Staged Training for Transformer Language Models

Sheng Shen[†] **Pete Walsh**[◇] **Kurt Keutzer**[†] **Jesse Dodge**[◇] **Matthew Peters**[◇] **Iz Beltagy**[◇]
[†] University of California, Berkeley
[◇] Allen Institute for AI
sheng.s@berkeley.edu keutzer@eecs.berkeley.edu
petew,jessed,matthewp,beltagy@allenai.org

Abstract

The current standard approach to scaling transformer language models trains each model size from a different random initialization. As an alternative, we consider a staged training setup that begins with a small model and incrementally increases the amount of compute used for training by applying a "growth operator" to increase the model depth and width. By initializing each stage with the output of the previous one, the training process effectively re-uses the compute from prior stages and becomes more efficient. Our growth operators each take as input the entire training state (including model parameters, optimizer state, learning rate schedule, etc.) and output a new training state from which training continues. We identify two important properties of these growth operators, namely that they preserve both the loss and the "training dynamics" after applying the operator. While the loss-preserving property has been discussed previously, to the best of our knowledge this work is the first to identify the importance of preserving the training dynamics (the rate of decrease of the loss during training). To find the optimal schedule for stages, we use the scaling laws from (Kaplan et al., 2020) to find a precise schedule that gives the most compute saving by starting a new stage when training efficiency starts decreasing. We empirically validate our growth operators and staged training for autoregressive language models, showing up to 22% compute savings compared to a strong baseline trained from scratch. Our code is available at https://github.com/allenai/staged-training.

1 Introduction

Language models form the backbone of many modern NLP systems, and these language models have become progressively larger in recent years. Parameter counts for these models have grown significantly from ELMo (94 M) (Peters et al., 2018) to GPT-3 (175 B) (Brown et al., 2020). While larger models with more learnable parameters perform better on a wide range of tasks, the computational cost to train or even just to evaluate these models has become prohibitively expensive (Schwartz et al., 2020). In this paper, we demonstrate a method to reduce the compute cost of training transformer language models (Vaswani et al., 2017) through a staged training setup that iteratively builds a large model from a smaller one.

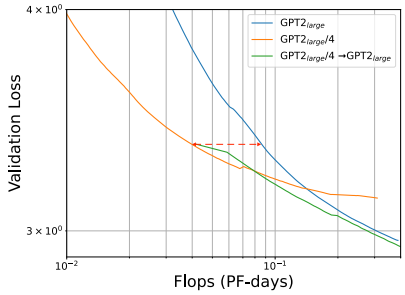


Figure 1: We train a GPT2$_{\text{LARGE}}$ (768M parameters) transformer language model by first training a model 1/4 the size (orange line), then increasing the model size by 4x by applying a growth operator to the entire training state, and restarting training (green line). The result is a large size model with comparable loss to one trained from scratch (blue line) but with reduced compute cost illustrated initially by the dashed red arrow.

Most prior work on scaling language models initializes each model size from a random initialization and trains to convergence (Kaplan et al., 2020). This work illustrated an intriguing property of model training shown in Figure 1, namely that smaller models are initially more compute efficient then larger models, but eventually the larger model will reach a lower loss. Our central idea is to take advantage of this property by first training a smaller model in the compute efficient region, applying a growth operator to the entire training state, and restarting training with a larger sized model. We introduce two operators to perform this growing

Corresponding author: beltagy@allenai.org

operation along the model depth and width dimensions. We also identify two important properties of the operators: first, the loss before growing the model is preserved, and second, the training dynamics of the grown model match that of an equivalent model trained from scratch. To maintain training dynamics, growth operators must take into the entire training state, including the optimizer state and learning rate schedule, in addition to the model weights. As can be seen in Figure 1 our growth operator is loss-preserving, and we show in subsequent sections that it also preserves the training dynamics. These properties also make applying the growth operators conceptually and algorithmically simple as they won't disrupt the training process, and prior results regarding the model size needed to converge to a particular loss still hold.

While prior work (Rusu et al., 2016; Wei et al., 2016; Gong et al., 2019; Liu et al., 2019; Li et al., 2020b; Press et al., 2020; Gu et al., 2021; Li et al., 2021; Rae et al., 2021; Evci et al., 2022) has examined some aspects of staged training, our work is the first to address all aspects including how to grow the entire training state and set the stage schedule. We begin by describing the details of our growth operators for the model weights and optimizer state. We then present a principled way to chose the stage schedule, including how to chose the model sizes and number of gradient for each stage. Intuitively, we should start a new stage when the training efficiency decreases and the rate of loss decrease starts to slow down. To formalize this intuition, we use the scaling laws from (Kaplan et al., 2020) to find the optimal schedule that gives the maximum compute saving. We then show how to approximate the optimal schedule in the realistic scenario without perfect knowledge of the scaling laws. We empirically validate our approach with GPT2 style (Radford et al., 2019) auto-regressive language model models and demonstrate 5-30 % compute savings measured by validation loss, and zero-shot perplexity using two benchmark datasets.

2 Definitions and Properties of Growth Operators

2.1 Definitions

We begin by defining some terms which will be used throughout the paper. We consider a model $\mathbf{y} = \phi(\mathbf{x}, \theta)$ which takes input $\mathbf{x}$, outputs $\mathbf{y}$, with parameters θ. The model is trained by minimizing a loss function $\mathit{loss}(\mathbf{y}, \hat{\mathbf{y}}) \in \mathbb{R}$ through a sequence of parameter updates, obtained by running an optimizer. We will also write $\mathit{loss}(\phi, \mathcal{D})$ for the total loss over a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}$ (or just $\mathit{loss}(\phi)$). The parameter updates for a particular mini-batch are determined by both the mini-batch and the training state, $\mathcal{T} = \{\theta, (m, v), \lambda(t)\}$, including the model parameters θ, optimizer state (m, v, here the first and second moments of the Adam optimizer), and learning rate schedule $\lambda(t)$. Given a training state, we apply a growth operator $\mathbb{G}(\mathcal{T}_{\text{orig}}) = \mathcal{T}_{\text{grow}}$ that takes the original training state and outputs a grown training state where the model size has increased, along with a corresponding compatible optimizer state and learning rate schedule.

2.2 Desired properties

In this section, we define two key properties of growth operators. Building on (Chen et al., 2016) we revisit the *loss-preserving* property, and we introduce a more challenging property, the *training-dynamic-preserving* property.

2.2.1 Preserving Loss

A function-preserving growth operator is one that takes as input an original model and returns a grown model that represents the same function as the original model. If an operator is function-preserving then it is also loss-preserving.

Mathematically, we can formulate loss-preserving as

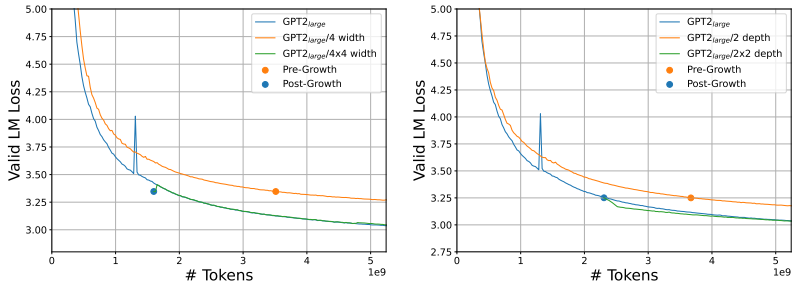
$$\mathit{loss}(\mathbb{G}(\phi)(\mathbf{x}), \mathbf{y}) = \mathit{loss}(\phi(\mathbf{x}), \mathbf{y}) \qquad (1)$$

for any $(\mathbf{x}, \mathbf{y})$. A growth operator that is not loss-preserving wastes time and compute initially until it recovers the same performance of the original model. Figure 2 (and the more detailed Figure 4) show examples of the proposed width and depth growth operators being loss-preserving.

2.2.2 Preserving Training Dynamics

We define the training dynamics as the rate of decrease of loss relative to the amount of compute, and a training-dynamics-preserving growth operator is one that allows the grown model's loss curve to match that of a target model (a model of the same size as the grown model but trained from scratch).

Formally, let ϕ_{k+1} be the resulting model after applying the optimizer update to model ϕ_k with training state $\mathcal{T}_k$. Applying the update requires some amount of compute C_k. The training process



(a) Width growth with $\text{GPT2}_{\text{LARGE}}$ as target (b) Depth growth with $\text{GPT2}_{\text{LARGE}}$ as target

Figure 2: Our growth operators are loss-preserving and training-dynamics preserving. Using (a) as an example, $\text{GPT2}_{\text{LARGE/4}}$ is the original model which is 4x smaller than the target model $\text{GPT2}_{\text{LARGE}}$. The model $\text{GPT2}_{\text{LARGE/4x4}}$ is the grown model resulting from growing $\text{GPT2}_{\text{LARGE/4}}$ by 4x (doubling model width). The PRE-GROWTH point is highlighted on the original model $\text{GPT2}_{\text{LARGE/4}}$, and the POST-GROWTH point is highlighted on the grown model $\text{GPT2}_{\text{LARGE/4x4}}$. The PRE-GROWTH and POST-GROWTH points have the same loss, showing that the width growth operator is loss-preserving. To demonstrate that it is also training dynamics-preserving, we overlay the loss curve for the grown model over the target model and confirm the rate of loss decrease with respect to the number of tokens is the same as the target model trained from scratch. The x-axis is number of training tokens since random initialization, or from the start of the training stage (for $\text{GPT2}_{\text{LARGE/4x4}}$). A similar result is seen in (b) for the depth growth operator.

produces a loss curve that associates the loss with the total amount of compute used for training

$$\mathcal{L}(\phi, C) = \{(\overline{C}_k, loss(\phi_k, \mathcal{D}_k)),\ \ k = 1, 2, \ldots\}$$

where $\overline{C}_k = \sum_{i<=k} C_i$ is the total compute used at step k. The training dynamics is the compute efficiency of training, the expected decrease in the loss relative to the amount of compute:

$$\frac{\partial}{\partial C}\mathcal{L}(\phi, C) = \mathbb{E}_D[\frac{loss(\phi_k, \mathcal{D}) - loss(\phi_{k+1}, \mathcal{D})}{C_k}]$$

which we denote by $\frac{\partial \mathcal{L}}{\partial C}$ as an abuse of notation. Practically it is easy to estimate during training by monitoring the model's loss.

We can now define a training-dynamics-preserving growth operator $\mathbb{G}$ at a point on the loss curve with loss $L_{\mathbb{G}}$ as one that preserves the efficiency of training of the grown model vs. a target model trained from scratch:

$$\frac{\partial}{\partial C}\mathcal{L}(\mathbb{G}(\phi_{orig}), C) = \frac{\partial}{\partial C}\mathcal{L}(\phi_{target}, C) \quad (2)$$

where efficiency is evaluated at $L_{\mathbb{G}}$.

Notice that while loss-preserving is a property comparing the original and grown models, training-dynamics-preserving is a property comparing the grown and target models. This property makes it possible for the grown model to "jump" from the loss curve of the smaller model to the large model and always benefit from faster convergence. A growth operator that does not satisfy this requirement could be creating a larger model but with limited capacity or one that is more difficult to train. Figure 2 (and the more detailed Figure 4) shows examples of the width and depth growth operators preserving the training dynamics, where the grown model perfectly follows the loss curves of the target model.

While the function-preserving property of a growth operator can be confirmed based on the implementation itself, preserving the training dynamics goes beyond just growing the model size; it must be empirically evaluated. To preserve training dynamics, one must address the whole training state including the learning rate and the optimizer state, which are hardly discussed in prior work. We discuss this further in the next section.

3 Growth Operators

We introduce two growth operators below; the operators are described generally, though of course the implementations are model-specific. Our experiments are using the GPT2 transformer architecture (Radford et al., 2019).

3.1 Width

Our width operator doubles the hidden dimension of the entire model, and therefore increases the number of parameters by approximately 4x. This operator applies to all weights in the network in-

cluding embeddings, feed forward layers, bias, and normalization layers, not just the feed forward layers as in prior work (Gu et al., 2021). It grows each layer in slightly different ways. Layer-normalization layers and bias terms with weights $W \in \mathbb{R}^d$ are duplicated:

$$\mathbb{G}(W) = [W, W].$$

where $[\cdot, \cdot]$ represents concatenation and we have overloaded $\mathbb{G}(\cdot)$ to apply to a weight matrix instead of the entire training state. Embedding layers are handled in a similar manner. For the feed forward weights $W \in R^{nxm}$, we design the growth operators as

$$\mathbb{G}(W) = \begin{pmatrix} W & Z \\ Z & W \end{pmatrix}$$

where Z is a zero matrix of size similar to W.

In this case, the input before the grown last feed forward layer that produces the logits is two times wider and the final logits are two times larger. To keep the whole network loss-preserving, we divide the grown weights of the last feed forward layer by a factor of two.

3.2 Depth

Our depth operator doubles the number of layers, and therefore increases the number of non-embedding parameters by 2x. Given a model with layers $(\phi_0, \phi_1, \ldots)$, the depth operator adds an identity layer ϕ_{id} after each layer in the original model, so that the grown model has layers $(\phi_0, \phi_{\text{id}}, \phi_1, \phi_{\text{id}}, \ldots)$. The identity layer is a layer where the input matches the output $\phi_{\text{id}}(\mathbf{x}) = \mathbf{x}$.

To construct the identity layer, we start with the formulation of each layer in GPT2 as two sublayers:

$$\begin{aligned} \mathbf{x}' &= \mathbf{x} + \text{Attention}(\text{LN}(\mathbf{x})), \\ \mathbf{y} &= \mathbf{x}' + \text{FFN}(\text{LN}(\mathbf{x}')) \end{aligned} \tag{3}$$

where $\mathbf{x} \in \mathrm{R}^{\mathrm{d}}$ is the input, $\mathbf{y} \in \mathrm{R}^{\mathrm{d}}$ is the output. LN, Attention, and FFN stands for the layer normalization, multi-head attention and feed-forward operations. We initialize both the scale and bias parameters of each LN in the identity layers to zero, so that $\text{LN}(\mathbf{x}) = \mathbf{0}$. We also set the bias parameters of all linear layers to zero, which combined with the LN initialization gives $\text{Attention}(\text{LN}(\mathbf{x})) = \mathbf{0}$ and $\text{FFN}(\text{LN}(\mathbf{x}')) = \mathbf{0}$, and the entire layer reduces to an identity layer at initialization. Overall, the resulting depth growth operator is loss-preserving.

3.3 Growth operator's impact on training state

In practice, to build a growth operator that preserves training dynamics, we find it important that optimizer state should be grown in a similar way to the model parameters; initial experiments indicated that it can take many training steps to re-estimate the optimizer state, and the initial phase after growth can be unstable. This is expected because training dynamics is the rate of loss change $\frac{\partial \mathcal{L}}{\partial C}$. To match the rate of loss change of the target model, the growth operator needs to reproduce the same scale of model updates, which are controlled by the update rule of the optimizer. Using the ADAM (Kingma and Ba, 2014) optimizer as an example, the update rules are

$$\begin{aligned} m_t &= \beta_1 * m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 * v_{t-1} + (1 - \beta_2) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\lambda(t)}{\sqrt{v_t} + \epsilon} m_t \end{aligned} \tag{4}$$

where g and g^2 are the first-order gradients and the element-wise squared first-order gradients, m is the first moment (average of g), v is the second moment (average of g^2), $\beta_1, \beta_2, \in [0, 1)$ are the exponential decay rates for the moment estimates, ϵ is a small constant, and t is the time-step. For the grown model to be updated at a rate similar to that of the target model, it needs to match its learning rate $\lambda(t)$ which we discuss in the next section. It also needs to produce an optimizer state m, v that's compatible with the gradients of the grown model, $g(\mathbb{G}(\phi))$ and $g^2(\mathbb{G}(\phi))$. Given that m, v are averages of g, g^2, we argue that m, v should be grown with growth operators $\mathbb{G}_m$, $\mathbb{G}_v$ that satisfy the following properties:

$$g(\mathbb{G}_m(\phi)) = \mathbb{G}_m(g(\phi)) \tag{5}$$

$$g^2(\mathbb{G}_v(\phi)) = \mathbb{G}_v(g^2(\phi)) \tag{6}$$

The first condition states that the "gradients of the grown model" should match "growing the gradients of the original model". The second condition is similar but for the squared gradients. To satisfy Eq. 5 and 6, the implementations of $\mathbb{G}_m$, $\mathbb{G}_v$ are slightly different from $\mathbb{G}$. For the width growth operator, some of the weights need to be scaled

Abusing the notation; Eq. 5, 6 are using g, g^2 as functions to compute gradients of a model, not the gradients themselves.

by 0.5x or 0.25x to account for the 2x scaling in the forward pass (see Section 3.1). For the depth growth operator, we copy m and v for the original model layers and set m and v to zero for the identity layers.

Along with the loss-preserving property, the training dynamics preserving property ensures that the new optimizer state is compatible with the grown model weights.

Learning rate To match training dynamics, the learning rate schedule of the grown model must match that of the target model. The intuition is that our growth operators allow the model state to "jump" from the loss curve of the original model $\mathcal{L}(\phi_{orig}, C)$ to $\mathcal{L}(\phi_{target}, C)$ at a point with loss $L_{\mathbb{G}}$. Because our growth operator is loss preserving, the loss $L_{\mathbb{G}}$ defines two points on the loss curves, PRE-GROWTH on $\mathcal{L}(\phi_{orig}, C)$ and GROWTH-TARGET on $\mathcal{L}(\phi_{target}, C)$. To match the training dynamics of $\mathcal{L}(\phi_{target}, C)$, we start training the grown model with a learning rate schedule that matches the target model but starts from GROWTH-TARGET. We discuss finding that matched GROWTH-TARGET point in Section 4 and 5.

4 Optimal Schedule

Prior work (Gu et al., 2021; Gong et al., 2019) used heuristics to determine the training schedule. In contrast, our goal is to find the optimal training schedule. An optimal training schedule is one that, given a target model size, specifies the optimal sequence of growth operators, intermediate model sizes, and number of training steps in each stage leading to the most compute saving. This section will explain our intuition behind our optimal schedule, then explains how to mathematically find it.

Training to OPTIMALITY We start from the scaling laws (Kaplan et al., 2020), which showed that the training of transformer language models is initially efficient with fast loss reduction, then the compute-efficient regime ends and the rate of the loss reduction slows down. In addition, the initial compute-efficient regime is longer for larger models. These ideas are illustrated in Figure 1where $\frac{\partial \mathcal{L}}{\partial C}$ is initially large then it slows down. As shown in Kaplan et al. (2020) and Li et al. (2020b), the optimal compute allocation should favor a large model size and stop the training by the end of the initial compute-efficient regime when $\frac{\partial \mathcal{L}}{\partial C} = \tau_{opt}$, where τ_{opt} is some threshold. We call this training to "Optimality" as opposed to training to "Completion" or to convergence. We discuss later this section how to find the point of OPTIMALITY using constrained optimization in an idealistic scenario, then later in Section 5 using a more practical method that estimates τ_{opt}.

Intermediate Stages We next discuss where in training to grow a model. Intuitively, the optimal schedule is one where the original small model is trained until its compute-efficient regime ends, then grown to a larger model to continue its compute-efficient regime. Figure 1 highlights one such potential schedule. Notice that there's a specific point on the loss curve of the original model that leads to the most compute saving; growing the model earlier means a wasted opportunity skipping some of the fast loss reduction stage of the original model, and growing the model later means wasting compute by continuing to train after the loss of the original model begins to plateau.

Schedule which minimizes compute Next, we describe how to mathematically find this optimal schedule. For that, we use the scaling laws (Kaplan et al., 2020) which derived empirical fits for the language model loss L as it relates to compute C, number of non-embedding parameters N, number of gradient update steps S, and batch size B. The total compute and the loss are given by

$$C \approx 6NBS,$$
$$L(N, S) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{S_c}{S}\right)^{\alpha_S} \quad (7)$$

where $\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*$ are all model-specific constants. Thus, finding the the optimal schedule can be formulated as a constrained optimization problem. The output is the intermediate model sizes, and the amount of compute for each stage. We discuss the details in Appendix C.

5 Practical Schedule

While the general scaling laws are known, re-estimating their constants ($\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*$) for our setup is challenging because it requires running a large number of models of different sizes. Instead of estimating all the constants, we make the observation that we only need to find three key points:

- PRE-GROWTH $\in \mathcal{L}(\phi_{orig}, C)$ at which we grow the original model

- GROWTH-TARGET $\in \mathcal{L}(\phi_{target}, C)$ that the model is grown towards to
- OPTIMALITY $\in \mathcal{L}(\phi_{grown}, C)$ at which we stop training the grown model.

Next we discuss the mathematical definition of each point, how to find them, and how to use them in the actual training procedure.

PRE-GROWTH and OPTIMALITY points We define PRE-GROWTH and OPTIMALITY using the slope of the loss curve, as they depend on the rate of change of the loss. Formally,

$$\begin{aligned} \text{PRE-GROWTH}: \frac{\partial}{\partial C}\mathcal{L}(\phi_{orig}, C) &= \tau_{\mathbb{G}} \\ \text{OPTIMALITY}: \frac{\partial}{\partial C}\mathcal{L}(\phi_{grown}, C) &= \tau_{opt} \end{aligned} \tag{8}$$

where $\tau_{\mathbb{G}}$ and τ_{opt} are empirically estimated thresholds.

Importantly, both thresholds are independent of the model size, and this independence can be derived from Eq. 7. We also empirically confirmed the model-size independence by training many models of different sizes; reconfirming results from Kaplan et al. (2020), the shape of the loss curves for different sized models were similar, just shifted and scaled. Additionally, while both thresholds are model-size independent, $\tau_{\mathbb{G}}$ is a function of the growth operator.

GROWTH-TARGET point The importance of the GROWTH-TARGET point is that it specifics the learning rate schedule of the grown model (Section 3.3). We will specify GROWTH-TARGET using the number of training steps $S_{\text{GROWTH-TARGET}}$. We found that it can be simply defined using

$$S_{\text{GROWTH-TARGET}} = \rho \times S_{\text{PRE-GROWTH}} \tag{9}$$

where ρ is an empirically estimated constant, and $S_{\text{PRE-GROWTH}}$ is number of training steps at PRE-GROWTH.

As above, and knowing that the loss curves of models of different sized models are scaled and shifted versions of each other, we can use Eq. 7 to show that ρ is independent of the model size, and it is only a function of the growth operator. We also verified this empirically using different model sizes.

Estimating τ and ρ Given that equations 8 and 9 are independent of model sizes, it is enough to estimate the values of τ and ρ using small models. To estimate them, we first identify the three necessary points. Specifically, for a single growth operator, we train an original model and a target model from scratch then follow the intuition discussed in Section 4 to choose a PRE-GROWTH point (on $\mathcal{L}(\phi_{orig}, C)$) and an OPTIMALITY point (on $\mathcal{L}(\phi_{target}, C)$). The GROWTH-TARGET point is simply the point on $\mathcal{L}(\phi_{target}, C)$) with the same loss at PRE-GROWTH. A plot like Figure 1 (and the more detailed Figure 5) make it easy to manually identify the three points, and we leave it to future work to automatically find these points.

Notice that this method required training a target model from scratch, but in practice we can estimate the constants once for smaller model sizes and apply them to larger sizes. Using the identified points, we use equations 8 and 9 to estimate values of $\tau_{\mathbb{G}}$ and ρ for each growth operator, and the value of τ_{opt}. The empirical values we estimated are in Appendix D. Notice that estimating these constants is much simpler than estimating all the constants of the scaling laws ($\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*$) making this procedure simpler to apply to a new setup.

Algorithm 1 Staged training for transformer LMs

Input:
ϕ: original model
$\lambda(t)$: learning rate schedule
M: number of stages
$(\mathbb{G}_i, \tau_{\mathbb{G},i}, \rho_{\mathbb{G},i})$: (growth op, τ, ρ) for stage i
$\mathbb{G}_1$: first stage operator assumed to be identity operator
τ_{opt}: last stage's τ

Begin:
$t \leftarrow 0$ // number of training steps
for $i = 1$ **to** M **do**
  if $i = M$ **then**
    $\tau \leftarrow \tau_{opt}$ // last stage, stop at optimality
  else
    $\tau \leftarrow \tau_{\mathbb{G},i}$
  end if
  while $\frac{\partial}{\partial C}\mathcal{L}(\phi, C) \leq \tau$ **do**
    Run training step and update ϕ
    $t \leftarrow t + 1$ // update learning rate using $\lambda(t)$
  end while
  $\phi \leftarrow \mathbb{G}(\phi)$
  $t \leftarrow t \times \rho_{\mathbb{G},i}$ // set learning rate for next stage
end for
return ϕ

Training Procedure Algorithm 1 summarizes the staged training procedure. Notice that it extends it to the M stage case. The algorithm starts with the model ϕ, trains it from scratch until PRE-GROWTH, grows the model, sets number of steps for the grown model, and repeats until no more stages, then continues training to OPTIMALITY.

6 Experiments

In this section we present our main empirical results, focusing on the amount of compute saving. We show results on in-domain data (validation loss) and in the zero-shot transfer setting. We also compare our work to prior work in growing transformer language models, establishing that previous methods fall short in one or more areas.

6.1 Experimental setting

We experiment with GPT2 from (Radford et al., 2019) (in base and large sizes) using the public C4 (Raffel et al., 2020) dataset. We follow the learning rate schedule in Kaplan et al. (2020) for all model sizes, where the warmup period is set to 3,000 steps, the batch size is set to 512 and the sequence length is 1,024. For the zero-shot transfer learning, we experiment on two tasks: Wikitext-103 (Merity et al., 2017) and LAMBADA (Paperno et al., 2016), similar to (Li et al., 2021). We report the compute saving for in-domain validation loss and zero-shot performance. We compare our practical schedule in Section 5 with the manual schedule that directly matches the loss of the PRE-GROWTH model with the target model to select the GROWTH-TARGET point. We choose and report different thresholds to decide the OPTIMALITY of the training in each stage.

6.2 Main results

Figure 1 shows the compute saving for growing $\text{GPT2}_{\text{LARGE/4}}$ original model to a target model of $\text{GPT2}_{\text{LARGE}}$ (we show the results of other combinations of growth operators and the results of growing to $\text{GPT2}_{\text{BASE}}$ in Appendix A in Figure 5). It is clear that our grown models are reaching the same loss as the target models but with less compute. It is important to note that as both models train longer, the amount of compute saving drops. This illustrates a key design in our schedule where we stop training of the grown model at OPTIMALITY when the compute-efficient regime ends. Figure 1 also demonstrates that small models achieve better loss trade-off when the total compute is limited, but saturate at higher loss when more compute is added, highlighting the advantages of our schedule in applying the growth operator to them before convergence.

		$\text{GPT2}_{\text{LARGE}}$ 5k	10k	14k	$\text{GPT2}_{\text{BASE}}$ 3.75k	7k	11k
Baseline	(**loss**)	3.21	3.03	2.97	3.61	3.45	3.38
Compute savings (percent saved vs. baseline)							
1 stage $_{\text{manual}}$	2xW	19.3	5.6	4.0	23.8	14.5	13.8
	2xD	33.5	7.8	6.3	24.0	23.6	21.8
1 stage $_{\text{practical}}$	2xW	22.5	7.3	5.2	24.3	20.2	19.7
	4xW	18.0	5.3	3.8	16.0	8.6	5.5
	2xD	37.0	11.0	6.1	24.7	20.4	19.8
	4xD	22.5	7.3	5.2	18.8	10.1	6.4
	2xDxW	28.8	5.4	3.8	19.0	9.5	6.83
2 stage $_{\text{practical}}$	2x2xW	26.8	10.9	7.8	26.8	17.9	11.4
	2x2xD	30.0	14.5	10.4	33.3	21.4	15.9

Table 1: Percentage compute savings for $\text{GPT2}_{\text{LARGE}}$ and $\text{GPT2}_{\text{BASE}}$ on in-domain validation loss on C4, "W" is width and "D" is depth growth operators. Percent savings is how much less compute our approach takes than the baseline to to train a model to equal or better loss than the baseline. Significant savings can be found using our both width and depth growth operators, and two growth stages can lead to even more savings than one growth stage. The derivative threshold at 5k steps is -0.1, at 10k steps it's -0.05, and at 14k steps it's -0.04; thus, 5k is undertrained, 10k is approximately at the optimality threshold of -0.052, and 14k is trained beyond optimality. Similar for $\text{GPT2}_{\text{BASE}}$.

Table 1 shows the amount of compute saving of our growth operators for two different model sizes. The table shows that our grown model reaches the same performance of the baseline (target model) trained from scratch while having considerable compute saving ranging from 30% to 5% for different thresholds. The first row in Table 1 denotes the number of steps we used to train the baseline model. Given that our growth operator is loss-preserving and training-dynamic preserving, we can always reach the same loss of the target model with less compute and the compute saving becomes larger when we decide to stop the target model earlier. Also, given the same growth ratio (4x growth), the depth growth operator is preferable versus width concerning the compute saving.

We also evaluate our pretrained models on other language modeling tasks in the zero-shot setting to verify that the grown models maintain their transfer learning capabilities. Table 2 shows results on Wikitext-103 and LAMBADA. It can be seen that using our practical schedule, we achieve comparable and sometimes better performance versus using

		Zero-shot Wikitext-103 (PPL)						Zero-shot LAMBADA (accuracy)					
		$GPT2_{LARGE}$			$GPT2_{BASE}$			$GPT2_{LARGE}$			$GPT2_{BASE}$		
		5k	10k	14k	3.75k	7k	11k	5k	10k	14k	3.75k	7k	11k
Baseline		41.0	32.3	30.3	68.5	57.1	50.0	39.6	43.2	44.7	31.1	33.0	34.7
Compute savings (percent saved vs. baseline, negative means more compute than baseline)													
1 stage $_{manual}$	2xwidth	-18.5	6.2	5.8	-19.7	0.3	2.4	3.2	6.7	8.3	-8.3	0.2	13.6
	2xdepth	28.5	11.8	12.0	24.0	28.0	17.3	33.5	16.8	13.8	24.0	22.9	18.2
1 stage $_{practical}$	2xwidth	-20.5	-0.25	8.7	-15.7	5.9	3.8	-15.5	12.3	14.8	-15.7	3.3	10.6
	4xwidth	-13.4	1.3	7.4	-12.0	1.1	6.4	-11.0	18.0	18.2	-12.0	10.1	8.6
	2xdepth	32.0	13.5	11.4	31.3	33.5	17.5	32.0	23.5	21.4	24.7	16.8	13.9
	4xdepth	21.0	17.5	9.5	14.6	7.9	3.1	21.0	20.5	14.6	14.6	9.4	7.8
	2xdepthxwidth	-10.5	-0.3	1.6	-12.0	3.6	4.5	-95.0	-37.5	0.7	5.4	6.4	6.4
2 stage $_{practical}$	2x2xwidth	-2.5	6.3	9.3	3.3	5.4	8.0	-3.3	13.4	20.3	-3.3	1.8	11.8
	2x2xdepth	30.0	12.5	12.8	33.3	14.3	11.8	19.0	14.5	23.6	19.9	10.6	15.0

Table 2: Percentage compute savings for $GPT2_{LARGE}$ and $GPT2_{BASE}$ on out-of-domain Wikitext-103 (perplexity) and LAMBADA (accuracy). Percent savings is how much less compute our approach takes than the baseline to to train a model to equal or better perplexity or accuracy than the baseline; negative numbers mean our approach took more compute than the baseline to achieve equal or better performance. Results are mixed in the early stages of training, but our approach leads to compute savings for all experiments later in training (14k for large, 11k for base). The derivative threshold at 5k steps is -0.1, at 10k steps it's -0.05, and at 14k steps it's -0.04; thus, 5k is undertrained, 10k is approximately at the optimality threshold of -0.052, and 14k is trained beyond optimality. Similar for $GPT2_{BASE}$.

the manual schedule for both in-domain loss and zero-shot transfer learning. In some cases we have negative compute saving with the width operator or when combing the width and depth operator on zero-shot transfer learning tasks early in training (indicating the grown model used more compute to get to the same (or better) performance), but the compute saving is always positive when training for longer. We assume that this is due to instabilities in optimization right after applying the growth operator. When the training proceeds, the zero-shot performance will shortly recover and the grown models will have better positive compute saving at OPTIMALITY for the two model sizes. It is also less of an issue for the `base` model size as GROWTH-TARGET. Moreover, the depth growth operator leads to better performance compute saving trade-off compared to the width operator under the same growth ratio. Finally, we show that applying the growth operator twice (in two stages) lead to the best-performing compute saving and performance. See Figure 7 for detailed evaluation plots.

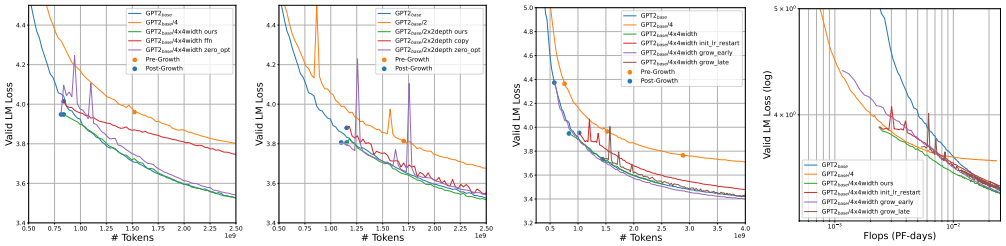
6.3 Ablations and prior work comparison

We experiment with prior work and conduct ablation studies for our method. We show that prior works fall short in one or more of the key components of our proposed method; preserving loss, preserving training dynamics, or following an optimal schedule.

Prior work comparison. We evaluate against two growth operators from previous work (Gu et al., 2021; Gong et al., 2019); one uses weight sharing to make a feed-forward network module wider, and the other makes an entire network deeper. Given the nature of these growth operators, the grown models do not represent the same function as the original model; as shown in Figure 3, neither retain the same loss as the original model, and thus they do not satisfy our loss-preserving property.

Prior work also mostly ignored the optimizer state. In Figure 3, we explore this as an ablation by simply setting the optimizer state to zero for our width and depth growth operators. Though the loss can be retained at the starting pointing for the grown model, the training dynamic becomes extreme unstable after applying the growth operators, and thus such a growth operator will not have the training-dynamics-preserving property.

Ablation studies We further perform two ablation studies concerning the learning rate schedule and growth schedule in Figure 3. For the learning rate schedule, we compare our setting of the learning rate to the GROWTH-TARGET point with restarting the learning rate schedule as in Rae et al. (2021). It can be seen that resetting the learning rate schedule leads to much unstable training dynamics that are not aligned with the GROWTH-TARGET. For



(a) Width op. and optimizer(b) Depth op. and optimizer (c) LR and optimal schedule(d) LR and optimal schedule

Figure 3: Comparing with three different baselines from prior work and ablation studies. • The width growth operator of Gu et al. (2021) in $\text{GPT2}_{\text{BASE/4x4width ffn}}$ and the depth growth operator of Gong et al. (2019) in $\text{GPT2}_{\text{BASE/2x2depth copy}}$ are not loss preserving (higher initial loss). Also, $\text{GPT2}_{\text{BASE/4x4width ffn}}$ is significantly underperforming the target model $\text{GPT2}_{\text{BASE}}$. • Resetting the optimizer state to zero instead of growing it (the `zero_opt` runs) have large instabilities and not preserving of the training dynamics. • 3c shows that restarting the learning rate schedule as in Rae et al. (2021) is not training-dynamics-preserving. • 3c, 3d also show that not following our optimal schedule and grow the model too early or too late is still loss-preserving and training-dynamics-preserving but leads to lower compute saving

the growth schedule, we experiment with growing the small model earlier or later than the GROWTH-TARGET point computed with our proposed practical schedule. It shows that growing the model too late/early is still loss-preserving and training-dynamics-preserving but the grown models lose the compute saving advantages.

Training to completion vs. to OPTIMALITY. While training to OPTIMALITY is not one of our contributions, it is an important part of our training schedule. Here we compare our models in Table 2 with equivalent models trained to completion. In Radford et al. (2019) , $\text{GPT2}_{\text{BASE}}$ trained to completion achieves 37.5 PPL on Wikitext-103 and 45.9 accuracy on LAMBADA. Our best-performing 2x2xdepth grown $\text{GPT2}_{\text{LARGE}}$ model achieves the same PPL on Wikitext-103 as this model with only 15% of the compute and the same accuracy as the this model on LAMBADA with 33% of the compute. Notice that we are comparing different model sizes, a large model trained to OPTIMALITY vs. a smaller model trained to completion.

7 Related Work

Perhaps the most similar prior work is (Gu et al., 2021). However, this work did not provide a method to decide when to apply a growth operator and instead evaluated the performance of their operators at 100/300/500/700K steps of a small model, or applied a heuristic to equally distribute training steps among different model sizes. They did not discuss the optimizer state, and they reset the learning rate to the maximum value of 1e-4 at the beginning of each state without warmup. Their work built on progressive stacking (discussed below), and their proposed method to grow the width only grew the feed-forward layers instead of the entire model width.

(Gong et al., 2019) proposed "progressive stacking" which doubles the depth of a BERT transformer model by copying layers; to construct a $2L$-layer model from a L layer model it copies layer $i \leq L$ in the smaller model to layer $(i + L)$ in the larger model. The optimizer state is reset at the beginning of each stage, but the learning rate is kept the same as the prior stage. They use heuristics to set the stacking schedule: 50K steps for 3-layer, 70K steps for 6-layer model, 280K steps for 12-layer model. In an ablation study they examined the sensitivity to the number of steps before applying their growth operator and concluded that there is a threshold number of steps and for switching times such that switching to the larger model before the threshold led to compute savings, but switching after the threshold didn't. This is consistent with our results that showed the amount of compute saving is closely related to the stage length.

(Li et al., 2020a) proposed growing encoder-decoder transformers in the context of training a machine translation system. Their depth growth operator is identical to progressive stacking, although they explore operations that increase the model by only copying some of the layers from the small to large model (e.g. growing from 12 to 18 layers). They do not mention optimizer state, and reset the

learning rate to max value at each stage.

In contemporaneous work, Gopher (Rae et al., 2021) introduced a method which tiles the weights from a small model to a larger one. However, their growth operator does not satisfy our two properties from Section 2, and they focus on training their models to completion. Evci et al. (2022) proposes a way to initialize the grown weights by maximizing the gradient norm of the new weights for vision models. Their growth operator also requires specified activation functions that can not be directly applied to transformers. Finally, Li et al. (2021) proposed applying a curriculum learning strategy to the sequence length to reduce the training cost when training large language models. Their work is orthogonal to our method and our methods could be combined; we leave this to future work.

8 Conclusion and future work

One direction of future work is to combine batch size warmup (Brown et al., 2020) and sequence length growth (Li et al., 2021) with our depth and width growth operators. Another applies our proposed methods to train a massive transformer.

We presented a staged training method for large transformer-based language models that grows the model size during training. We demonstrated the importance of two properties of the growth operators (loss-preserving and training-dynamics-preserving), and provided depth and width operators that satisfy both requirements. Finally, we devised a principled approach to find the optimal schedule and a simple method to apply the schedule in practice. Empirical evaluations show up to 22% compute saving.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher 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 *Neural Information Processing Systems (NeurIPS)*.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. 2016. Net2net: Accelerating learning via knowledge transfer. In *International Conference on Learning Representations*.

Utku Evci, Max Vladymyrov, Thomas Unterthiner, Bart van Merriënboer, and Fabian Pedregosa. 2022. Gradmax: Growing neural networks using gradient information. *arXiv preprint arXiv:2201.05125*.

Linyuan Gong, Di He, Zhuohan Li, Tao Qin, Liwei Wang, and Tieyan Liu. 2019. Efficient training of bert by progressively stacking. In *International Conference on Machine Learning*, pages 2337–2346. PMLR.

Xiaotao Gu, Liyuan Liu, Hongkun Yu, Jing Li, Chen Chen, and Jiawei Han. 2021. On the transformer growth for progressive bert training. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5174–5180.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Bei Li, Ziyang Wang, Hui Liu, Yufan Jiang, Quan Du, Tong Xiao, Huizhen Wang, and Jingbo Zhu. 2020a. Shallow-to-deep training for neural machine translation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 995–1005, Online. Association for Computational Linguistics.

Conglong Li, Minjia Zhang, and Yuxiong He. 2021. Curriculum learning: A regularization method for efficient and stable billion-scale gpt model pretraining. *arXiv preprint arXiv:2108.06084*.

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. 2020b. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning*, pages 5958–5968. PMLR.

Qiang Liu, Lemeng Wu, and Dilin Wang. 2019. Splitting steepest descent for growing neural architectures. *Advances in neural information processing systems*.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer sentinel mixture models. In *International Conference on Learning Representations*.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. 2016. The lambada dataset: Word prediction requiring a broad discourse context. In *ACL*.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proc. of NAACL*.

Ofir Press, Noah A Smith, and Mike Lewis. 2020. Shortformer: Better language modeling using shorter inputs. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. 2021. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research (JMLR)*, 21(140).

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Communications of the ACM (CACM)*, 63(12):54–63.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000–6010.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. 2016. Network morphism. In *International Conference on Machine Learning*, pages 564–572. PMLR.

A Additional Plots and Results

In Figure 4 and 5, we show loss curve plot and compute plot for applying growth operator to $\text{GPT2}_{\text{BASE}}$ model. These suggest our growth operator is loss-preserving, training dynamic preserving and saves compute to train the target BASE size model.

In Figure 6, we demonstrate the loss curve and derivatives regarding the validation loss and compute in log scale. It clearly shows the used practical threshold gives us a good estimate of the OPTIMALITY of the loss curve.

In Figure 7, we present the evaluation results for every checkpoints across the steps for target and grown $\text{GPT2}_{\text{BASE}}$ model. The performance of the $\text{GPT2}_{\text{BASE/2x2depth}}$ can always perform on par with or better than the baseline while $\text{GPT2}_{\text{BASE/4x4width}}$ underperforms baseline in the initial phase after growing but can catch up quickly. This also explains the potential negative compute we have in the main text in the early phase of the evaluation and better compute saving at OPTIMALITY.

B Model Architecture

We use $\text{GPT2}_{\text{BASE}}$ and $\text{GPT2}_{\text{LARGE}}$ models. For $\text{GPT2}_{\text{BASE}}$ model, it consists of 12 layers, 768 hidden dimensions, 12 heads and 125M parameters. For $\text{GPT2}_{\text{LARGE}}$ model, it consists of 24 layers, 1536 hidden dimensions, 16 heads and 760M parameters.

C Optimal Stage Schedule

This appendix defines a constrained optimization problem that produces the optimal staged-training schedule. The scaling laws of Kaplan et al. (2020) derived empirical fits for the language model loss L as it relates to the total amount of compute C, number of non-embedding parameters N, number of gradient update steps S, and batch size B. The total compute is given by

$$C \approx 6NBS, \tag{10}$$

and the loss L for any model size N and number of steps S is given by:

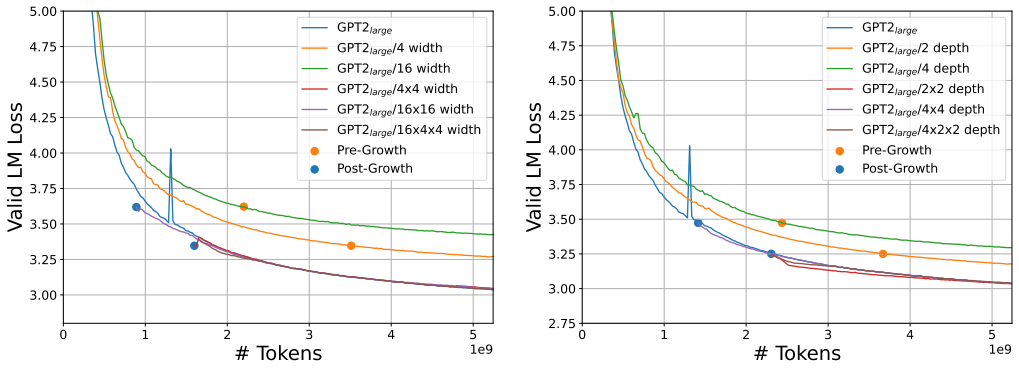
$$L(N, S) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{S_c}{S}\right)^{\alpha_S} \tag{11}$$

when training at the critical batch size $B_{crit} = \frac{B_*}{L^{1/\alpha_B}}$, and where $\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*$ are all model-specific constants.

Our goal is to minimize the total compute to train a model of a given size N_{target} to a given target loss L_{target}. We assume we have access to perfect growth operators that are loss-preserving and training dynamics-preserving, and that can grow from any model size to any other size. We consider a training regime consisting of a number of

This neglects contributions proportional to the context length, n_{ctx}, and may not be valid in regime of large n_{ctx} where $n_{ctx} \geq 12 d_{model}$

We can restrict the model size increases by adding additional constraints.



(a) Width growth with GPT2$_{\text{LARGE}}$ as target

(b) Depth growth with GPT2$_{\text{LARGE}}$ as target

Figure 4: Similar to Figure 2, our width and depth growth operators are loss-preserving and training dynamics preserving. • GPT2$_{\text{LARGE/16x16-WIDTH}}$ indicates starting from a 16x smaller model then growing it 16x by doubling the width twice. • GPT2$_{\text{LARGE/16x4x4-WIDTH}}$ indicates growing the model 16x over two stages, by doubling the width once, continue training the model, then doubling the width again. • The same applies to the depth growth operator.

stages. In each stage k we train a model with size N_k for S_k gradient steps, with the goal of reaching size N_{target} and achieving a target loss L_{target} at the end of the final stage. We assume that $N_k \geq N_{k-1}$, and that there exists some way to initialize the model with size N_k from one of size N_{k-1} without changing the loss. For simplicity, we neglect the batch size contribution to compute, and assume training is always at the critical batch size $B_*/L_{target}^{1/\alpha_B}$.

With these assumptions the total compute at the end of training for M stages is:

$$C = \sum_{k=1}^{M} 6N_k \frac{B_*}{L_{target}^{1/\alpha_B}} S_k \tag{12}$$

We can compute the loss at the end of each stage in an iterative fashion. The loss at the end of the first stage is given by $L_1(N_1, S_1)$ from Eqn. 11. Then for each subsequent stage, we assume the loss curves can be translated and the loss at the end of stage k is computed by starting with the loss at the end of the prior stage and decreased for S_k steps. To do so, first compute the effective number of steps $S_{eff,k}$ needed to reach initial loss for the stage L_{k-1} with model size N_k, and then compute the loss at the end of the stage by $L_k(N_k, S_{eff,k} + S_k)$. In summary:

$$\begin{aligned} L_1 &= \left(\frac{N_c}{N_1}\right)^{\alpha_N} + \left(\frac{S_c}{S_1}\right)^{\alpha_S} \\ L_k &= \left(\frac{N_c}{N_k}\right)^{\alpha_N} + \left(\frac{S_c}{S_{eff,k} + S_k}\right)^{\alpha_S}, k > 1 \\ S_{eff,k} &= \frac{S_c}{(L_{k-1} - (N_c/N_k)^{\alpha_N})^{1/\alpha_S}}. \end{aligned}$$

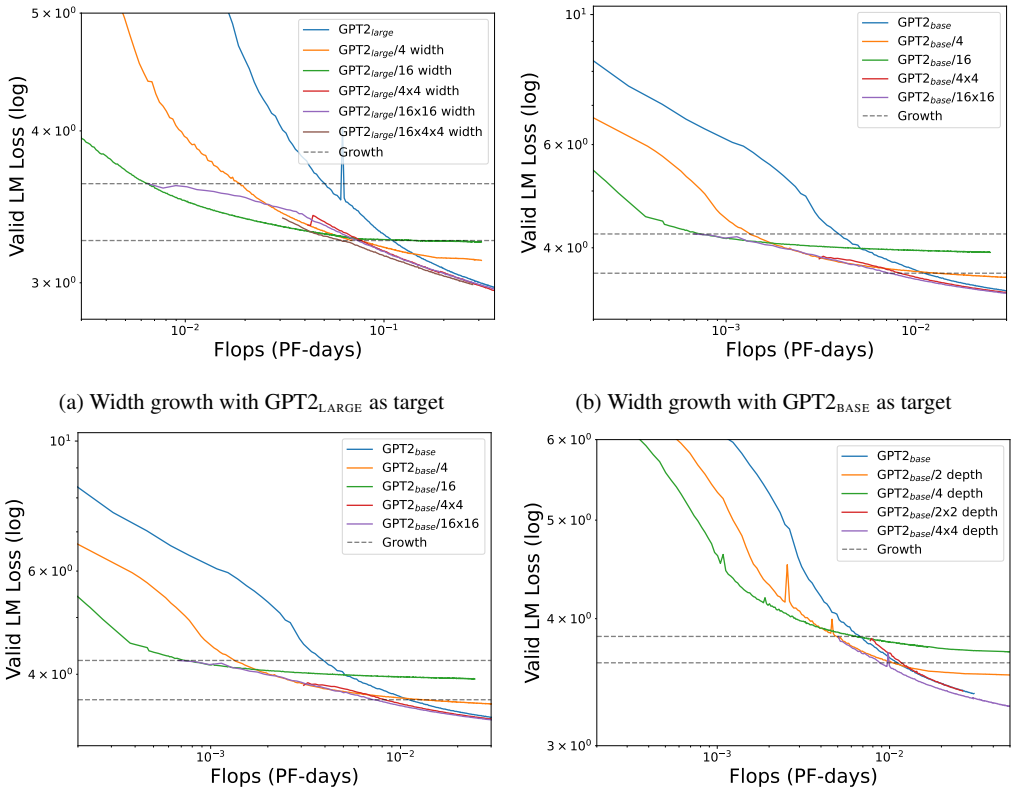
With this in hand, we can use a constrained optimizer to solve for the optimal schedule by minimizing Eqn. 12, subject to the constraint that the final model size is the target size, and loss at the end of training is the target loss. Formally,

$$\min_{\{(N_k, S_k)\}} \sum_{k=1}^{M} 6N_k \frac{B_*}{L_{target}^{1/\alpha_B}} S_k$$

subject to

$$\begin{aligned} L_M &= L_{target} \\ N_M &= N_{target} \\ 0 &< N_1 \\ N_{k-1} &\leq N_k, k > 1 \\ 0 &\leq S_k \end{aligned}$$

Note that in the single stage case ($M = 1$) with no N_{target} condition, this formulation reduces to the optimal calculation in Appendix C of Kaplan et al. (2020) to find the optimal model size to reach a target loss. This matches our training to OPTIMALITY.



(a) Width growth with GPT2$_{\text{LARGE}}$ as target

(b) Width growth with GPT2$_{\text{BASE}}$ as target

(c) Depth growth with GPT2$_{\text{LARGE}}$ as target

(d) Depth growth with GPT2$_{\text{BASE}}$ as target

Figure 5: Growth operator and total compute. Our grown models are saving compute compared with target model.

Number of stages	Compute reduction factor
1	1.0
2	0.83
3	0.792
4	0.779
5	0.771
10	0.763

Table 3: Decrease in compute costs over an optimal single stage training regime.

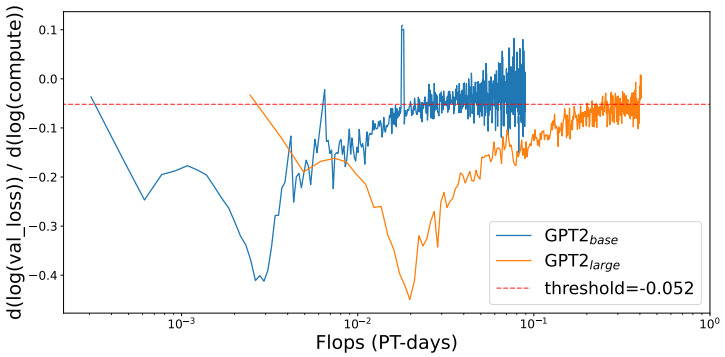
Stage	N_k	S_k	L_k	C_k
1	2.7M	15.4K	4.09	0.0033
2	22.1M	24.3K	3.58	0.0154
3	71.9M	30.8K	3.31	0.0365
4	163M	36.0K	3.13	0.0665
5	306M	40.5K	3.00	0.1056

Table 4: Sample optimal schedule for a five stage regime to train to $L_{target} = 3$ showing the number of non-embedding parameters N_k, the number of gradient steps S_k, the loss at the end of the stage L_k, and the compute in the stage C_k.

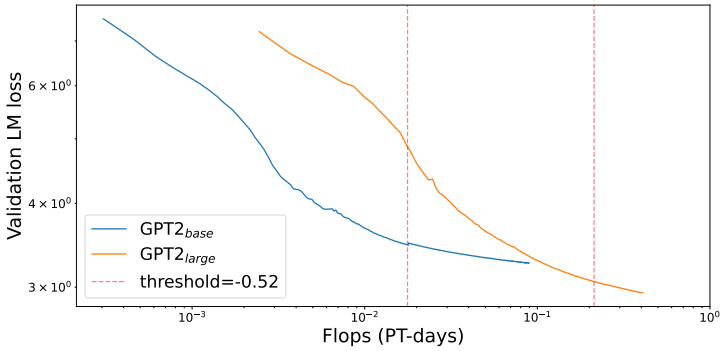
Measuring compute saving To measure the compute saving from staged training to reach a certain L_{target}, we use a single staged training ($M = 1$) with no N_{target} condition, and use our optimization algorithm to find the optimal compute and model size; this becomes N_{target}. Next, we run the optimization problem with N_{target}, L_{target}, and $M > 1$ to get the optimal training schedule and the expected compute, which we compare with $M = 1$ to find the expected compute saving.

Observations An implementation of this optimization shows that the increase in compute efficiency for using multiple stages is independent of the target loss L_{target}, and quickly approaches 0.76 as the number of stages increases (using the scaling parameters for autoregressive transformer language models).

We also found that constraining the ratio between consecutive model sizes ($\frac{N_k}{N_{k-1}}$) to be 2, 4 or 8, leads to almost the same compute savings. These constraints come from the practical constraints of our implementation of the growth operators.



(a) First derivative on the validation loss curve for GPT2$_{\text{BASE}}$ and GPT2$_{\text{LARGE}}$. This is an approximation to the training dynamics $\frac{\partial \mathcal{L}}{\partial C}$.



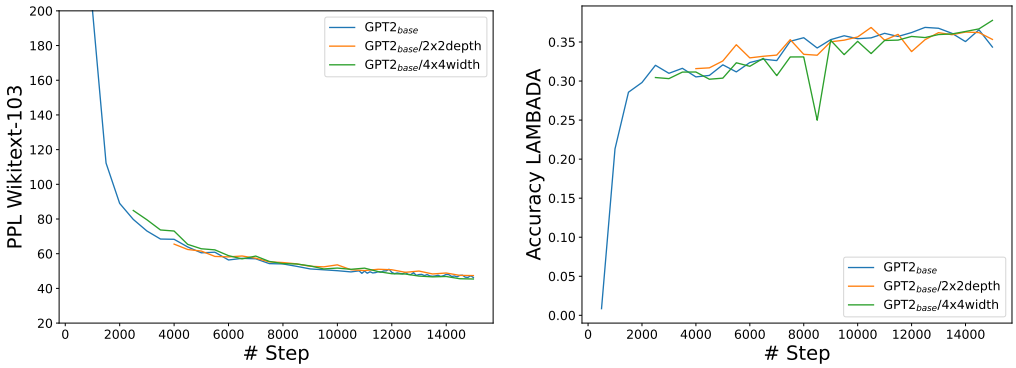
(b) Validation loss curve for GPT2$_{\text{BASE}}$ and GPT2$_{\text{LARGE}}$

Figure 6: 6a shows the first derivative of the validation loss curve for GPT2$_{\text{BASE}}$ and GPT2$_{\text{LARGE}}$. The horizontal dotted lines shows the empirical value of the derivative threshold we use to identify the point of OPTIMALITY. 6b shows how the threshold value in 6a maps to a point in the loss curve for both model sizes (the first vertical line for GPT2$_{\text{BASE}}$ and the second for GPT2$_{\text{LARGE}}$). Both point show slowed training and the end of the compute-efficient regime.

D Practical Stage Schedule

The empirical values we estimated for the constants are:

$$
\begin{aligned}
\tau_{opt} &= -0.052 \\
\tau_{depth} &= -0.0575 \\
\tau_{width} &= -0.0475 \\
\tau_{depth-width} &= -0.03 \\
\rho_{depth} &= 0.70 \\
\rho_{width} &= 0.55 \\
\rho_{depth-width} &= 0.40
\end{aligned}
$$



(a) Wikitext-103 evaluation with $\text{GPT2}_{\text{BASE}}$ as target

(b) LAMBADA evaluation with GPT2BASE as target.

Figure 7: The detailed evaluation plots on Wikitext-103 and LAMBADA for $\text{GPT2}_{\text{BASE}}$ trained from scratch and two grown model using depth or width operator. The performance of the $\text{GPT2}_{\text{BASE/2x2depth}}$ perfectly follows or even outperform the target model, while $\text{GPT2}_{\text{BASE/4x4width}}$ underperforms the target model initially after growing but it catches up quickly. This explains the initial "negative" compute saving in Table 2 for the width growth operator, followed by positive compute saving as the model trains longer.

Staged Training for Transformer Language Models

Sheng Shen[†] **Pete Walsh**[*] **Kurt Keutzer**[†] **Jesse Dodge**[*] **Matthew Peters**[*] **Iz Beltagy**[*1]

[†] University of California, Berkeley

[*] Allen Institute for AI

Abstract

The current standard approach to scaling transformer language models trains each model size from a different random initialization. As an alternative, we consider a staged training setup that begins with a small model and incrementally increases the amount of compute used for training by applying a "growth operator" to increase the model depth and width. By initializing each stage with the output of the previous one, the training process effectively re-uses the compute from prior stages and becomes more efficient. Our growth operators each take as input the entire training state (including model parameters, optimizer state, learning rate schedule, etc.) and output a new training state from which training continues. We identify two important properties of these growth operators, namely that they preserve both the loss and the "training dynamics" after applying the operator. While the loss-preserving property has been discussed previously, to the best of our knowledge this work is the first to identify the importance of preserving the training dynamics (the rate of decrease of the loss during training). To find the optimal schedule for stages, we use the scaling laws from (**?**) to find a precise schedule that gives the most compute saving by starting a new stage when training efficiency starts decreasing. We empirically validate our growth operators and staged training for autoregressive language models, showing up to 22% compute savings compared to a strong baseline trained from scratch. Our code is available at `https://github.com/allenai/staged-training`.

1. Introduction

Language models form the backbone of many modern NLP systems, and these language models have become progressively larger in recent years. Parameter counts for these models have grown significantly from ELMo (94 M) (**?**) to GPT-3 (175 B) (**?**). While larger models with more learnable parameters perform better on a wide range of tasks, the computational cost to train or even just to evaluate these models has become prohibitively expensive (**?**). In this paper, we demonstrate a method to reduce the compute cost of training transformer language models (**?**) through a staged training setup that iteratively builds a large model from a smaller one.

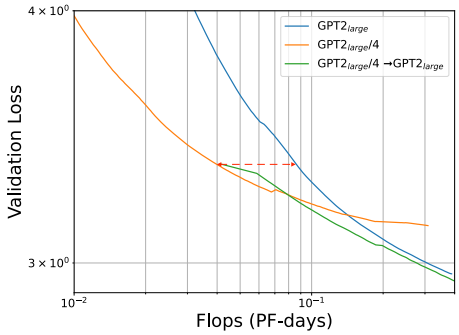


Figure 1: We train a GPT2$_{\text{LARGE}}$ (768M parameters) transformer language model by first training a model 1/4 the size (orange line), then increasing the model size by 4x by applying a growth operator to the entire training state, and restarting training (green line). The result is a large size model with comparable loss to one trained from scratch (blue line) but with reduced compute cost illustrated initially by the dashed red arrow.

Most prior work on scaling language models initializes each model size from a random initialization and trains to convergence (**?**). This work illustrated an intriguing property of model training shown in Figure 1, namely that smaller models are initially more compute efficient then larger models, but eventually the larger model will reach a lower loss. Our central idea is to take advantage of this property by first training a smaller model in the compute efficient region, applying a growth operator to the entire training state, and restarting training with a larger sized model. We introduce two operators to perform this grow-

[1]Correspondence to: beltagy@allenai.org

ing operation along the model depth and width dimensions. We also identify two important properties of the operators: first, the loss before growing the model is preserved, and second, the training dynamics of the grown model match that of an equivalent model trained from scratch. To maintain training dynamics, growth operators must take into the entire training state, including the optimizer state and learning rate schedule, in addition to the model weights. As can be seen in Figure 1 our growth operator is loss-preserving, and we show in subsequent sections that it also preserves the training dynamics. These properties also make applying the growth operators conceptually and algorithmically simple as they won't disrupt the training process, and prior results regarding the model size needed to converge to a particular loss still hold.

While prior work (**??????????**) has examined some aspects of staged training, our work is the first to address all aspects including how to grow the entire training state and set the stage schedule. We begin by describing the details of our growth operators for the model weights and optimizer state. We then present a principled way to chose the stage schedule, including how to chose the model sizes and number of gradient for each stage. Intuitively, we should start a new stage when the training efficiency decreases and the rate of loss decrease starts to slow down. To formalize this intuition, we use the scaling laws from (**?**) to find the optimal schedule that gives the maximum compute saving. We then show how to approximate the optimal schedule in the realistic scenario without perfect knowledge of the scaling laws. We empirically validate our approach with GPT2 style (**?**) auto-regressive language model models and demonstrate 5-30 % compute savings measured by validation loss, and zero-shot perplexity using two benchmark datasets.

2. Definitions and Properties of Growth Operators

2.1. Definitions

We begin by defining some terms which will be used throughout the paper. We consider a model $\mathbf{y} = \phi(\mathbf{x}, \theta)$ which takes input $\mathbf{x}$, outputs $\mathbf{y}$, with parameters θ. The model is trained by minimizing a loss function $\mathit{loss}(\mathbf{y}, \hat{\mathbf{y}}) \in \mathbb{R}$ through a sequence of parameter updates, obtained by running an optimizer. We will also write $\mathit{loss}(\phi, \mathcal{D})$ for the total loss over a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}$ (or just $\mathit{loss}(\phi)$). The parameter updates for a particular mini-batch are determined by both the mini-batch and the training state, $\mathcal{T} = \{\theta, (m, v), \lambda(t)\}$, including the model parameters θ, optimizer state $(m, v$, here the first and second moments of the Adam optimizer), and learning rate schedule $\lambda(t)$. Given a training state, we apply a growth operator $\mathbb{G}(\mathcal{T}_{\text{orig}}) = \mathcal{T}_{\text{grow}}$ that takes the original training state and outputs a grown training state where the model size has increased, along with a corresponding compatible optimizer state and learning rate schedule.

2.2. Desired properties

In this section, we define two key properties of growth operators. Building on (**?**) we revisit the *loss-preserving* property, and we introduce a more challenging property, the *training-dynamic-preserving* property.

2.2.1. Preserving Loss

A function-preserving growth operator is one that takes as input an original model and returns a grown model that represents the same function as the original model. If an operator is function-preserving then it is also loss-preserving.

Mathematically, we can formulate loss-preserving as

$$\mathit{loss}(\mathbb{G}(\phi)(\mathbf{x}), \mathbf{y}) = \mathit{loss}(\phi(\mathbf{x}), \mathbf{y}) \tag{1}$$

for any $(\mathbf{x}, \mathbf{y})$. A growth operator that is not loss-preserving wastes time and compute initially until it recovers the same performance of the original model. Figure 2 (and the more detailed Figure 4) show examples of the proposed width and depth growth operators being loss-preserving.

2.2.2. Preserving Training Dynamics

We define the training dynamics as the rate of decrease of loss relative to the amount of compute, and a training-dynamics-preserving growth operator is one that allows the grown model's loss curve to match that of a target model (a model of the same size as the grown model but trained from scratch).

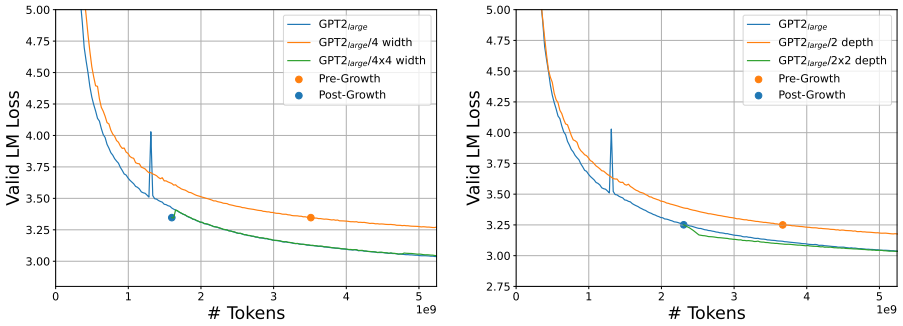
Formally, let ϕ_{k+1} be the resulting model after applying the optimizer update to model ϕ_k with training state $\mathcal{T}_k$. Applying the update requires some amount of compute C_k. The training process produces a loss curve that associates the loss with the total amount of compute used for training

$$\mathcal{L}(\phi, C) = \{(\overline{C}_k, \mathit{loss}(\phi_k, \mathcal{D}_k)),\ \ k = 1, 2, \ldots\}$$

where $\overline{C}_k = \sum_{i<=k} C_i$ is the total compute used at step k. The training dynamics is the compute efficiency of training, the expected decrease in the loss relative to the amount of compute:

$$\frac{\partial}{\partial C}\mathcal{L}(\phi, C) = \mathbb{E}_D[\frac{\mathit{loss}(\phi_k, \mathcal{D}) - \mathit{loss}(\phi_{k+1}, \mathcal{D})}{C_k}]$$

which we denote by $\frac{\partial \mathcal{L}}{\partial C}$ as an abuse of notation. Practically it is easy to estimate during training by monitoring the model's loss.



(a) Width growth with $\text{GPT2}_{\text{LARGE}}$ as target (b) Depth growth with $\text{GPT2}_{\text{LARGE}}$ as target

Figure 2: Our growth operators are loss-preserving and training-dynamics preserving. Using (a) as an example, $\text{GPT2}_{\text{LARGE/4}}$ is the original model which is 4x smaller than the target model $\text{GPT2}_{\text{LARGE}}$. The model $\text{GPT2}_{\text{LARGE/4x4}}$ is the grown model resulting from growing $\text{GPT2}_{\text{LARGE/4}}$ by 4x (doubling model width). The PRE-GROWTH point is highlighted on the original model $\text{GPT2}_{\text{LARGE/4}}$, and the POST-GROWTH point is highlighted on the grown model $\text{GPT2}_{\text{LARGE/4x4}}$. The PRE-GROWTH and POST-GROWTH points have the same loss, showing that the width growth operator is loss-preserving. To demonstrate that it is also training dynamics-preserving, we overlay the loss curve for the grown model over the target model and confirm the rate of loss decrease with respect to the number of tokens is the same as the target model trained from scratch. The x-axis is number of training tokens since random initialization, or from the start of the training stage (for $\text{GPT2}_{\text{LARGE/4x4}}$). A similar result is seen in (b) for the depth growth operator.

We can now define a training-dynamics-preserving growth operator $\mathbb{G}$ at a point on the loss curve with loss $L_{\mathbb{G}}$ as one that preserves the efficiency of training of the grown model vs. a target model trained from scratch:

$$\frac{\partial}{\partial C}\mathcal{L}(\mathbb{G}(\phi_{orig}), C) = \frac{\partial}{\partial C}\mathcal{L}(\phi_{target}, C) \tag{2}$$

where efficiency is evaluated at $L_{\mathbb{G}}$.

Notice that while loss-preserving is a property comparing the original and grown models, training-dynamics-preserving is a property comparing the grown and target models. This property makes it possible for the grown model to "jump" from the loss curve of the smaller model to the large model and always benefit from faster convergence. A growth operator that does not satisfy this requirement could be creating a larger model but with limited capacity or one that is more difficult to train. Figure 2 (and the more detailed Figure 4) shows examples of the width and depth growth operators preserving the training dynamics, where the grown model perfectly follows the loss curves of the target model.

While the function-preserving property of a growth operator can be confirmed based on the implementation itself, preserving the training dynamics goes beyond just growing the model size; it must be empirically evaluated. To preserve training dynamics, one must address the whole training state including the learning rate and the optimizer state, which are hardly discussed in prior work. We discuss this further in the next section.

3. Growth Operators

We introduce two growth operators below; the operators are described generally, though of course the implementations are model-specific. Our experiments are using the GPT2 transformer architecture (**?**).

3.1. Width

Our width operator doubles the hidden dimension of the entire model, and therefore increases the number of parameters by approximately 4x. This operator applies to all weights in the network including embeddings, feed forward layers, bias, and normalization layers, not just the feed forward layers as in prior work (**?**). It grows each layer in slightly different ways. Layer-normalization layers and bias terms with weights $W \in \mathbb{R}^d$ are duplicated:

$$\mathbb{G}(W) = [W, W].$$

where $[\cdot, \cdot]$ represents concatenation and we have overloaded $\mathbb{G}(\cdot)$ to apply to a weight matrix instead of the entire training state. Embedding layers are handled in a similar manner. For the feed forward weights $W \in R^{nxm}$, we design the growth operators as

$$\mathbb{G}(W) = \begin{pmatrix} W & Z \\ Z & W \end{pmatrix}$$

where Z is a zero matrix of size similar to W.

In this case, the input before the grown last feed forward layer that produces the logits is two times wider and the

final logits are two times larger. To keep the whole network loss-preserving, we divide the grown weights of the last feed forward layer by a factor of two.

3.2. Depth

Our depth operator doubles the number of layers, and therefore increases the number of non-embedding parameters by 2x. Given a model with layers $(\phi_0, \phi_1, \ldots)$, the depth operator adds an identity layer ϕ_{id} after each layer in the original model, so that the grown model has layers $(\phi_0, \phi_{\text{id}}, \phi_1, \phi_{\text{id}}, \ldots)$. The identity layer is a layer where the input matches the output $\phi_{\text{id}}(\mathbf{x}) = \mathbf{x}$.

To construct the identity layer, we start with the formulation of each layer in GPT2 as two sublayers:

$$\begin{aligned} \mathbf{x}' &= \mathbf{x} + \text{Attention}(\text{LN}(\mathbf{x})), \\ \mathbf{y} &= \mathbf{x}' + \text{FFN}(\text{LN}(\mathbf{x}')) \end{aligned} \tag{3}$$

where $\mathbf{x} \in \mathrm{R}^{\mathrm{d}}$ is the input, $\mathbf{y} \in \mathrm{R}^{\mathrm{d}}$ is the output. LN, Attention, and FFN stands for the layer normalization, multi-head attention and feed-forward operations. We initialize both the scale and bias parameters of each LN in the identity layers to zero, so that $\text{LN}(\mathbf{x}) = \mathbf{0}$. We also set the bias parameters of all linear layers to zero, which combined with the LN initialization gives $\text{Attention}(\text{LN}(\mathbf{x})) = \mathbf{0}$ and $\text{FFN}(\text{LN}(\mathbf{x}')) = \mathbf{0}$, and the entire layer reduces to an identity layer at initialization. Overall, the resulting depth growth operator is loss-preserving.

3.3. Growth operator's impact on training state

In practice, to build a growth operator that preserves training dynamics, we find it important that optimizer state should be grown in a similar way to the model parameters; initial experiments indicated that it can take many training steps to re-estimate the optimizer state, and the initial phase after growth can be unstable. This is expected because training dynamics is the rate of loss change $\frac{\partial \mathcal{L}}{\partial C}$. To match the rate of loss change of the target model, the growth operator needs to reproduce the same scale of model updates, which are controlled by the update rule of the optimizer. Using the ADAM (**?**) optimizer as an example, the update rules are

$$\begin{aligned} m_t &= \beta_1 * m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 * v_{t-1} + (1-\beta_2) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\lambda(t)}{\sqrt{v_t} + \epsilon} m_t \end{aligned} \tag{4}$$

where g and g^2 are the first-order gradients and the element-wise squared first-order gradients, m is the first moment (average of g), v is the second moment (average of g^2), $\beta_1, \beta_2, \in [0, 1)$ are the exponential decay rates for the moment estimates, ϵ is a small constant, and t is the time-step. For the grown model to be updated at a rate similar to that of the target model, it needs to match its learning rate $\lambda(t)$ which we discuss in the next section. It also needs to produce an optimizer state m, v that's compatible with the gradients of the grown model, $g(\mathbb{G}(\phi))$ and $g^2(\mathbb{G}(\phi))$. Given that m, v are averages of g, g^2, we argue that m, v should be grown with growth operators $\mathbb{G}_m$, $\mathbb{G}_v$ that satisfy the following properties:[2]

$$g(\mathbb{G}_m(\phi)) = \mathbb{G}_m(g(\phi)) \tag{5}$$

$$g^2(\mathbb{G}_v(\phi)) = \mathbb{G}_v(g^2(\phi)) \tag{6}$$

The first condition states that the "gradients of the grown model" should match "growing the gradients of the original model". The second condition is similar but for the squared gradients. To satisfy Eq. 5 and 6, the implementations of $\mathbb{G}_m, \mathbb{G}_v$ are slightly different from $\mathbb{G}$. For the width growth operator, some of the weights need to be scaled by 0.5x or 0.25x to account for the 2x scaling in the forward pass (see Section 3.1). For the depth growth operator, we copy m and v for the original model layers and set m and v to zero for the identity layers.

Along with the loss-preserving property, the training dynamics preserving property ensures that the new optimizer state is compatible with the grown model weights.

Learning rate To match training dynamics, the learning rate schedule of the grown model must match that of the target model. The intuition is that our growth operators allow the model state to "jump" from the loss curve of the original model $\mathcal{L}(\phi_{orig}, C)$ to $\mathcal{L}(\phi_{target}, C)$ at a point with loss $L_{\mathbb{G}}$. Because our growth operator is loss preserving, the loss $L_{\mathbb{G}}$ defines two points on the loss curves, PRE-GROWTH on $\mathcal{L}(\phi_{orig}, C)$ and GROWTH-TARGET on $\mathcal{L}(\phi_{target}, C)$. To match the training dynamics of $\mathcal{L}(\phi_{target}, C)$, we start training the grown model with a learning rate schedule that matches the target model but starts from GROWTH-TARGET. We discuss finding that matched GROWTH-TARGET point in Section 4 and 5.

4. Optimal Schedule

Prior work (**??**) used heuristics to determine the training schedule. In contrast, our goal is to find the optimal training schedule. An optimal training schedule is one that, given a target model size, specifies the optimal sequence of growth operators, intermediate model sizes, and number of training steps in each stage leading to the most compute saving. This section will explain our intuition behind our optimal schedule, then explains how to mathematically find it.

[2] Abusing the notation; Eq. 5, 6 are using g, g^2 as functions to compute gradients of a model, not the gradients themselves.

Training to OPTIMALITY We start from the scaling laws (**?**), which showed that the training of transformer language models is initially efficient with fast loss reduction, then the compute-efficient regime ends and the rate of the loss reduction slows down. In addition, the initial compute-efficient regime is longer for larger models. These ideas are illustrated in Figure 1where $\frac{\partial \mathcal{L}}{\partial C}$ is initially large then it slows down. As shown in **?** and **?**, the optimal compute allocation should favor a large model size and stop the training by the end of the initial compute-efficient regime when $\frac{\partial \mathcal{L}}{\partial C} = \tau_{opt}$, where τ_{opt} is some threshold. We call this training to "Optimality" as opposed to training to "Completion" or to convergence. We discuss later this section how to find the point of OPTIMALITY using constrained optimization in an idealistic scenario, then later in Section 5 using a more practical method that estimates τ_{opt}.

Intermediate Stages We next discuss where in training to grow a model. Intuitively, the optimal schedule is one where the original small model is trained until its compute-efficient regime ends, then grown to a larger model to continue its compute-efficient regime. Figure 1 highlights one such potential schedule. Notice that there's a specific point on the loss curve of the original model that leads to the most compute saving; growing the model earlier means a wasted opportunity skipping some of the fast loss reduction stage of the original model, and growing the model later means wasting compute by continuing to train after the loss of the original model begins to plateau.

Schedule which minimizes compute Next, we describe how to mathematically find this optimal schedule. For that, we use the scaling laws (**?**) which derived empirical fits for the language model loss L as it relates to compute C, number of non-embedding parameters N, number of gradient update steps S, and batch size B. The total compute and the loss are given by

$$\begin{aligned} C &\approx 6NBS, \\ L(N,S) &= \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{S_c}{S}\right)^{\alpha_S} \end{aligned} \tag{7}$$

where $\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*$ are all model-specific constants. Thus, finding the the optimal schedule can be formulated as a constrained optimization problem. The output is the intermediate model sizes, and the amount of compute for each stage. We discuss the details in Appendix C.

5. Practical Schedule

While the general scaling laws are known, re-estimating their constants ($\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*$) for our setup is challenging because it requires running a large number of models of different sizes. Instead of estimating all the constants, we make the observation that we only need to find three key points:

- PRE-GROWTH $\in \mathcal{L}(\phi_{orig}, C)$ at which we grow the original model
- GROWTH-TARGET $\in \mathcal{L}(\phi_{target}, C)$ that the model is grown towards to
- OPTIMALITY $\in \mathcal{L}(\phi_{grown}, C)$ at which we stop training the grown model.

Next we discuss the mathematical definition of each point, how to find them, and how to use them in the actual training procedure.

PRE-GROWTH and OPTIMALITY points We define PRE-GROWTH and OPTIMALITY using the slope of the loss curve, as they depend on the rate of change of the loss. Formally,

$$\begin{aligned} \text{PRE-GROWTH} &: \frac{\partial}{\partial C}\mathcal{L}(\phi_{orig}, C) = \tau_{\mathbb{G}} \\ \text{OPTIMALITY} &: \frac{\partial}{\partial C}\mathcal{L}(\phi_{grown}, C) = \tau_{opt} \end{aligned} \tag{8}$$

where $\tau_{\mathbb{G}}$ and τ_{opt} are empirically estimated thresholds.

Importantly, both thresholds are independent of the model size, and this independence can be derived from Eq. 7. We also empirically confirmed the model-size independence by training many models of different sizes; reconfirming results from **?**, the shape of the loss curves for different sized models were similar, just shifted and scaled. Additionally, while both thresholds are model-size independent, $\tau_{\mathbb{G}}$ is a function of the growth operator.

GROWTH-TARGET point The importance of the GROWTH-TARGET point is that it specifics the learning rate schedule of the grown model (Section 3.3). We will specify GROWTH-TARGET using the number of training steps $S_{\text{GROWTH-TARGET}}$. We found that it can be simply defined using

$$S_{\text{GROWTH-TARGET}} = \rho \times S_{\text{PRE-GROWTH}} \tag{9}$$

where ρ is an empirically estimated constant, and $S_{\text{PRE-GROWTH}}$ is number of training steps at PRE-GROWTH.

As above, and knowing that the loss curves of models of different sized models are scaled and shifted versions of each other, we can use Eq. 7 to show that ρ is independent of the model size, and it is only a function of the growth operator. We also verified this empirically using different model sizes.

Estimating τ and ρ Given that equations 8 and 9 are independent of model sizes, it is enough to estimate the values of τ and ρ using small models. To estimate them,

Algorithm 1 Staged training for transformer LMs

Input:
ϕ: original model
$\lambda(t)$: learning rate schedule
M: number of stages
$(\mathbb{G}_i, \tau_{\mathbb{G},i}, \rho_{\mathbb{G},i})$: (growth op, τ, ρ) for stage i
$\mathbb{G}_1$: first stage operator assumed to be identity operator
τ_{opt}: last stage's τ
Begin:
$t \leftarrow 0$ // number of training steps
for $i = 1$ **to** M **do**
  if $i = M$ **then**
    $\tau \leftarrow \tau_{opt}$ // last stage, stop at optimality
  else
    $\tau \leftarrow \tau_{\mathbb{G},i}$
  end if
  while $\frac{\partial}{\partial C}\mathcal{L}(\phi, C) \leq \tau$ **do**
    Run training step and update ϕ
    $t \leftarrow t + 1$ // update learning rate using $\lambda(t)$
  end while
  $\phi \leftarrow \mathbb{G}(\phi)$
  $t \leftarrow t \times \rho_{\mathbb{G},i}$ // set learning rate for next stage
end for
return ϕ

we first identify the three necessary points. Specifically, for a single growth operator, we train an original model and a target model from scratch then follow the intuition discussed in Section 4 to choose a PRE-GROWTH point (on $\mathcal{L}(\phi_{orig}, C)$) and an OPTIMALITY point (on $\mathcal{L}(\phi_{target}, C)$). The GROWTH-TARGET point is simply the point on $\mathcal{L}(\phi_{target}, C))$ with the same loss at PRE-GROWTH. A plot like Figure 1 (and the more detailed Figure 5) make it easy to manually identify the three points, and we leave it to future work to automatically find these points.

Notice that this method required training a target model from scratch, but in practice we can estimate the constants once for smaller model sizes and apply them to larger sizes. Using the identified points, we use equations 8 and 9 to estimate values of $\tau_{\mathbb{G}}$ and ρ for each growth operator, and the value of τ_{opt}. The empirical values we estimated are in Appendix D. Notice that estimating these constants is much simpler than estimating all the constants of the scaling laws $(\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*)$ making this procedure simpler to apply to a new setup.

Training Procedure Algorithm 1 summarizes the staged training procedure. Notice that it extends it to the M stage case. The algorithm starts with the model ϕ, trains it from scratch until PRE-GROWTH, grows the model, sets number of steps for the grown model, and repeats until no more stages, then continues training to OPTIMALITY.

6. Experiments

In this section we present our main empirical results, focusing on the amount of compute saving. We show results on in-domain data (validation loss) and in the zero-shot transfer setting. We also compare our work to prior work in growing transformer language models, establishing that previous methods fall short in one or more areas.

6.1. Experimental setting

We experiment with GPT2 from (**?**) (in base and large sizes) using the public C4 (**?**) dataset. We follow the learning rate schedule in **?** for all model sizes, where the warmup period is set to 3,000 steps, the batch size is set to 512 and the sequence length is 1,024. For the zero-shot transfer learning, we experiment on two tasks: Wikitext-103 (**?**) and LAMBADA (**?**), similar to (**?**). We report the compute saving for in-domain validation loss and zero-shot performance. We compare our practical schedule in Section 5 with the manual schedule that directly matches the loss of the PRE-GROWTH model with the target model to select the GROWTH-TARGET point. We choose and report different thresholds to decide the OPTIMALITY of the training in each stage.

6.2. Main results

Figure 1 shows the compute saving for growing $\text{GPT2}_{\text{LARGE/4}}$ original model to a target model of $\text{GPT2}_{\text{LARGE}}$ (we show the results of other combinations of growth operators and the results of growing to $\text{GPT2}_{\text{BASE}}$ in Appendix A in Figure 5). It is clear that our grown models are reaching the same loss as the target models but with less compute. It is important to note that as both models train longer, the amount of compute saving drops. This illustrates a key design in our schedule where we stop training of the grown model at OPTIMALITY when the compute-efficient regime ends. Figure 1 also demonstrates that small models achieve better loss trade-off when the total compute is limited, but saturate at higher loss when more compute is added, highlighting the advantages of our schedule in applying the growth operator to them before convergence.

Table 1 shows the amount of compute saving of our growth operators for two different model sizes. The table shows that our grown model reaches the same performance of the baseline (target model) trained from scratch while having considerable compute saving ranging from 30% to 5% for different thresholds. The first row in Table 1 denotes the number of steps we used to train the baseline model. Given that our growth operator is loss-preserving and training-dynamic preserving, we can always reach the same loss of

		$GPT2_{LARGE}$			$GPT2_{BASE}$		
		5k	10k	14k	3.75k	7k	11k
Baseline	(**loss**)	3.21	3.03	2.97	3.61	3.45	3.38
Compute savings (percent saved vs. baseline)							
1 stage $_{manual}$	2xW	19.3	5.6	4.0	23.8	14.5	13.8
	2xD	33.5	7.8	6.3	24.0	23.6	21.8
1 stage $_{practical}$	2xW	22.5	7.3	5.2	24.3	20.2	19.7
	4xW	18.0	5.3	3.8	16.0	8.6	5.5
	2xD	37.0	11.0	6.1	24.7	20.4	19.8
	4xD	22.5	7.3	5.2	18.8	10.1	6.4
	2xDxW	28.8	5.4	3.8	19.0	9.5	6.83
2 stage $_{practical}$	2x2xW	26.8	10.9	7.8	26.8	17.9	11.4
	2x2xD	30.0	14.5	10.4	33.3	21.4	15.9

Table 1: Percentage compute savings for $GPT2_{LARGE}$ and $GPT2_{BASE}$ on in-domain validation loss on C4, "W" is width and "D" is depth growth operators. Percent savings is how much less compute our approach takes than the baseline to to train a model to equal or better loss than the baseline. Significant savings can be found using our both width and depth growth operators, and two growth stages can lead to even more savings than one growth stage. The derivative threshold at 5k steps is -0.1, at 10k steps it's -0.05, and at 14k steps it's -0.04; thus, 5k is undertrained, 10k is approximately at the optimality threshold of -0.052, and 14k is trained beyond optimality. Similar for $GPT2_{BASE}$.

the target model with less compute and the compute saving becomes larger when we decide to stop the target model earlier. Also, given the same growth ratio (4x growth), the depth growth operator is preferable versus width concerning the compute saving.

We also evaluate our pretrained models on other language modeling tasks in the zero-shot setting to verify that the grown models maintain their transfer learning capabilities. Table 2 shows results on Wikitext-103 and LAMBADA. It can be seen that using our practical schedule, we achieve comparable and sometimes better performance versus using the manual schedule for both in-domain loss and zero-shot transfer learning. In some cases we have negative compute saving with the width operator or when combing the width and depth operator on zero-shot transfer learning tasks early in training (indicating the grown model used more compute to get to the same (or better) performance), but the compute saving is always positive when training for longer. We assume that this is due to instabilities in optimization right after applying the growth operator. When the training proceeds, the zero-shot performance will shortly recover and the grown models will have better positive compute saving at OPTIMALITY for the two model sizes. It is also less of an issue for the `base` model size as GROWTH-TARGET. Moreover, the depth growth operator leads to better performance compute saving trade-off compared to the width operator under the same growth ratio. Finally, we show that applying the growth operator twice (in two stages) lead to the best-performing compute saving and performance. See Figure 7 for detailed evaluation plots.

6.3. Ablations and prior work comparison

We experiment with prior work and conduct ablation studies for our method. We show that prior works fall short in one or more of the key components of our proposed method; preserving loss, preserving training dynamics, or following an optimal schedule.

Prior work comparison. We evaluate against two growth operators from previous work (**??**); one uses weight sharing to make a feed-forward network module wider, and the other makes an entire network deeper. Given the nature of these growth operators, the grown models do not represent the same function as the original model; as shown in Figure 3, neither retain the same loss as the original model, and thus they do not satisfy our loss-preserving property.

Prior work also mostly ignored the optimizer state. In Figure 3, we explore this as an ablation by simply setting the optimizer state to zero for our width and depth growth operators. Though the loss can be retained at the starting pointing for the grown model, the training dynamic becomes extreme unstable after applying the growth operators, and thus such a growth operator will not have the training-dynamics-preserving property.

Ablation studies We further perform two ablation studies concerning the learning rate schedule and growth schedule in Figure 3. For the learning rate schedule, we compare our setting of the learning rate to the GROWTH-TARGET point with restarting the learning rate schedule as in **?**. It can be seen that resetting the learning rate schedule leads to much unstable training dynamics that are not aligned with the GROWTH-TARGET. For the growth schedule, we experiment with growing the small model earlier or later than the GROWTH-TARGET point computed with our proposed practical schedule. It shows that growing the model too late/early is still loss-preserving and training-dynamics-preserving but the grown models lose the compute saving advantages.

Training to completion vs. to OPTIMALITY. While training to OPTIMALITY is not one of our contributions, it is an important part of our training schedule. Here we compare our models in Table 2 with equivalent models trained to completion. In **?** , $GPT2_{BASE}$ trained to completion achieves 37.5 PPL on Wikitext-103 and 45.9 accuracy on LAMBADA. Our best-performing 2x2xdepth grown $GPT2_{LARGE}$ model achieves the same PPL on Wikitext-103 as this model with only 15% of the compute and the same

		Zero-shot Wikitext-103 (PPL)						Zero-shot LAMBADA (accuracy)					
		$GPT2_{LARGE}$			$GPT2_{BASE}$			$GPT2_{LARGE}$			$GPT2_{BASE}$		
		5k	10k	14k	3.75k	7k	11k	5k	10k	14k	3.75k	7k	11k
Baseline		41.0	32.3	30.3	68.5	57.1	50.0	39.6	43.2	44.7	31.1	33.0	34.7
Compute savings (percent saved vs. baseline, negative means more compute than baseline)													
1 stage $_{manual}$	2xwidth	-18.5	6.2	5.8	-19.7	0.3	2.4	3.2	6.7	8.3	-8.3	0.2	13.6
	2xdepth	28.5	11.8	12.0	24.0	28.0	17.3	33.5	16.8	13.8	24.0	22.9	18.2
1 stage $_{practical}$	2xwidth	-20.5	-0.25	8.7	-15.7	5.9	3.8	-15.5	12.3	14.8	-15.7	3.3	10.6
	4xwidth	-13.4	1.3	7.4	-12.0	1.1	6.4	-11.0	18.0	18.2	-12.0	10.1	8.6
	2xdepth	32.0	13.5	11.4	31.3	33.5	17.5	32.0	23.5	21.4	24.7	16.8	13.9
	4xdepth	21.0	17.5	9.5	14.6	7.9	3.1	21.0	20.5	14.6	14.6	9.4	7.8
	2xdepthxwidth	-10.5	-0.3	1.6	-12.0	3.6	4.5	-95.0	-37.5	0.7	5.4	6.4	6.4
2 stage $_{practical}$	2x2xwidth	-2.5	6.3	9.3	3.3	5.4	8.0	-3.3	13.4	20.3	-3.3	1.8	11.8
	2x2xdepth	30.0	12.5	12.8	33.3	14.3	11.8	19.0	14.5	23.6	19.9	10.6	15.0

Table 2: Percentage compute savings for $GPT2_{LARGE}$ and $GPT2_{BASE}$ on out-of-domain Wikitext-103 (perplexity) and LAMBADA (accuracy). Percent savings is how much less compute our approach takes than the baseline to to train a model to equal or better perplexity or accuracy than the baseline; negative numbers mean our approach took more compute than the baseline to achieve equal or better performance. Results are mixed in the early stages of training, but our approach leads to compute savings for all experiments later in training (14k for large, 11k for base). The derivative threshold at 5k steps is -0.1, at 10k steps it's -0.05, and at 14k steps it's -0.04; thus, 5k is undertrained, 10k is approximately at the optimality threshold of -0.052, and 14k is trained beyond optimality. Similar for $GPT2_{BASE}$.

accuracy as the this model on LAMBADA with 33% of the compute. Notice that we are comparing different model sizes, a large model trained to OPTIMALITY vs. a smaller model trained to completion.

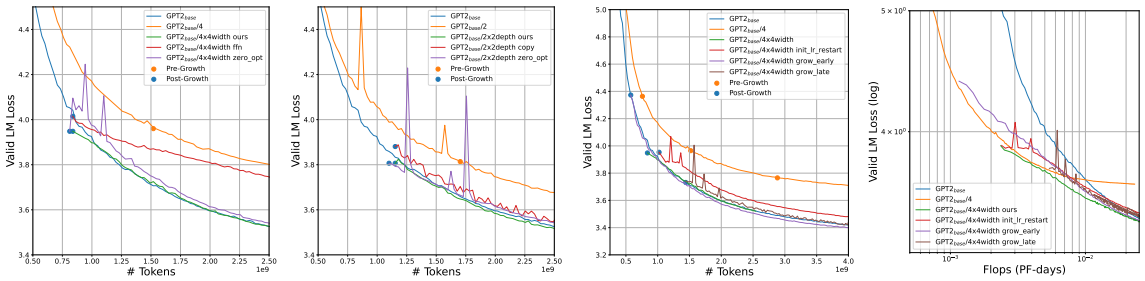
7. Related Work

Perhaps the most similar prior work is (**?**). However, this work did not provide a method to decide when to apply a growth operator and instead evaluated the performance of their operators at 100/300/500/700K steps of a small model, or applied a heuristic to equally distribute training steps among different model sizes. They did not discuss the optimizer state, and they reset the learning rate to the maximum value of 1e-4 at the beginning of each state without warmup. Their work built on progressive stacking (discussed below), and their proposed method to grow the width only grew the feed-forward layers instead of the entire model width.

(**?**) proposed "progressive stacking" which doubles the depth of a BERT transformer model by copying layers; to construct a $2L$-layer model from a L layer model it copies layer $i \leq L$ in the smaller model to layer $(i + L)$ in the larger model. The optimizer state is reset at the beginning of each stage, but the learning rate is kept the same as the prior stage. They use heuristics to set the stacking schedule: 50K steps for 3-layer, 70K steps for 6-layer model, 280K steps for 12-layer model. In an ablation study they examined the sensitivity to the number of steps before applying their growth operator and concluded that there is a threshold number of steps and for switching times such that switching to the larger model before the threshold led to compute savings, but switching after the threshold didn't. This is consistent with our results that showed the amount of compute saving is closely related to the stage length.

(**?**) proposed growing encoder-decoder transformers in the context of training a machine translation system. Their depth growth operator is identical to progressive stacking, although they explore operations that increase the model by only copying some of the layers from the small to large model (e.g. growing from 12 to 18 layers). They do not mention optimizer state, and reset the learning rate to max value at each stage.

In contemporaneous work, Gopher (**?**) introduced a method which tiles the weights from a small model to a larger one. However, their growth operator does not satisfy our two properties from Section 2, and they focus on training their models to completion. **?** proposes a way to initialize the grown weights by maximizing the gradient norm of the new weights for vision models. Their growth operator also requires specified activation functions that can not be directly applied to transformers. Finally, **?** proposed applying a curriculum learning strategy to the sequence length to reduce the training cost when training large language models. Their work is orthogonal to our method and our methods could be combined; we leave this to future work.



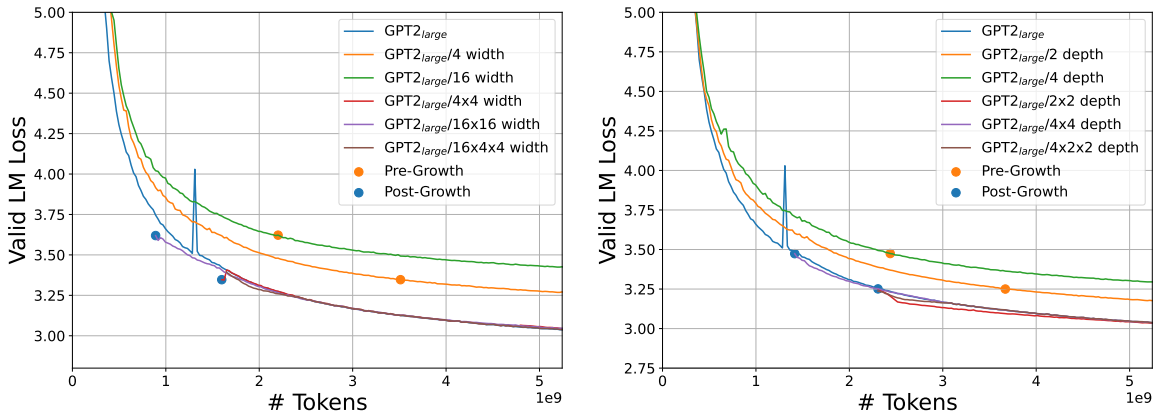
(a) Width op. and optimizer (b) Depth op. and optimizer (c) LR and optimal schedule (d) LR and optimal schedule

Figure 3: Comparing with three different baselines from prior work and ablation studies. • The width growth operator of **?** in $\text{GPT2}_{\text{BASE/4x4width ffn}}$ and the depth growth operator of **?** in $\text{GPT2}_{\text{BASE/2x2depth copy}}$ are not loss preserving (higher initial loss). Also, $\text{GPT2}_{\text{BASE/4x4width ffn}}$ is significantly underperforming the target model $\text{GPT2}_{\text{BASE}}$. • Resetting the optimizer state to zero instead of growing it (the `zero_opt` runs) have large instabilities and not preserving of the training dynamics. • 3c shows that restarting the learning rate schedule as in **?** is not training-dynamics-preserving. • 3c, 3d also show that not following our optimal schedule and grow the model too early or too late is still loss-preserving and training-dynamics-preserving but leads to lower compute saving

8. Conclusion and future work

One direction of future work is to combine batch size warmup (**?**) and sequence length growth (**?**) with our depth and width growth operators. Another applies our proposed methods to train a massive transformer.

We presented a staged training method for large transformer-based language models that grows the model size during training. We demonstrated the importance of two properties of the growth operators (loss-preserving and training-dynamics-preserving), and provided depth and width operators that satisfy both requirements. Finally, we devised a principled approach to find the optimal schedule and a simple method to apply the schedule in practice. Empirical evaluations show up to 22% compute saving.



(a) Width growth with $\text{GPT2}_{\text{LARGE}}$ as target

(b) Depth growth with $\text{GPT2}_{\text{LARGE}}$ as target

Figure 4: Similar to Figure 2, our width and depth growth operators are loss-preserving and training dynamics preserving. • $\text{GPT2}_{\text{LARGE/16x16-WIDTH}}$ indicates starting from a 16x smaller model then growing it 16x by doubling the width twice. • $\text{GPT2}_{\text{LARGE/16x4x4-WIDTH}}$ indicates growing the model 16x over two stages, by doubling the width once, continue training the model, then doubling the width again. • The same applies to the depth growth operator.

A. Additional Plots and Results

In Figure 4 and 5, we show loss curve plot and compute plot for applying growth operator to $\text{GPT2}_{\text{BASE}}$ model. These suggest our growth operator is loss-preserving, training dynamic preserving and saves compute to train the target BASE size model.

In Figure 6, we demonstrate the loss curve and derivatives regarding the validation loss and compute in log scale. It clearly shows the used practical threshold gives us a good estimate of the OPTIMALITY of the loss curve.

In Figure 7, we present the evaluation results for every checkpoints across the steps for target and grown $\text{GPT2}_{\text{BASE}}$ model. The performance of the $\text{GPT2}_{\text{BASE/2x2depth}}$ can always perform on par with or better than the baseline while $\text{GPT2}_{\text{BASE/4x4width}}$ underperforms baseline in the initial phase after growing but can catch up quickly. This also explains the potential negative compute we have in the main text in the early phase of the evaluation and better compute saving at OPTIMALITY.

B. Model Architecture

We use $\text{GPT2}_{\text{BASE}}$ and $\text{GPT2}_{\text{LARGE}}$ models. For $\text{GPT2}_{\text{BASE}}$ model, it consists of 12 layers, 768 hidden dimensions, 12 heads and 125M parameters. For $\text{GPT2}_{\text{LARGE}}$ model, it consists of 24 layers, 1536 hidden dimensions, 16 heads and 760M parameters.

C. Optimal Stage Schedule

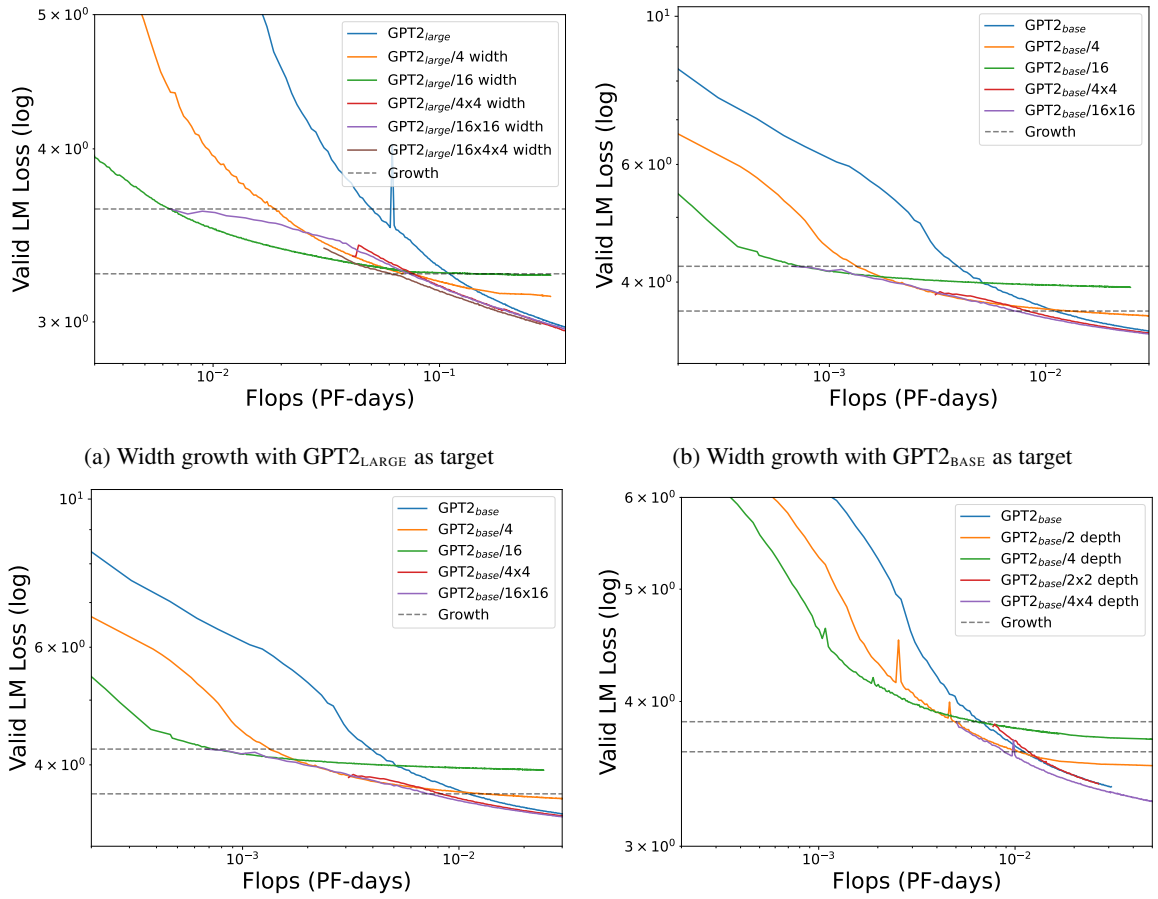
This appendix defines a constrained optimization problem that produces the optimal staged-training schedule. The scaling laws of **?** derived empirical fits for the language model loss L as it relates to the total amount of compute C, number of non-embedding parameters N, number of gradient update steps S, and batch size B. The total compute[3] is given by

$$C \approx 6NBS, \tag{10}$$

and the loss L for any model size N and number of steps S is given by:

$$L(N, S) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{S_c}{S}\right)^{\alpha_S} \tag{11}$$

[3]This neglects contributions proportional to the context length, n_{ctx}, and may not be valid in regime of large n_{ctx} where $n_{ctx} \geq 12d_{model}$



(a) Width growth with $\text{GPT2}_{\text{LARGE}}$ as target

(b) Width growth with $\text{GPT2}_{\text{BASE}}$ as target

(c) Depth growth with $\text{GPT2}_{\text{LARGE}}$ as target

(d) Depth growth with $\text{GPT2}_{\text{BASE}}$ as target

Figure 5: Growth operator and total compute. Our grown models are saving compute compared with target model.

when training at the critical batch size $B_{crit} = \frac{B_*}{L^{1/\alpha_B}}$, and where $\alpha_N, \alpha_S, \alpha_B, N_c, S_c, B_*$ are all model-specific constants.

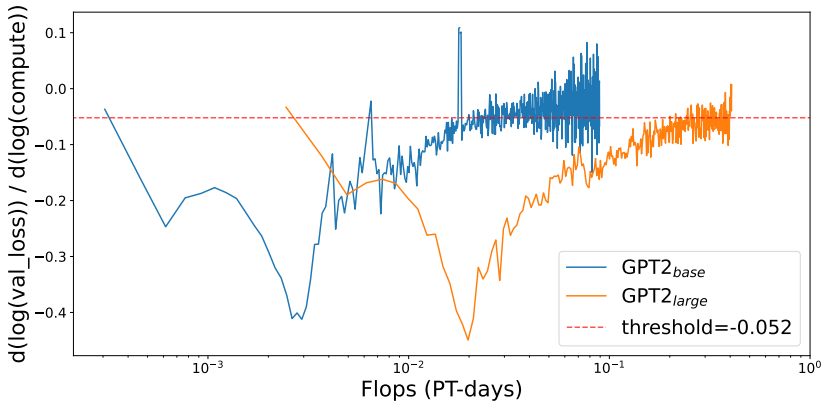
Our goal is to minimize the total compute to train a model of a given size N_{target} to a given target loss L_{target}. We assume we have access to perfect growth operators that are loss-preserving and training dynamics-preserving, and that can grow from any model size to any other size[4]. We consider a training regime consisting of a number of stages. In each stage k we train a model with size N_k for S_k gradient steps, with the goal of reaching size N_{target} and achieving a target loss L_{target} at the end of the final stage. We assume that $N_k \geq N_{k-1}$, and that there exists some way to initialize the model with size N_k from one of size N_{k-1} without changing the loss. For simplicity, we neglect the batch size contribution to compute, and assume training is always at the critical batch size $B_*/L_{target}^{1/\alpha_B}$.

With these assumptions the total compute at the end of training for M stages is:

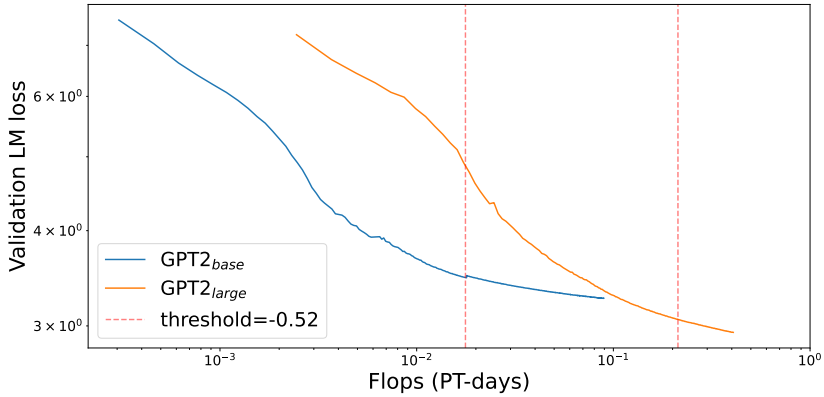
$$C = \sum_{k=1}^{M} 6N_k \frac{B_*}{L_{target}^{1/\alpha_B}} S_k \tag{12}$$

We can compute the loss at the end of each stage in an iterative fashion. The loss at the end of the first stage is given by $L_1(N_1, S_1)$ from Eqn. 11. Then for each subsequent stage, we assume the loss curves can be translated and the loss at the end of stage k is computed by starting with the loss at the end of the prior stage and decreased for S_k steps. To do so, first compute the effective number of steps $S_{eff,k}$ needed to reach initial loss for the stage L_{k-1} with model size N_k, and then

[4]We can restrict the model size increases by adding additional constraints.



(a) First derivative on the validation loss curve for GPT2$_{\text{BASE}}$ and GPT2$_{\text{LARGE}}$. This is an approximation to the training dynamics $\frac{\partial \mathcal{L}}{\partial C}$.



(b) Validation loss curve for GPT2$_{\text{BASE}}$ and GPT2$_{\text{LARGE}}$

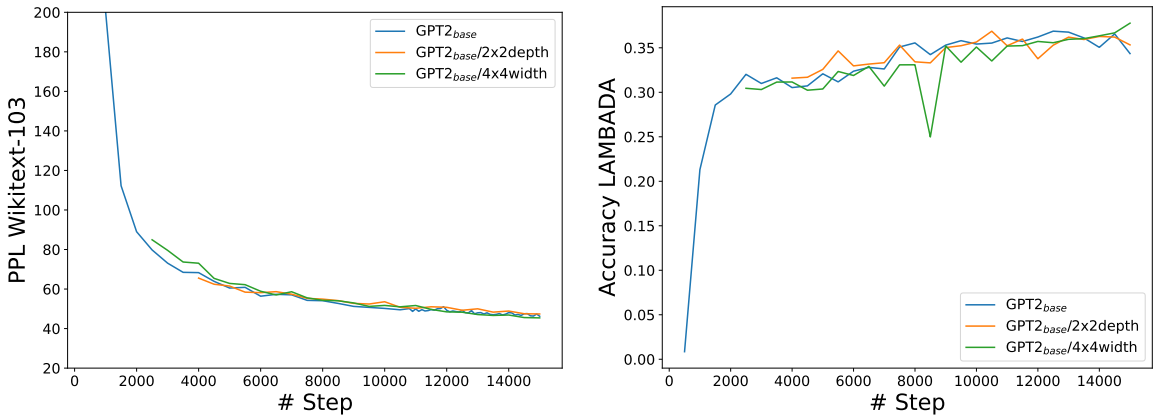
Figure 6: 6a shows the first derivative of the validation loss curve for GPT2$_{\text{BASE}}$ and GPT2$_{\text{LARGE}}$. The horizontal dotted lines shows the empirical value of the derivative threshold we use to identify the point of OPTIMALITY. 6b shows how the threshold value in 6a maps to a point in the loss curve for both model sizes (the first vertical line for GPT2$_{\text{BASE}}$ and the second for GPT2$_{\text{LARGE}}$). Both point show slowed training and the end of the compute-efficient regime.

compute the loss at the end of the stage by $L_k(N_k, S_{eff,k} + S_k)$. In summary:

$$
\begin{aligned}
L_1 &= \left(\frac{N_c}{N_1}\right)^{\alpha_N} + \left(\frac{S_c}{S_1}\right)^{\alpha_S} \\
L_k &= \left(\frac{N_c}{N_k}\right)^{\alpha_N} + \left(\frac{S_c}{S_{eff,k} + S_k}\right)^{\alpha_S}, k > 1 \\
S_{eff,k} &= \frac{S_c}{(L_{k-1} - (N_c/N_k)^{\alpha_N})^{1/\alpha_S}}.
\end{aligned}
$$

With this in hand, we can use a constrained optimizer to solve for the optimal schedule by minimizing Eqn. 12, subject to the constraint that the final model size is the target size, and loss at the end of training is the target loss. Formally,

$$
\min_{\{(N_k, S_k)\}} \sum_{k=1}^{M} 6 N_k \frac{B_*}{L_{target}^{1/\alpha_B}} S_k
$$



(a) Wikitext-103 evaluation with $\text{GPT2}_{\text{BASE}}$ as target

(b) LAMBADA evaluation with GPT2BASE as target.

Figure 7: The detailed evaluation plots on Wikitext-103 and LAMBADA for $\text{GPT2}_{\text{BASE}}$ trained from scratch and two grown model using depth or width operator. The performance of the $\text{GPT2}_{\text{BASE/2x2depth}}$ perfectly follows or even outperform the target model, while $\text{GPT2}_{\text{BASE/4x4width}}$ underperforms the target model initially after growing but it catches up quickly. This explains the initial "negative" compute saving in Table 2 for the width growth operator, followed by positive compute saving as the model trains longer.

Number of stages	Compute reduction factor
1	1.0
2	0.83
3	0.792
4	0.779
5	0.771
10	0.763

Table 3: Decrease in compute costs over an optimal single stage training regime.

subject to

$$\begin{aligned} L_M &= L_{target} \\ N_M &= N_{target} \\ 0 &< N_1 \\ N_{k-1} &\leq N_k, k > 1 \\ 0 &\leq S_k \end{aligned}$$

Note that in the single stage case ($M = 1$) with no N_{target} condition, this formulation reduces to the optimal calculation in Appendix C of **?** to find the optimal model size to reach a target loss. This matches our training to OPTIMALITY.

Measuring compute saving To measure the compute saving from staged training to reach a certain L_{target}, we use a single staged training ($M = 1$) with no N_{target} condition, and use our optimization algorithm to find the optimal compute and model size; this becomes N_{target}. Next, we run the optimization problem with N_{target}, L_{target}, and $M > 1$ to get the optimal training schedule and the expected compute, which we compare with $M = 1$ to find the expected compute saving.

Observations An implementation of this optimization shows that the increase in compute efficiency for using multiple stages is independent of the target loss L_{target}, and quickly approaches 0.76 as the number of stages increases (using the scaling parameters for autoregressive transformer language models).

We also found that constraining the ratio between consecutive model sizes ($\frac{N_k}{N_{k-1}}$) to be 2, 4 or 8, leads to almost the same compute savings. These constraints come from the practical constraints of our implementation of the growth operators.

Stage	N_k	S_k	L_k	C_k
1	2.7M	15.4K	4.09	0.0033
2	22.1M	24.3K	3.58	0.0154
3	71.9M	30.8K	3.31	0.0365
4	163M	36.0K	3.13	0.0665
5	306M	40.5K	3.00	0.1056

Table 4: Sample optimal schedule for a five stage regime to train to $L_{target} = 3$ showing the number of non-embedding parameters N_k, the number of gradient steps S_k, the loss at the end of the stage L_k, and the compute in the stage C_k.

D. Practical Stage Schedule

The empirical values we estimated for the constants are:

$$\begin{aligned}
\tau_{opt} &= -0.052 \\
\tau_{depth} &= -0.0575 \\
\tau_{width} &= -0.0475 \\
\tau_{depth-width} &= -0.03 \\
\rho_{depth} &= 0.70 \\
\rho_{width} &= 0.55 \\
\rho_{depth-width} &= 0.40
\end{aligned}$$