You can visit the monuments of the Egyptian History, the Pyramids of Giza, the Cairo Opera House, the Sphinx, the Giza Plateau, the Aswan Dam, the Valley of the Kings, the Coptic Museum, the Tomb of Tutankhamun, the Medinet Habu, the Giza plateau, the Valley of the Kings, the ancient city of Luxor, the Giza Pyramids and the Hanging Churches.
Cairo is also the center of Arabic culture and the Middle East, and the best way to see it is by seeing the Cairo Opera House, the Cairo International Museum, the Egyptian Museum, the Egyptian national library, the Alexandria Library, the Egyptian National Museum, the Cairo bazaar, the Cairo University, the Khan El-Khalili square, and the Cairo Pyramids and Sphinx.
All these sites are only some of the sites you can see in Egypt. For more information on what to do and where to go, please visit our Egypt Travel Blog.

LayerSkip-LD+EE

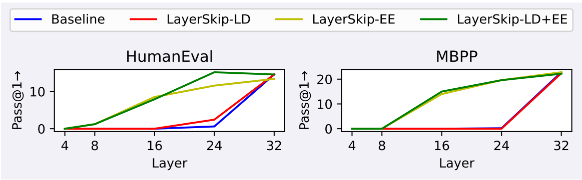
The capital of Egypt is Cairo, located in the northeast of the country. It is a cosmopolitan city, where you can find modern international hotels, traditional inns, and a large number of museums.
The most important sightseeing attractions include the Mohamed Ali Mosque, the Al Kut (Bahari) Mosque, the Said Sadan Mosque, the Abdel Kader Mosque, the Al Mansur Mosque, the El-Hakim Mosque, the El-Qahira Mosque, the Al Khalidi Mosque, the Giza Pyramids, the Dar El-Arqam Mosque, the El Sayed Mosque, the Abu Salim Mosque, the El Karnak Temple, the Al Hakim Mosque, the Cairo Opera House, the El Karnak Catacombs, the El-Mahareb Mosque, the Abdel Moneim El-Khozami Mosque, the El-Qasr Mosque, and the Al Fatah Mosque.
The oldest historical monuments in Cairo include the Al Kahira Mosque, the Al-Kahira Fort, the Al-Azhar Mosque

(a) Llama2 7B (that has 32 layers) pretrained from scratch on 26B tokens only, exiting at layer 24.

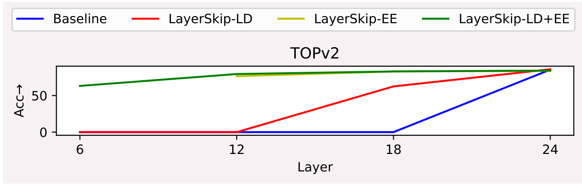
Figure A3: Early exit text generation examples for models pretrained from scratch on 26B tokens with and without LayerSkip. Blue: The prompt fed into the model. Red: incorrect phrases or words generated by the model (whether factually or grammatically wrong, or hallucinations). With self-speculative decoding, we fix those incorrect phrases by verifying with remaining layers.

A.5 Ablation Studies

KV Cache in Self-Speculation In §4.3.3 we introduced the re-use of KV cache as a method for improving model generation speed. We measure its effect in Table A7. We follow the same inference setup as described in §6.2. We find that the use of



(a) Finetuning Llama1 7B on code.



(b) Finetuning Llama1 1.5B on TOPv2 training set.

Figure A4: Early exit evaluation of finetuning on domain-specific or task-specific data.

Generation	E	EM	Token Acc.	Time per Token (ms)	Speedup
Autoregressive	-	85.9%	-	36	1.00×
Early Exit	18	83.3%	-	28	-
Early Exit	12	79.4%	-	19	-
Early Exit	6	62.9%	-	10	-
Self Speculative	18	82.9%	98.9%	29	1.24×
Self Speculative	12	82.9%	97.6%	22	1.64×
Self Speculative	6	82.9%	76.0%	18	**2.0**×

Table A6: Generation results on TOPv2 task for Llama 1.5B finetuned on TOPv2 training data.

Generation	TOPv2 ms/t	CNN/DM ms/t
Self Speculation($E = 18$)	**134**	**166**
w.o KVQ Reuse	143	182
Self Speculation($E = 12$)	**104**	**165**
w.o KVQ Reuse	110	185

Table A7: Ablation on re-use of the KV cache and exit query cache. Results are presented on CPU inference.

KV cache is able to consistently save us 9-20 ms per token depending on the task.

Optimized Implementation The self-speculative decoding performance results were based on HuggingFace Wolf et al. (2020) in eager mode. We have developed another implementation on gpt-fast Team (2024) that optimizes performance using `torch.compile()`. We provide a prompt “Hello, my name is” to our continually pretrained Llama2 7B and measure the average tokens per second running a 1000 times on a single NVIDIA A100 GPU. We also compare with regular speculative decoding where the draft model is 4-bit quantized model. We use the optimal number of speculations and early exit layer based on a sweep for both our self-speculative and speculative decoding solutions. The results are presented in Table A8. We can see that:

- Our proposed self-speculative decoding solution consumes the same memory (total memory for weights, activations, and KV-cache) as the baseline auto-regressive solution, and less than the standard speculative decoding solution that requires 2 models, as we re-use the earlier subset of layers of the model as the draft stage.
- Our proposed self-speculative decoding solution is faster than the standard one. This does not necessarily mean self-speculation is faster than speculation. More experiments on different sized draft models are required to evaluate that.

CPU Inference Experiments We conduct our task specific fine-tuning on Llama 1.5B to measure decoding performance on CPU as well, showing a near 2× speed up on CPU as well, presented in Table A9. We conduct our experiments using the first 100 samples from the TOPv2 test set, leveraging 7 speculations, generating the next 50 tokens with greedy decoding.

Scaling with Pretraining Tokens In order to understand how the accuracy of last and middle layers change across time when pretraining from scratch, we ran 3 training experiments with different number of tokens on Llama 1.5B and show the results in Figure A5. Each experiment trained for 50,000 steps, per device batch size of 4, context window of 4096, but changed the number of GPUs to 32, 64, 128. We plotted the perplexity of a held out split of The Stack dataset on the last layer (layer 24) and the middle layer (layer 12). As expected, perplexity on last layer decreases as we train on more tokens. However, surprisingly, we discover that perplexity on middle layer increases drastically by default in training, unless we apply early exit loss. Layer dropout reduces the increase as well. This could open the door to more research on the dynamics of transformers and the evolution of embeddings in earlier layers to understand why embeddings across layers are close to each other early on in training but diverge drastically as training progresses. This

Generation	Temperature	Draft	d	Total Memory (GB)	Tokens per Second	Speedup
Autoregressive	-	-	-	13.90	108.52	1×
Speculative	0.0	Llama2 7B Int4	5	18.26	125.06	1.15×
Self-Speculative	0.0	$E = 5$	3	**13.90**	**150.07**	**1.38**×
Speculative	0.6	Llama2 7B Int4	5	19.30	122.05	1.12×
Self-Speculative	0.6	$E = 4$	3	**13.90**	**133.98**	**1.23**×

Table A8: Decoding performance evaluation on PyTorch gpt-fast of Llama2 7B continually pretrained with LayerSkip.

Generation	EM	Acceptance	Time per Token (ms)
Autoregressive	85.39	-	165
Early Exit			
$E = 18$	82.0	-	124
$E = 12$	77.2	-	84
$E = 6$	29.8	-	44
Self Speculation			
$E = 18$	82.9	99	134
$E = 12$	82.9	97	104
$E = 6$	82.9	76	87

Table A9: Generation results on CPU for TOPv2 task for small Llama-like finetuned on TOPv2 training data.

could also present a motivation for our training recipe that has minimal drop in last layer accuracy while significantly improves accuracy of earlier layers.

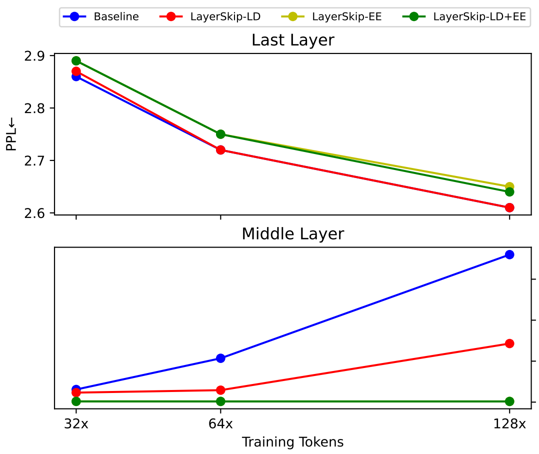


Figure A5: Perplexity on The Stack Kocetkov et al. (2022) test set when pretraining Llama 1.5B from scratch with different number of tokens.

Layer Dropout Configurations In Figure A6 we show that our layer dropout configuration leads to lower loss compared to a constant layer dropout across all layers with the same average value.

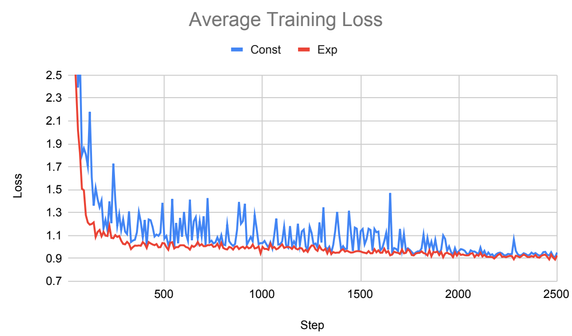


Figure A6: Training loss using different layer dropout configurations. "Const" refers to equal dropout on all layers equal to 0.0889, and "Exp" refers to dropout exponentially increasing from 0 at the first layer to 0.2 at the last layer. Both configurations have equivalent average dropout across all layers.

A.6 Self Speculation Pseudo Code

Below we share pseudo code for implementing self speculation

```
def self_speculate(
    model,
    input,
    num_speculations,
):
    """
    Input Arguments:

    model: Decoder LLM with L layers, supports 2 main
    functions:

    * forward_early: computes inference with
    the first E layers of the transformer,
    saves KV states and the exit layer query cache (kvq)

     * forward_remainder: computes verification
    with the last L-E layers reusing the kvq cache

    input: the input prompt sequence for the model

    num_speculations: the number of speculations from the
         draft forward pass
    """
    # output contains the generations
    # kvq_cache is the kv cache for generation
    # with the addition of the exit layers query
    # cache to speed up verification
    output, kvq_cache = []

    # continue with speculative generation
    # until the maximum sequence length is hit
    while len(output) < max_tokens:
        # produce `num_speculation` draft tokens
        # using the first N layers of the model
        draft_tokens = []
```

```
        draft_input = input

        # conduct num_speculations drafts
        # to produce the draft sequence
        for i in range(num_speculations):
            draft_token = model.forward_e(
                draft_input, kvq_cache
            )
            draft_tokens.append(draft_token)
            draft_input = draft_token

        # verify each of the predictions with the
        # full model inference using a single
        # forward passs
        verified_tokens = model.forward_remainder(
            input + draft_tokens, kvq_cache
        )

        # find the number of accepted tokens
        # by comparing the matches between the draft
        # and verified tokens
        matched_tokens = match(
            draft_tokens, verified_tokens)

        output.extend(matched_tokens)
        # the last matched token doesn't have a cache
        # and needs to be added
        input = matched_tokens[-1]

        # the kvq_cache needs to be updated to remove
        # the last token
        kvq_cache = crop(
            kvq_cache, output_len-1)
    return output
```