Preparing Lessons for Progressive Training on Language Models

SPEAKER: NAGA SAI SIVANI, TUTIKA

AUTHOR: YAN PAN, YE YUAN, YICHUM YIN, JIAXIN SHI, ZENGLIN XU, MING ZHANG, LIFENG SHANG, XIN JIANG, QUN LIU

Introduction

- Training large Transformers consumes significant natural resources, with increasing model sizes leading to longer training times and higher computational costs.
- Traditional methods rely on progressive layer stacking, where layers are added gradually to improve efficiency, but this approach often fails to provide substantial acceleration.
- Apollo introduces a novel solution by utilizing Interpolation and Low-Value-Prioritized Sampling (LVPS) to make training more efficient.
- Unlike conventional methods, Apollo enables lower-level layers to learn high-level functionalities early in training, reducing redundancy and improving model expansion.
- This approach significantly accelerates training, minimizes computational costs, and enhances model scalability without relying on pretrained models.

Background

Existing Approaches & Limitations

- **Progressive Layer Stacking**
  Trains models by gradually adding layers but struggles with slow convergence due to weak knowledge transfer.
- **Pretrained Small Models**
  Improve efficiency but lack flexibility for novel architectures and rely on fixed prior knowledge.
- **General Challenges**
  High training cost and environmental impact remain key issues in scaling Transformer models.

Solution:

- We need a flexible and efficient training approach that reduces computation, accelerates convergence, and adapts to evolving model architectures—all without relying on pretraining.

Apollo

Apollo is a **progressive training method** designed to accelerate Transformer training **without pretraining** by gradually expanding layers using **Low-Value-Prioritized Sampling (LVPS) and Interpolation**.

Efficient Training:

- Apollo gradually expands layers, ensuring stable learning and reduced computation.
- By sharing weights, it prevents instability and accelerates convergence.
- Training follows a progressive expansion rule:

  $$N(s)<N(s+1),$$

  where N(s) represents the number of trainable weights at stage s.

Preparing Lessons:

- To optimize layer expansion, Apollo pre-learns high-level features using Low-Value-Prioritized Sampling (LVPS), which favors shallow layers first, improving efficiency.
- The LVPS probability function is:

  $$P_{LVPS}(L(t)) = \frac{b}{(L(t)+k)^2},$$

  where b and k control sampling efficiency, ensuring balanced computation and strong performance.

$$b = \frac{(N^{(s)}+k)*(L+k)}{L-N^{(s)}},\quad c = \frac{L+k}{L-N^{(s)}}. \qquad (10)$$

Method – Notations

Network Structure:

- $f^{(L)}(\cdot)$: L-layered network function
- $f_l(\cdot)$: Function of the l-th layer
- $f^{(L)}(x)$: Output of the network given input x
- **L(t) : Number of layers at step t selected for weight sharing**
- $f^{(L(t))}$: Network at step t
- **N(s): no. of trainable layers (unfrozen)**

Wights & Parameters:

- $\{\theta_i\}^{i=1 \text{ to } N}$: Set of N weight parameters
- $\theta(f_l)$: Weights of the l-th layer
- g(·) : Mapping function for weight sharing
- $\theta f(_l) = \theta g(l)$: Mapping of weights to layers
- $\theta_l = W_l^{\{Q,K,V,O,IN,OUT\}}$: Weights of layer l

Transformer Layer:

- Input: $Q_l, K_l, V_l \in R^d$ (query, key, value)
- Function: $fl(Q_l, K_l, V_l) = FFN(MHSA(Q_l, K_l, V_l))$

Multi-Head Self-Attention (MHSA):

- $W_l^{Q,K,V,O}$: Attention weight matrices
- $\text{Att}_m(Q_l, K_l, V_l) = \text{softmax}(\frac{Q_l W_{l,m}^{Q} (K_l, W_{l,m})^T}{\sqrt{d_k}}) V_l W_{l,m}^{V} W_{l,m}^{O^T}$
- $\text{MHSA}(Q_l, K_l, V_l) = \sum_{M}^{m=1} \text{Att}_m(Ql, Kl, Vl)$

Method – Notations

Feed-Forward Network (FFN)

- X_l : Input to FFN layer
- $W_l^{IN} W_l^{OUT}$: Weight matrices
- $FFN(X_l) = GeLU(X_l W_l^{IN}) W_l^{OUT}$

Activation & Normalization

- GeLU(·) : Gaussian Error Linear Unit activation
- LayerNorm(·) : Layer normalization

Residual Connections

- Used in MHSA & FFN layers to prevent gradient vanishing and stabilize training.

Method – Apollo Algorithm

Layers N(s) - [1, 3, 6, 12]

Epochs t - [0, 2, 4, 10]

L = 12

Algorithm 1: Process of Apollo

Require: the input data x, the ground-truth y, the stage setting $\{s_t, t \in [1, T], s_t \in [1, S]\}$: a non-decreasing list, indicating the stage of each step.

1: **for** $t = 1$ **to** T **do**
2: **if** $t > 1$ **and** $s_t > s_{t-1}$ **then**
3: **for** $n = 1$ **to** $N^{(s)}$ **do**
4: $\theta_n := \text{COPY}\left(\theta_{g^{N^{(s_{prev})}:N^{(s)}}_{\text{interpolation}}(n)}\right)$
5: **end for**
6: **end if**
7: $L^{(t)} = \text{LVPS}(N^{(s)}, L)$
8: **for** $l = 1$ **to** $L^{(t)}$ **do**
9: $\Theta(f_l) := \text{SHARE}\left(\theta_{g^{N^{(s)}:L^{(t)}}_{\text{interpolation}}(l)}\right)$
10: **end for**
11: $\mathcal{L} = Loss\left(f^{(L^{(s)})}(x), y\right)$
12: $\{\Delta\theta_i\}^{N^{(t)}} = \mathcal{L}.\text{backward}()$
13: Update all the weights $\{\theta_i\}^{N^{(s)}}$ through $\{\Delta\theta_i\}^{N^{(s)}}$
14: **end for**

Ensure: The trained model $f^{(L)}$ with $\{\theta_i\}^L$

Iterating through each step t

Check if the network expanded

Using Interpolation $N^{(s)}$ weights are copied from previous stage to new stage.

Select a number of layers L(t) using LVPS

Share the weights of layers [1,N(s)] to layers [1, L(t)]

Loss calculation, updating gradient descent and updating weights using back propagation.

Method – Apollo

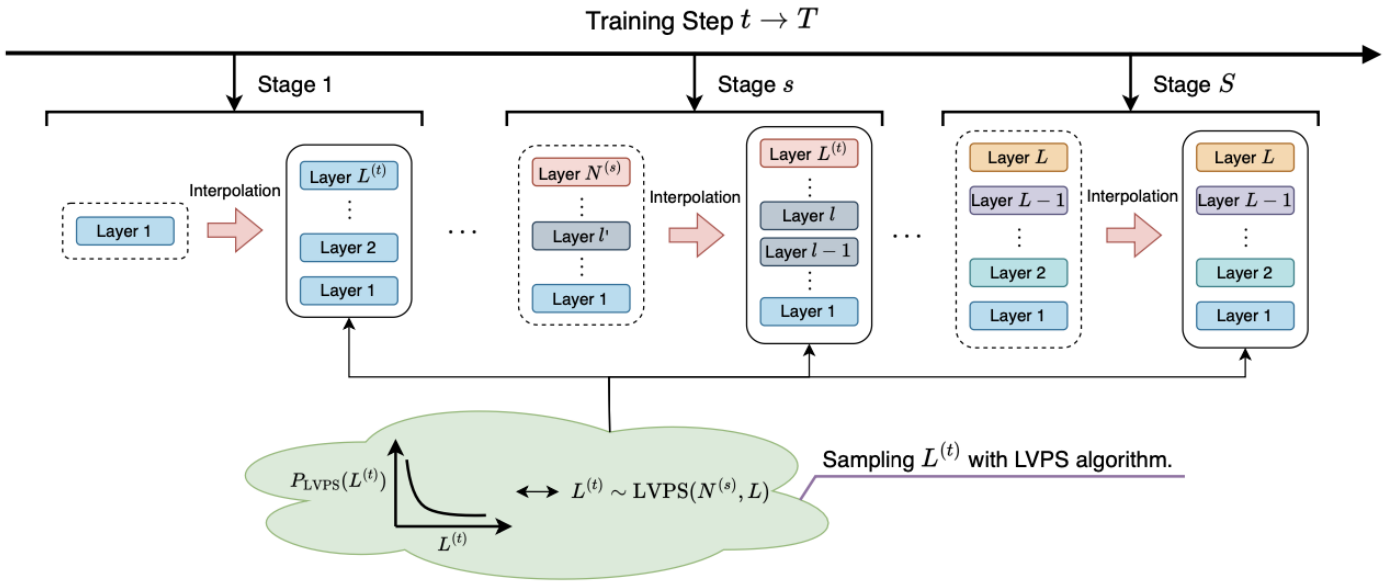


Figure 1: An illustration of the Apollo for training an L-layered model within T steps. We divide this training process into S stages. In the t-th step at the s-th stage, the model weights are $N^{(s)}$ layers (the left layers in each stage in the figure). To let the $N^{(s)}$ layers learn functionality in high layers in advance, we construct $L^{(t)}$ layers (the right layers in each stage in the figure) by sharing the $N^{(s)}$ weights through an interpolation method, where $N^{(s)} \leq L^{(t)}$. As shown in the figure, the same color denotes the same weight. We randomly choose $L^{(t)}$ at t-th step through a probability function Low-Value-Prioritized Sampling (LVPS). Since LVPS tends to select shallower layers, it can greatly save computation costs. Furthermore, we progressively increase the $N^{(s)}$ weights when stepping into the next stage. Since weights in the early stage can learn the properties of higher layers, Apollo can significantly contribute to the training efficiency.

Experiments - Environment

Aspect	Details	
Model	BERT	GPT
Optimizer	AdamW	
Learning Rate	10^{-4}	
Weight Decay	10^{-2}	
Batch sizes	768	512
Baselines	Scratch model, StackBERT, bert2BERT, LiGO	
Apollo layer numbers	[1, 3, 6, 12]	
Apollo Layer Change Epochs	[2, 4, 10]	
LiGO warm up steps	100	
Training Dataset	English Wikipedia + Toronto Book Corpus (Zhu et al., 2015)	
Epochs (T)	40	35

Config	BERT-Small	BERT-Base	BERT-Large	GPT-Small	GPT-Base
# layers	12	12	24	12	12
# hidden	512	768	1024	512	768
# heads	8	12	16	8	12
# vocab	30522	30522	30522	50257	50257
seq. length	512	512	512	1024	1024

Table 5: The structures of BERT and GPT.

Experiments – Expanding Method

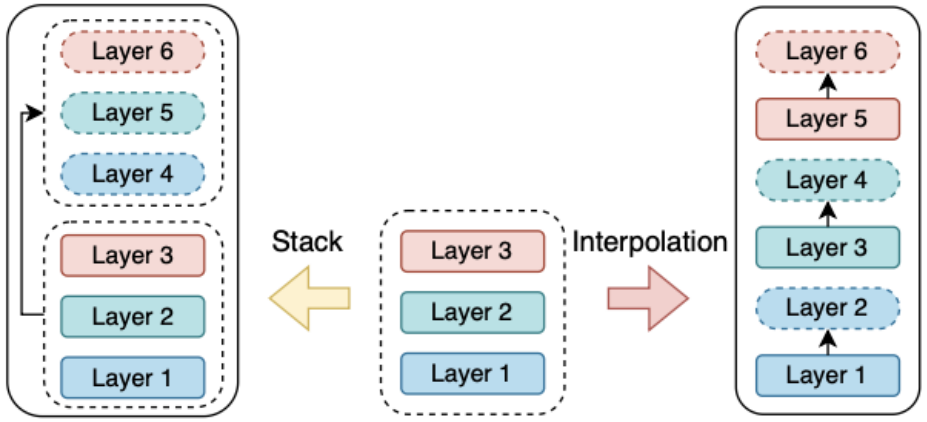


Figure 4: A case of expanding 3 layers to 6 layers. The same color denotes the same weight. The stacking method recurrently arranges the layers, e.g., the 1-st layer → the 4-th layer. By contrast, the interpolation method arranges the layers in a neighbor, e.g., the 1-st layer → the 2-nd layer.

Stacking: Directly copies lower-layer weights into higher layers, causing abrupt weight transitions and instability in training.

$$g_{\text{stack}}^{L_1:L_2}(l_2) = l_2 \bmod L_1, \tag{11}$$

Interpolation: Smoothly distributes weights across expanded layers, ensuring gradual transitions and stable training.

$$g_{\text{interpolation}}^{L_1:L_2}(l_2) = \lfloor \frac{l_2 * L_1}{L_2} \rceil, \tag{12}$$

Experiments – Expanding Method

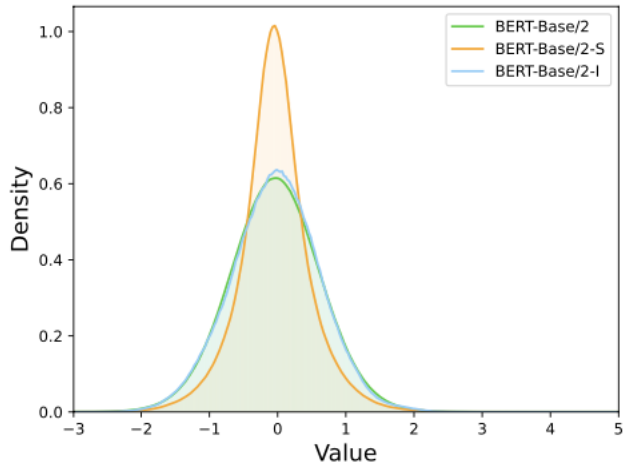


Figure 5: Distribution of output activations. BERT-Base/2 is half of BERT-Base. BERT-Base/2-S and BERT-Base/2-I denote to stack and interpolate BERT-Base/2 to BERT-Base, respectively. After stacking BERT-Base/2, the distribution of output activations changes a lot, while the interpolation method keeps the distribution well.

- Output activation values indicate how well neurons learn and generalize.
- In the graph, **BERT-Base/2** represents a 6-layer BERT-Base model, with **BERT-S (Stacking)** and **BERT-I (Interpolation)** as expansion methods.
- Stacking **significantly alters** the activation distribution from the original model, while Interpolation **preserves it**, ensuring stable training.

Experiments – Expanding Method

Model	Trainable	Layer	Loss	Gradient (1e-3)
BERT-Base Ra.	-	12	10.56	11.45±26.46
BERT-Base/2	-	6	1.82	1.66±4.71
BERT-Base/2-S	✗	12	7.32	55.28±241.52
BERT-Base/2-I	✓	12	3.97	44.01±225.76

Table 2: Results of the analysis of expanding methods. BERT-Base Ra. means a randomly initialized BERT-Base. When expanding BERT-Base/2 from 6 to 12, the loss and gradients of the stacking method rise sharply, which causes failure in further training. By contrast, the interpolation method achieves smaller gradients and loss and is thus trainable later. However, both of the two methods cause higher gradients than a random model, indicating an unstable state.

Both methods yield high gradients, but interpolation has lower loss and is slightly more stable than stacking.

Model	Acc. Ratio	Layer	Loss	Gradient (1e-3)
Apollo-S	39.7%	6	1.82	1.65±3.72
		12	1.76	1.42±3.50
Apollo-I	41.6%	6	1.82	1.59±3.33
		12	1.76	1.39±3.01

Table 3: Results of expanding methods for Apollo. "S" and "I" mean the stacking and the interpolation methods, respectively. Apollo can decrease the gradient values loss after expanding layers since learning the functionality of a 12-layer network. Moreover, the interpolation method can derive a comparably better acceleration ratio in terms of FLOPs and smaller gradient values.

In Apollo, interpolation achieved better acceleration with smaller gradients and lower variance.

Experiments – Sampling Methods

Apollo uses LVPS for sampling. To evaluate its effectiveness, the paper compares it with other sampling methods.

Method	Name	Description	Probability Density Function
LVPS	Low Value Priority Sampling	Prefers **shallower layers** to reduce computation while occasionally sampling deeper ones.	$P_{LVPS}(L(t)) = \frac{b}{(L(t)+k)^2}$
ES	Edge Sampling	Samples **lowest and highest layers**, skipping middle layers for efficiency.	$P_{ES}(L(t)) = \frac{1}{k}\left(\frac{1}{L(t)-N(s)-b}+\frac{1}{L+b-L(t)}\right)$
US	Uniform Sampling	**Equal probability** for all layers, ensuring fairness but increasing computation.	$P_{US}(L(t)) = \frac{1}{L=N(S)}$
FS	Full Sampling	Always selects the **deepest layer**, maximizing learning but costly in resources.	$P_{FS}(L) = 1$

Since LVPS is the chosen sampling method, k=0 is used in all experiments for consistency, as it requires the least computation. For Edge sampling k = 10.

Experiments – LVPS

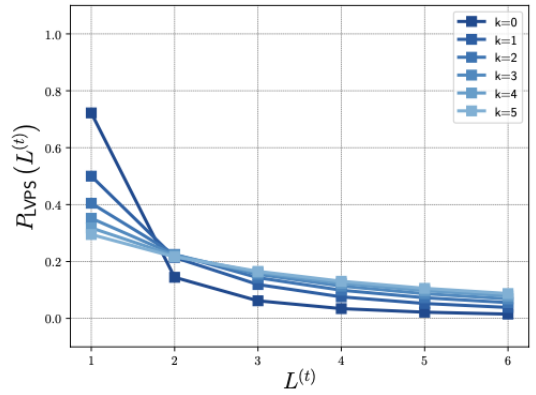


Figure 2: A case of choosing hyper-parameters k of LVPS to sample 1-6 layer number.

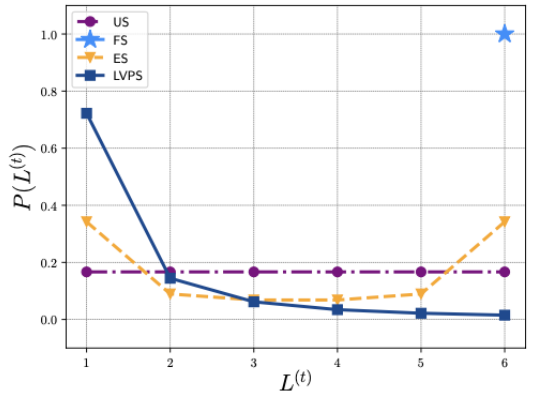


Figure 3: Comparison among US, FS, ES, and LVPS to sample 1-6 layer number.

Higher k → More balanced sampling across all layer depths (flatter distribution).
Lower k → Stronger preference for shallower layers (steep drop, especially high probability on layer 1).

FS: Full Sampling always selects layer 6.
US: Uniform Sampling gives equal probability to all layers.
ES: Edge Sampling favors layers 1 and 6. (k = 10)
LVPS: LVPS prioritizes shallower layers with a gradually decreasing probability for deeper ones.

Experiments – Sampling Methods

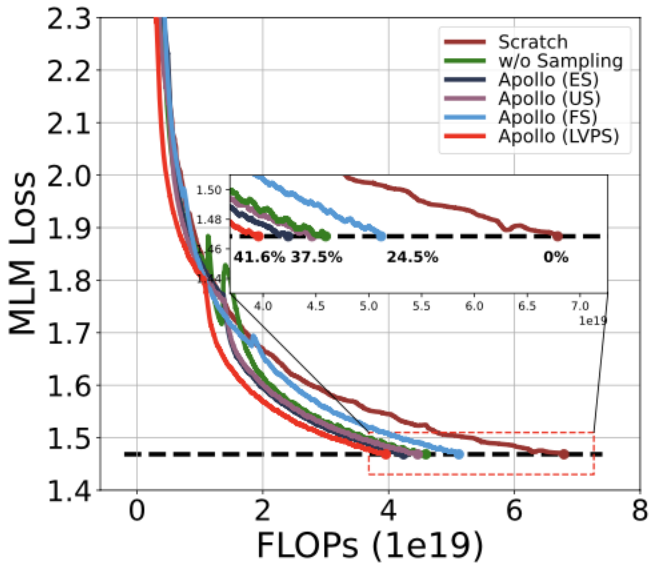


Figure 7: Results of the analysis on sampling methods. LVPS achieves the highest acceleration ratio at 41.6%, surpassing ES and FS at 4.1% and +17.1%, respectively. Most sampling methods are faster than Apollo w/o Sampling. FS performs the lowest acceleration by always sampling the deepest depth, which is resource-consuming.

- Apollo was tested with different sampling methods, Apollo without sampling, and a scratch model.
- LVPS achieved the highest acceleration (41.6%), proving it to be the most efficient method.
- Apollo without sampling still reached 37% acceleration, showing Apollo's effectiveness even without LVPS.
- Apollo with LVPS outperformed Full Sampling (FS) by 17.1% and Edge Sampling (ES) by 4.1%, while Uniform Sampling (US) lagged behind at only 24.5%.

Experiments – BERT & GPT

Next, Apollo was tested on two different architectures to evaluate its performance.

The selected models were:

- BERT
- GPT

Training Setup:

- StackBERT, Apollo, and Scratch were trained from scratch.
- bert2BERT and LiGO were trained by expanding from BERT-Small to BERT-Base.
- Training epochs: 40 for BERT, 35 for GPT.

Results – BERT

Model	Saving (FLOPs)	Saving (Wall Time)	SQuADv1.1 (F1)	SQuADv2.0 (F1)	SST-2 (Acc)	MNLI (Acc)	MRPC (Acc)	COLA (Mcc)	QNLI (Acc)	STS-B (Acc)	QQP (Acc)	GLUE Avg.	SQuAD Avg.
Scratch	-	-	89.05	77.49	92.04	84.05	87.65	56.95	91.39	89.16	91.17	84.63	83.27
Training from the Pretrained Model: BERT-Small→BERT-Base													
bert2BERT	35.6%	35.2%	90.02	78.99	92.89	84.92	86.91	60.32	91.81	88.11	90.72	85.10	84.50
LiGO	33.5%	33.2%	90.09	78.34	92.75	84.99	87.44	61.10	91.33	87.94	90.42	85.14	84.22
Progressive Training form Scratch													
StackBERT	29.5%	28.9%	89.82	78.21	92.94	84.63	87.65	61.61	90.95	87.13	90.20	85.01	84.01
Apollo	**41.6%**	**41.1%**	89.87	78.42	92.28	84.81	87.06	60.57	91.43	88.27	90.69	85.02	84.15

Table 4: Experiments on downstream tasks of BERT-Base on GLUE (Wang et al. 2019a), SQuADv1.1 (Rajpurkar et al. 2016), and SQuADv2.0 (Rajpurkar, Jia, and Liang 2018) dataset. The terms "Training from the Pretrained Model" and "Progressive Training from Scratch" denote the pretraining type of the methods in the table. Compared with baselines, Apollo can achieve the highest FLOPs saving under similar downstream performance, even better than training from the pretrained model.

SQuAD (F1 Score): Evaluates question-answering accuracy.
GLUE Benchmark Tasks:

- Test NLP performance across:
  - SST-2: Sentiment analysis.
  - MNLI: Sentence entailment classification.
  - MRPC: Paraphrase detection.
  - COLA: Grammar correctness.
  - QNLI: Question answering relevance.
  - STS-B: Text similarity.
  - QQP: Duplicate question detection.

Apollo showed significant acceleration of 41% from scratch and showed improved metrics in some cases.

Results – BERT

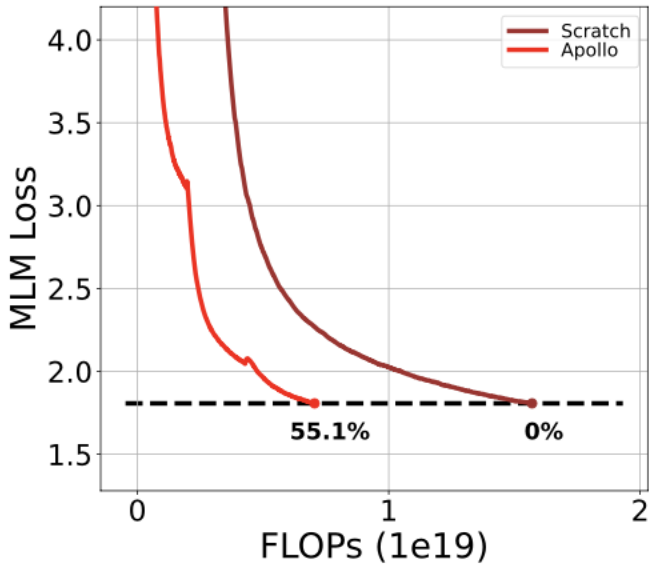
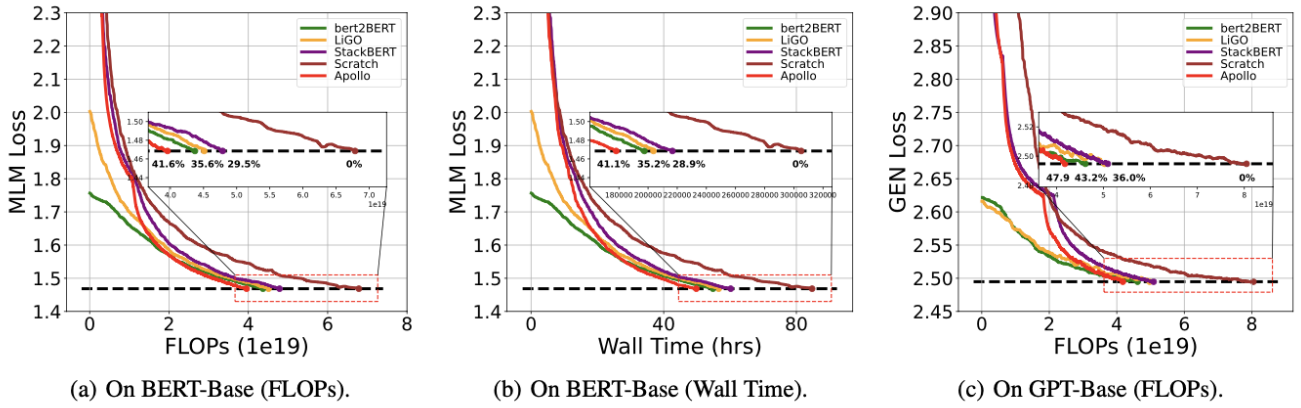


Figure 8: Results of BERT-Large.

- BERT-Large trained from scratch and BERT-Large trained using Apollo was compared without using any pre-trained model.
- The results show that Apollo is **55.1% faster** than training from scratch.

Results



(a) On BERT-Base (FLOPs). (b) On BERT-Base (Wall Time). (c) On GPT-Base (FLOPs).

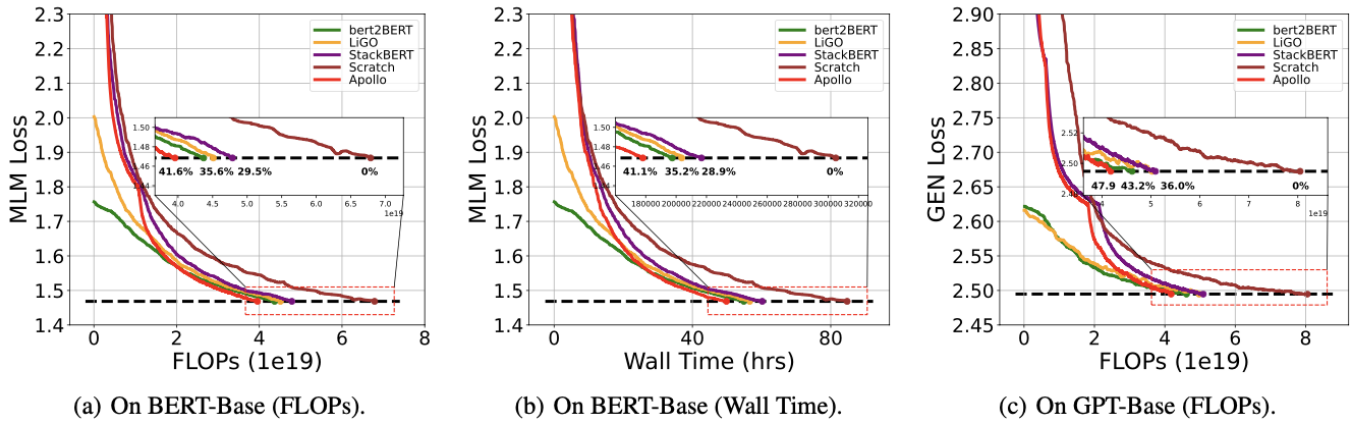
Figure 6: Results of BERT-Base and GPT-Base. Apollo achieves the highest acceleration on BERT-Base and GPT-Base in terms of FLOPs at 41.6% and 47.9%, respectively. In addition, Apollo can keep the best training efficiency on wall time for BERT-Base at 41.1%. Apollo surpasses methods (i.e., bert2BERT and LiGO) relying on pretrained models in all cases.

FLOPs – theoretical calculation
Wall time – actual time taken in real application

BERT

- Apollo achieved 41.6% FLOPs saving and 41.1% in wall time, surpassing both bert2BERT and LiGO, despite their pretraining advantage. And outperforming other methods.
- LiGo and bert2BERT were however faster than stackBERT.
- It also converged faster, proving its efficiency.

Results



(a) On BERT-Base (FLOPs). (b) On BERT-Base (Wall Time). (c) On GPT-Base (FLOPs).

Figure 6: Results of BERT-Base and GPT-Base. Apollo achieves the highest acceleration on BERT-Base and GPT-Base in terms of FLOPs at 41.6% and 47.9%, respectively. In addition, Apollo can keep the best training efficiency on wall time for BERT-Base at 41.1%. Apollo surpasses methods (i.e., bert2BERT and LiGO) relying on pretrained models in all cases.

GPT

- Apollo achieved 47.9% acceleration, the highest among all models.
- Apollo outperformed StackBERT (+11.9%), proving its effectiveness in progressive training.
- Apollo also surpassed bert2BERT (+6.8%) and LiGO (+9.9%), showing its superior training efficiency.
- Despite differences between GPT and BERT architectures, Apollo remained consistently the best performer.

Conclusion

- Apollo accelerates training while maintaining model performance, reducing computational and financial costs.
- LVPS, weight sharing, and interpolation improve efficiency and stability, enabling faster convergence.
- Achieves 41.6% FLOPs reduction on BERT-Base, 47.9% on GPT-Base, and 55.1% on BERT-Large, significantly speeding up training.
- Outperforms pretrained-based methods while remaining universally applicable to new architectures.
- Supports sustainable AI by reducing training overhead and enabling efficient deep learning.

References

Paper:
Pan, Y., Yuan, Y., Yin, Y., Shi, J., Xu, Z., Zhang, M., Shang, L., Jiang, X., & Liu, Q. (2024). *Preparing Lessons for Progressive Training on Language Models*. arXiv preprint arXiv:2401.09192. https://arxiv.org/abs/2401.09192
Code: https://github.com/yuanyehome/Apollo-AAAI-2024-Release

Thank you!

Appendix

$$b = \frac{(N(s)+k)\times(L+k)}{L-N(s)}$$

Structures of Language Models

We show structures of the used language models including the BERT and GPT models in Table 5.

Config	BERT-Small	BERT-Base	BERT-Large	GPT-Small	GPT-Base
# layers	12	12	24	12	12
# hidden	512	768	1024	512	768
# heads	8	12	16	8	12
# vocab	30522	30522	30522	50257	50257
seq. length	512	512	512	1024	1024

Table 5: The structures of BERT and GPT.

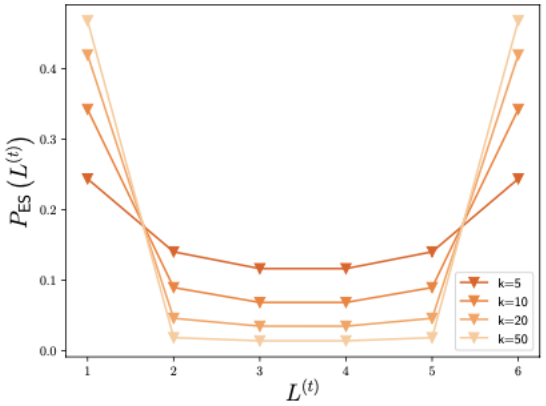


Figure 9: A case of choosing hyper-parameters k of ES to sample 1-6 layer number.

$$P_{\mathrm{LVPS}}(L^{(t)}) = \begin{cases} \frac{b}{(L^{(t)}+k)^2}, & \text{if } L^{(t)} \in [N^{(s)}, L], \\ 0, & \text{otherwise}, \end{cases} \tag{8}$$

$$\text{w.r.t} \int P_{\mathrm{LVPS}}(L^{(t)})\,\mathrm{d}L^{(t)} = 1, \tag{9}$$

where b and c can be solved in terms of Eq. (9) as

$$b = \frac{(N^{(s)}+k)*(L+k)}{L-N^{(s)}}, \quad c = \frac{L+k}{L-N^{(s)}}. \tag{10}$$