Efficient Scaling of Language Model pretraining

Iz Beltagy

Larger, better, more expensive

LM are getting larger, more expensive

But also getting better, new capabilities, zero-shot generalization, better generation

As we scale the models to larger sizes, they will keep getting better

This requires new efficient methods for training, and better modeling choices

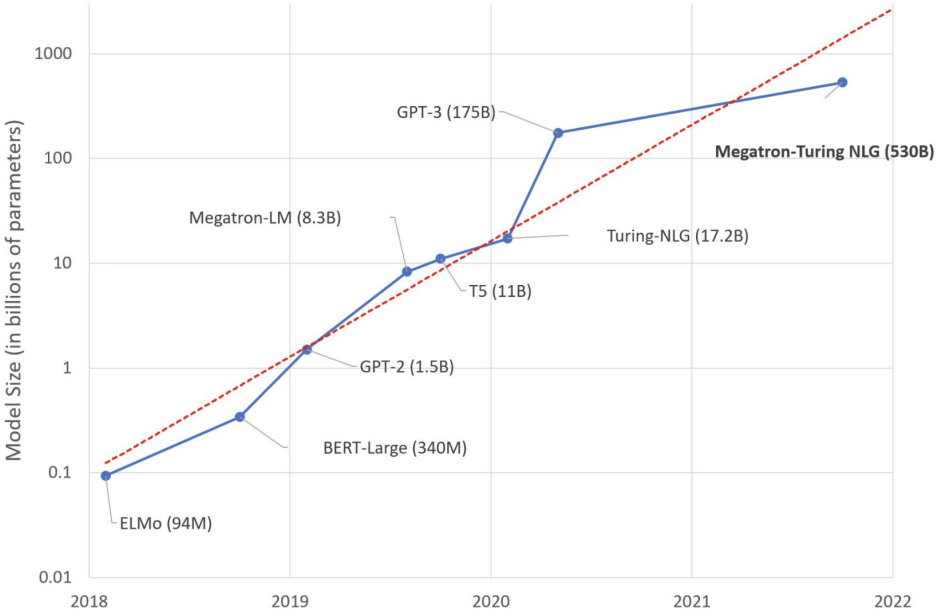


Photo credit: Microsoft Research Blog, Alvi et. al., 2021

Larger, better, more expensive, and efficient

I will discuss two recent works

- Staged Training for Transformer Language Models

  A training regime that can save up to 20% of the compute

- What Language Model to Train if You Have One Million GPU Hours?

  Modeling choices to get the best model

Staged Training for Transformer Language Models

Sheng Shen, Pete Walsh, Kurt Keutzer, Jesse Dodge, Matthew Peters, Iz Beltagy

Staged Training

Goal: Train a large language model

Now (1 stage training): [Large Model] ⇒ {Train} ⇒ [Target Model]

Proposed (multi-stage training):

[Small Model] ⇒ {Train} ⇒ {Grow} ⇒ [Larger Model] ⇒ {Train} ⇒ {Grow} .... ⇒ [Target Model]

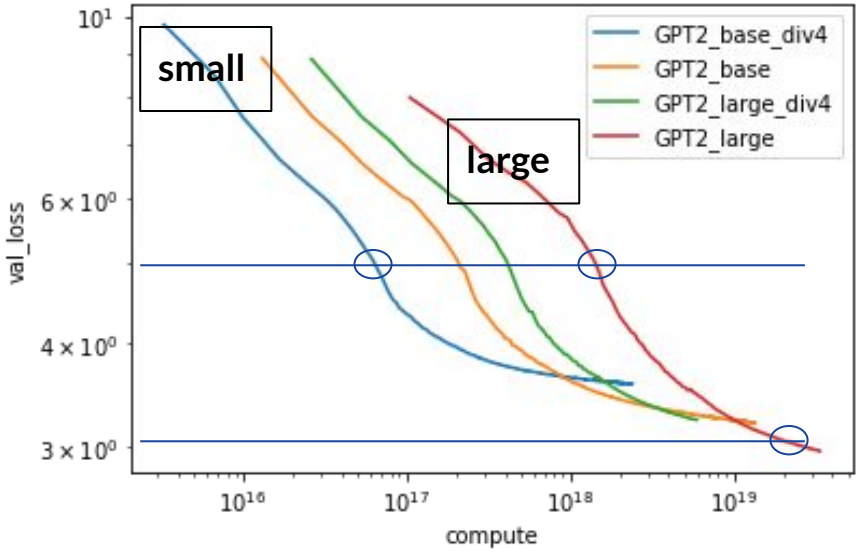
Prior work (e.g. [1]) proposed the same method but missing key ideas and intuitions to get it to work reliably and achieve max compute saving

[1] Net2Net: Accelerating Learning via Knowledge Transfer, Chen et. al., ICLR 2016

Staged Training - Intuition

Smaller models are **initially faster** to train then they **plateau**

Larger models initially **slower** than smaller models but eventually become **more efficient**



Staged Training - Intuition

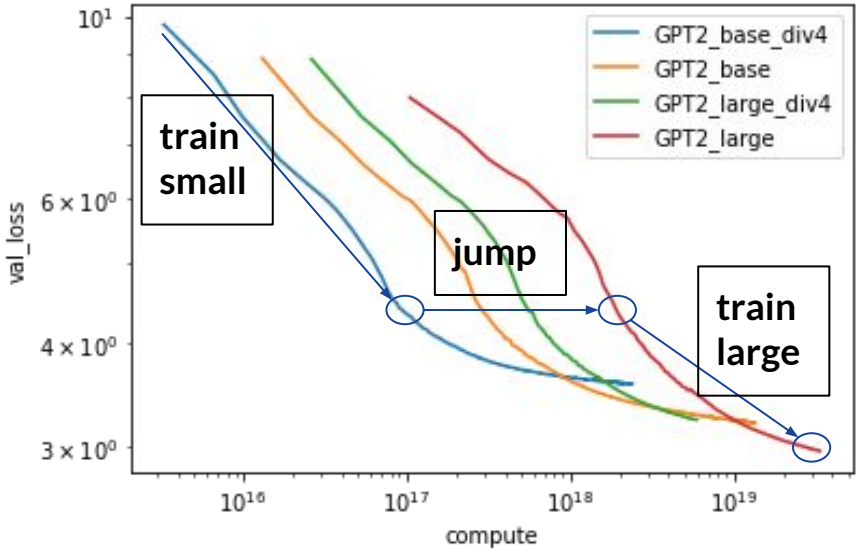
Training regime:

- train small model until loss slows down
- “jump” to a larger one
- train larger one until loss slows down

Why?

- the jump saves compute
- intermediate model sizes for free

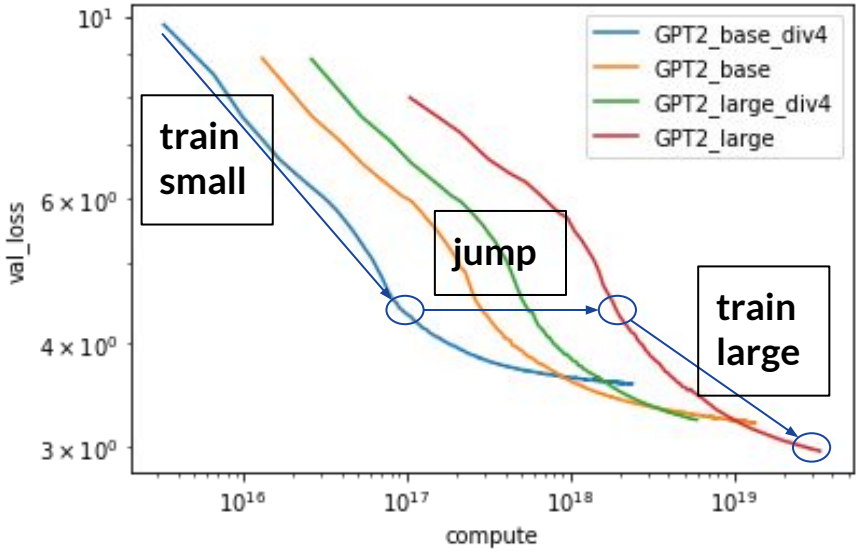
We call the “jump” a **Growth Operator**



Staged Training - Intuition

- How to jump **effectively**

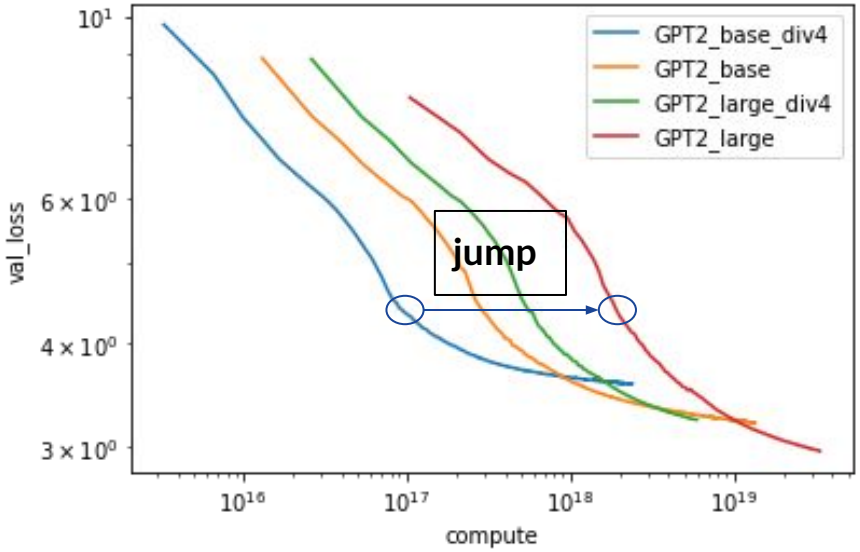
- How to identify the 3 points for **optimal** compute saving



Properties for Growth Operator

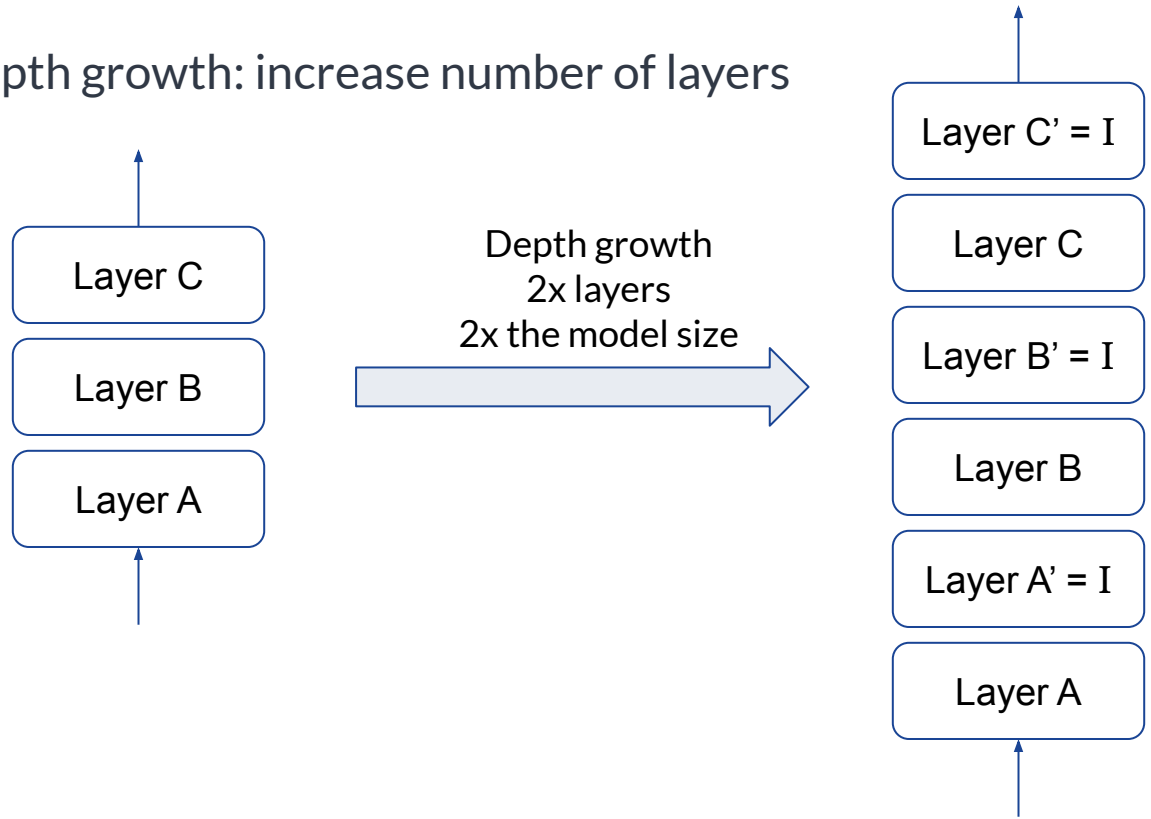
To effectively jump between learning curves, growth operator should have the following properties

1) **loss-preserving** (function-preserving): loss before growing model is the same as after



Growth operators - Depth

Depth growth: increase number of layers

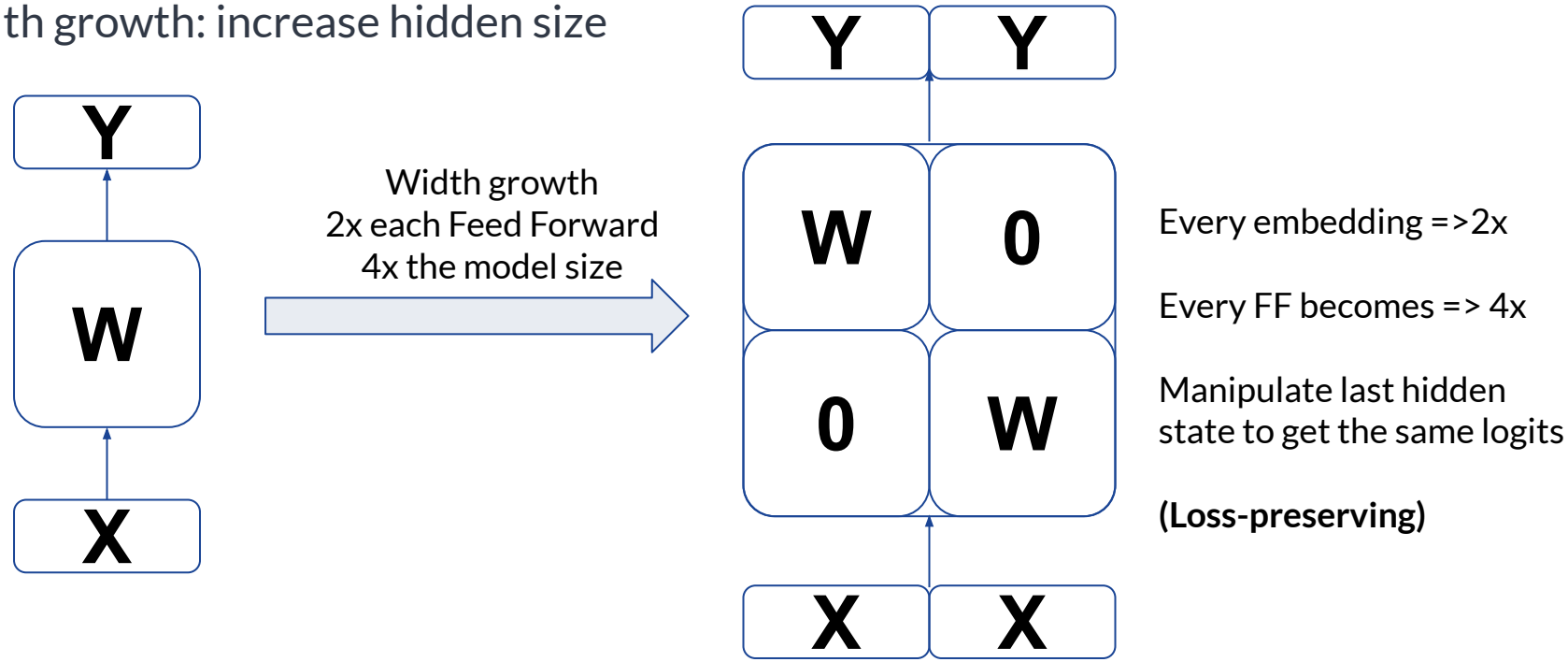


Copy layers then manipulate a few weights to convert it into an Identity

(Loss-preserving)

Growth operators - Width

Width growth: increase hidden size



Every embedding =>2x

Every FF becomes => 4x

Manipulate last hidden state to get the same logits

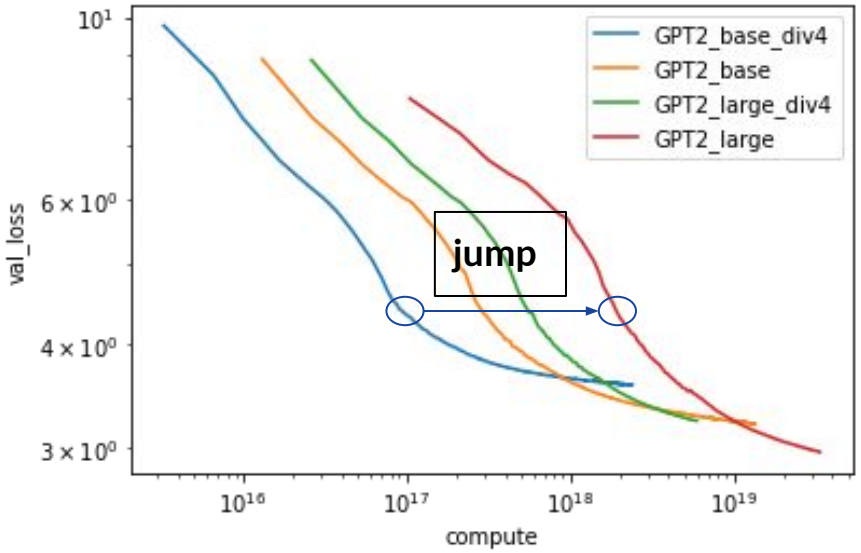
(Loss-preserving)

Properties for Growth Operator

To effectively jump between learning curves, growth operator should have the following properties

1) **loss-preserving** (function-preserving): loss before growing model is the same as after

2) **training-dynamics-preserving**: rate of loss change after growing the model is the same as training the model from "scratch"

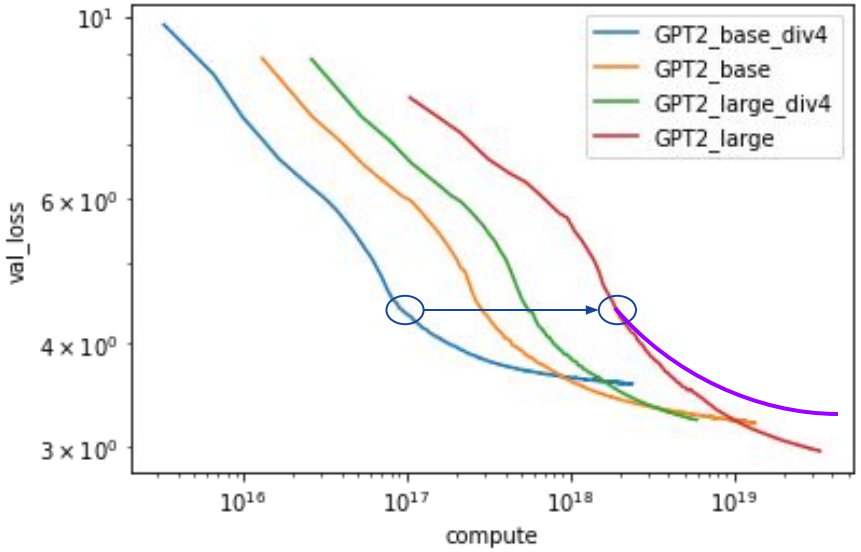


Properties for Growth Operator

training-dynamics-preserving: after growth, model trains as fast as the model trained from scratch

An ineffective growth operator creates a larger model but one that doesn't train fast

We are the first to recognize the importance of this property



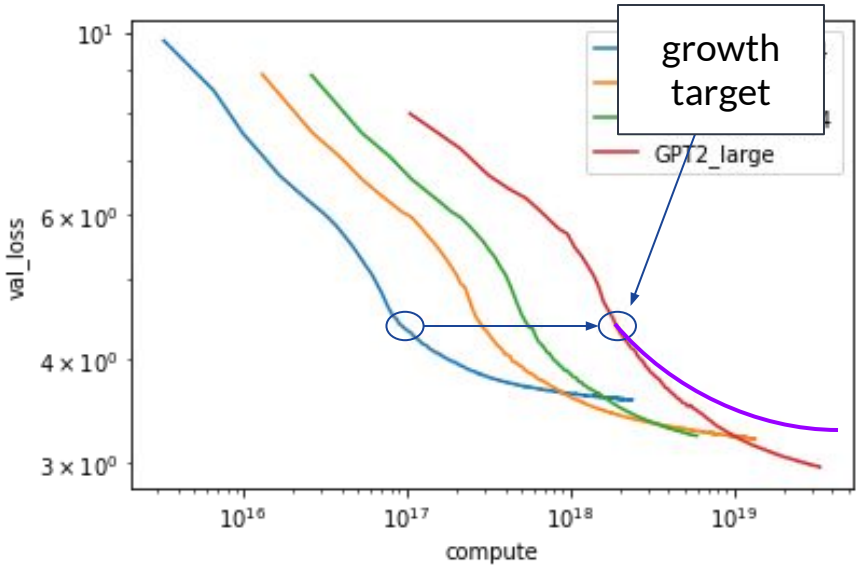
Properties for Growth Operator

To preserve training dynamics, growth operator should grow **whole training state (optimizer state and LR)** not just model

Intuition - get the whole training state to match that of one trained from scratch

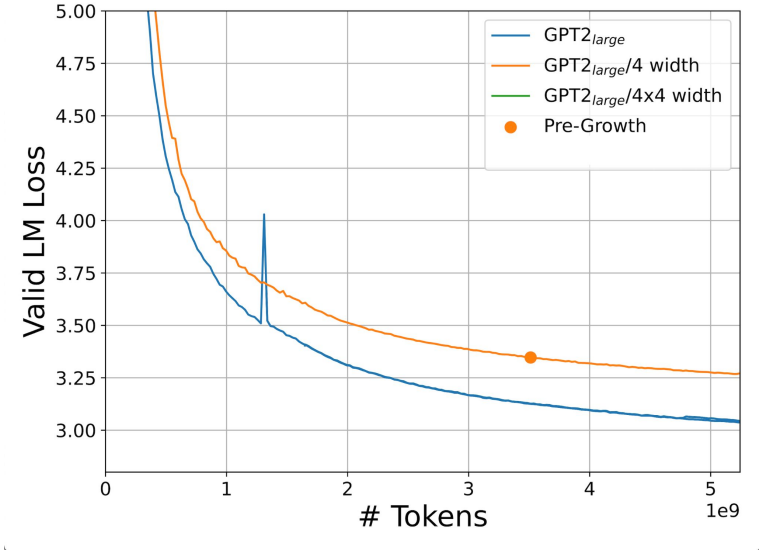
LR: use LR at growth target

Optimizer: grow optimizer state with a mostly similar growth operator to model growth (check paper for details)



Properties for Growth Operator - Evaluation

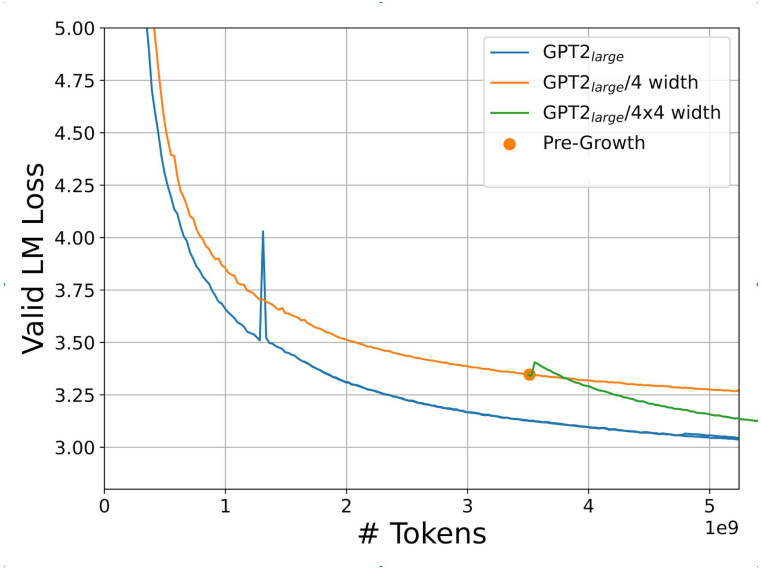
Two models trained from scratch



Properties for Growth Operator - Evaluation

Grow width of the small model. Grown model matches size of the larger model

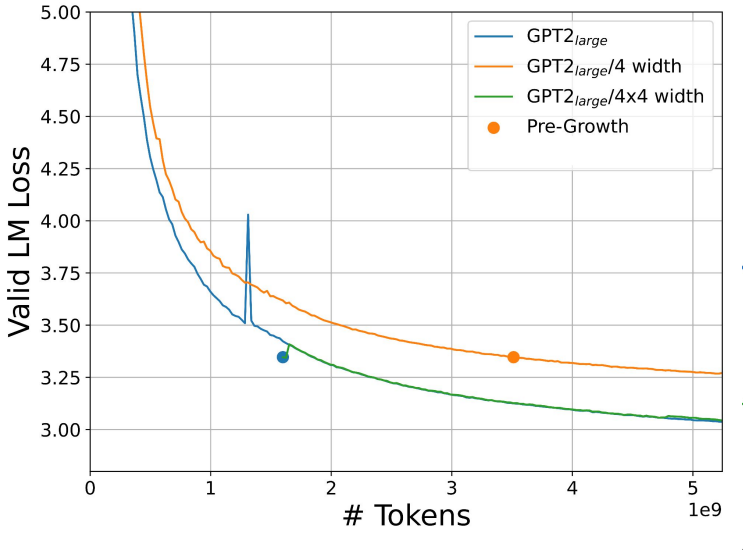
(Loss preserving)



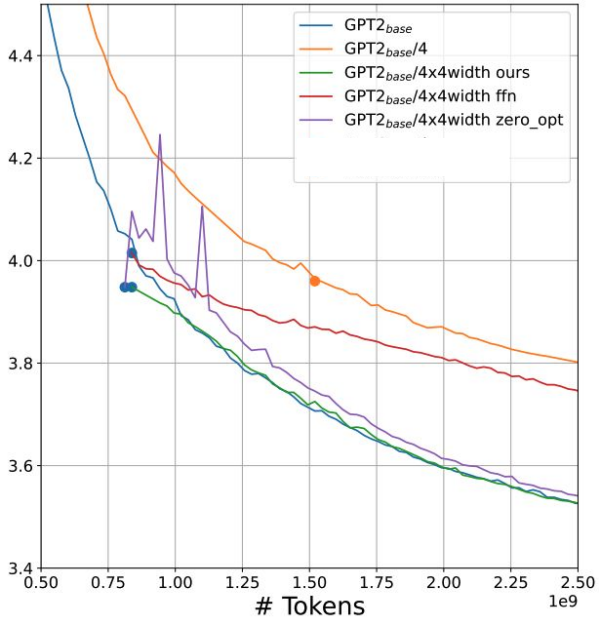
Properties for Growth Operator - Evaluation

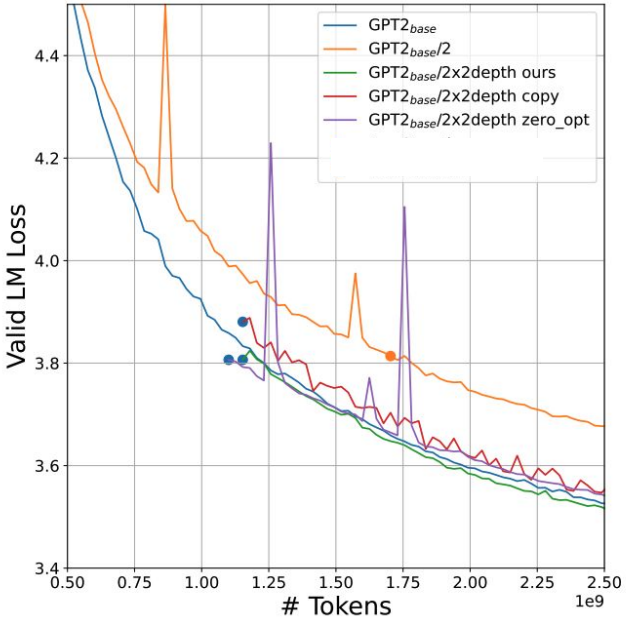
Overlay grown model over larger model trained from scratch

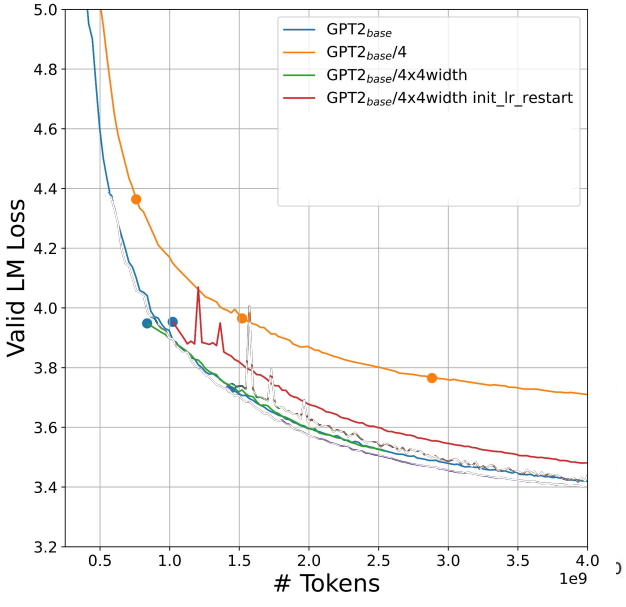
(Preserving training dynamics)



Properties for Growth Operator - Evaluation







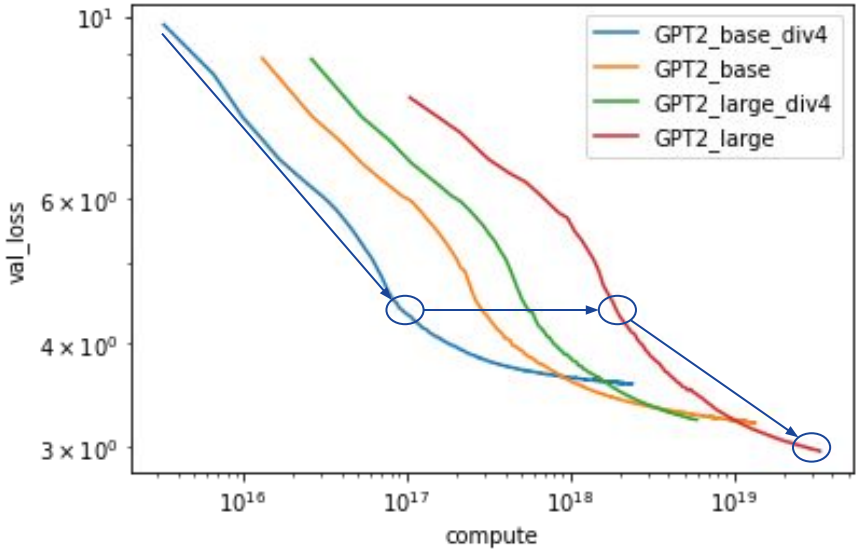
Staged Training

- Properties of growth operators
  - Loss preserving
    - Depth and Width operators
  - Training dynamics preserving
    - Optimizer and Learning rate
- Optimal Training Schedule
- Practical Training Schedule
- Evaluation

Optimal Training Schedule

Prior work splits the compute heuristically between the stages.

Here we see there's a **precise schedule** with the **optimal compute saving**

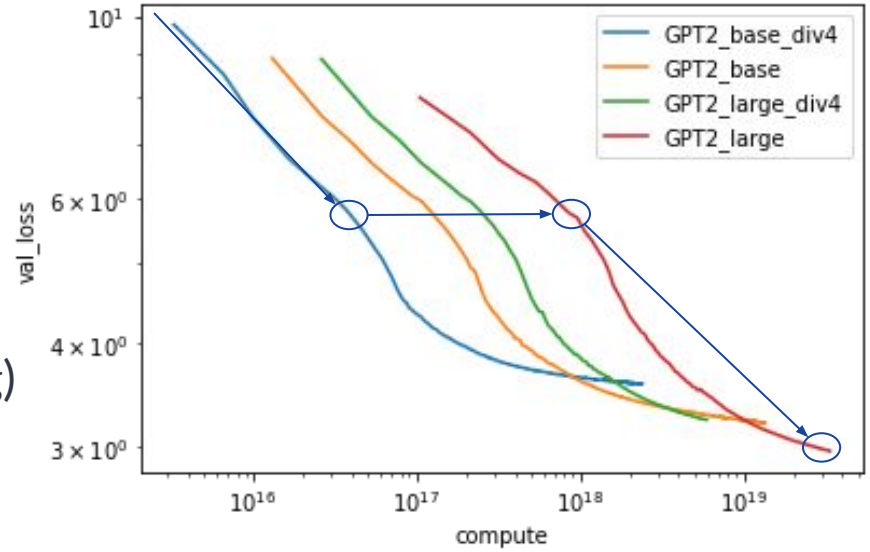


Optimal Training Schedule

Prior work splits the compute heuristically between the stages.

Here we see there's a precise schedule with the optimal compute saving

Grow model **too early**, and you will waste the **opportunity to save more** (x-axis is log)



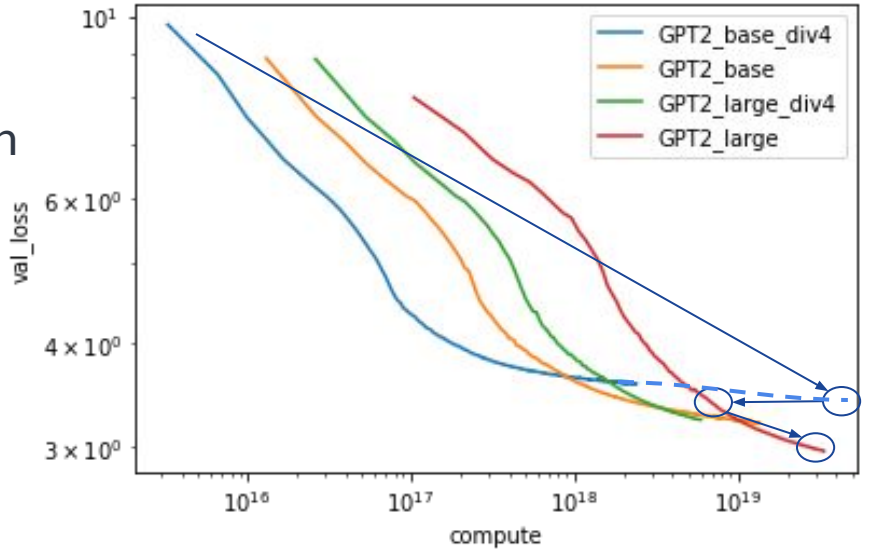
Optimal Training Schedule

Prior work splits the compute heuristically between the stages.

Here we see there's a precise schedule with the optimal compute saving

Grow model too early, and you will waste the opportunity to save more (x-axis is log)

Grow model **too late**, and you will **waste more compute** than you save



Scaling Laws

Earlier learning curves are empirical, but they can be predicted using scaling laws [Kaplan et. al., 2020]

Scaling laws predict **loss (L)** as function of **model size (N)** and **compute (C)** and a list of empirically fitted constants

$$C \approx 6NBS,$$

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

Optimal Training Schedule

Optimal training schedule is a constrained optimization problem

minimize $\sum$(compute per stages)

subject to

- Scaling laws L = F(C, N)
- Size increments constrained to 2x or 4x
- Target model size N_target
- Target model loss L_target

Optimal Training Schedule - Observations

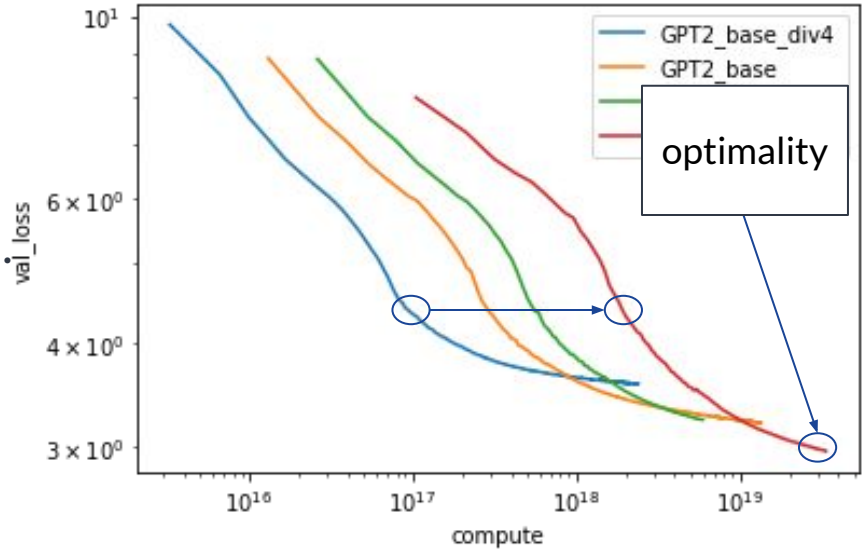
Assuming scaling laws fitting of Kaplan et. al.,

With no staged training (#stages = 1):

- The optimal solution reproduces the optimal compute allocation of Kaplan et. al.

- Train a large model and stop long before convergence

- We call this point **optimality**



Optimal Training Schedule - Observations

Assuming scaling laws fitting of Kaplan et. al.,

Expected compute saving plateaus at 3 stages

(grow model 2 times)

With 3 stages, compute allocation is:

- stage 1: 13%
- stage 2: 10%
- stage 3: 62%
- save: 15%

# stages	saving
1	0 %
2	10%
3	15%
4	16.5%
5	17.4%
6	17.9%
7	18.1%

Staged Training

- Properties of growth operators
  - Loss preserving
    - Depth and Width operators
  - Training dynamics preserving
    - Optimizer and Learning rate
- Optimal Training Schedule
- Practical Training Schedule
- Evaluation

Practical Training Schedule

Optimal schedule assumes you know scaling laws

Fitting scaling laws is expensive and time consuming because you need to train many models of many different sizes

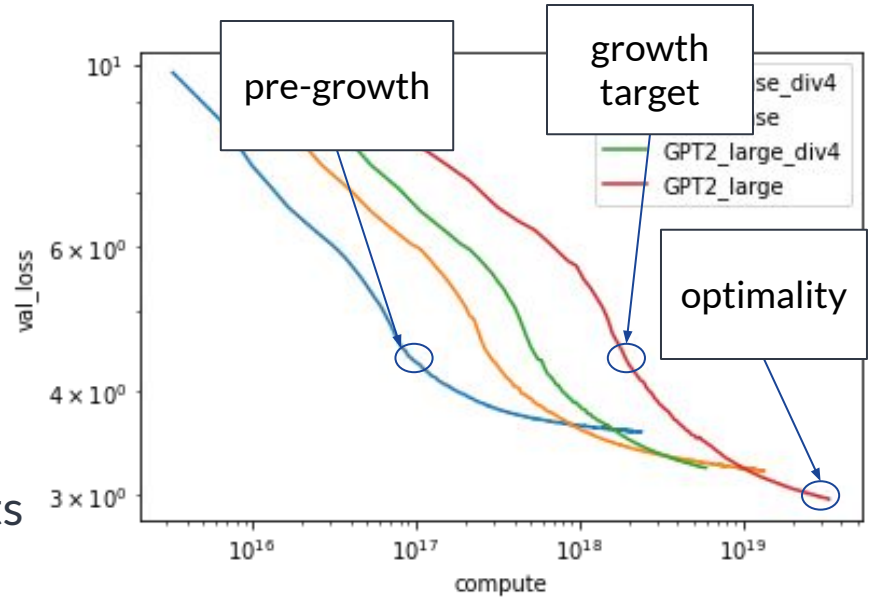
Instead, we develop a more practical training schedule that follows the intuitions we learn from the optimal one

Practical Training Schedule

Each stage is characterized by 3 points:

- **pre-growth**: when to grow
- **growth-target**: LR after growth
- **optimality**: stop training

We don't need to fit scaling laws and solve optimization problem. Just find the 3 points



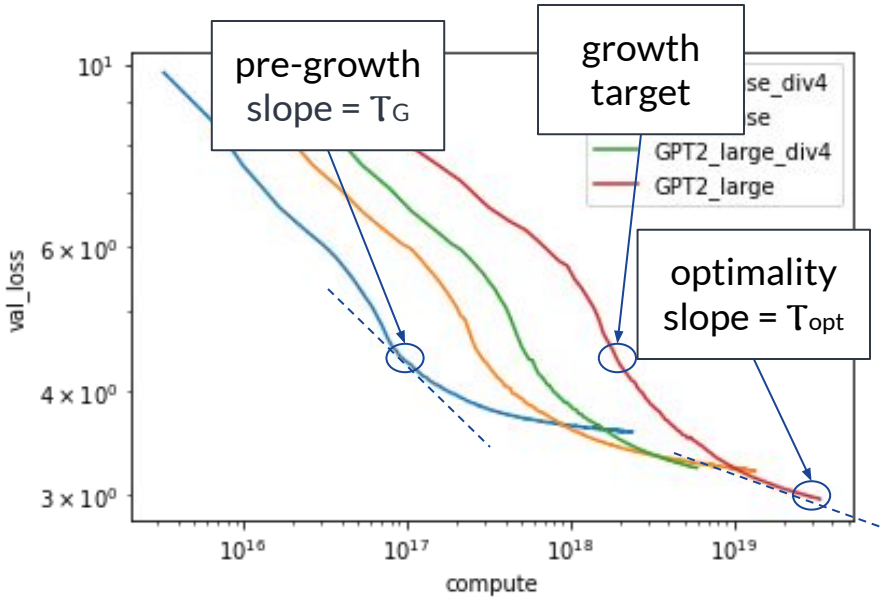
Q: Can we find them visually from the learning curves?

A: No, because we want to avoid training the large model from scratch

Practical Training Schedule

Each stage is characterized by 3 points:

- **pre-growth**: when to grow
  - slope of learning curve, τ_{depth}, τ_{width}
- **optimality**: stop training
  - slope of learning curve, τ_{opt}
- **growth-target**: LR after growth
  - ratio $\frac{\text{steps@pre-growth}}{\text{steps@growth-target}} = \varrho$
  - function of the growth OP: ϱ_{depth}, ϱ_{width}



Practical Training Schedule

T_g, T_{opt}, ρ_g are independent of the model size, so

Replaced fitting scaling laws with estimating 3 simple constants

Procedure:

- Train small models for a few thousand gradient updates:
- Use them to estimate T_g, T_{opt}, ρ_g
- Then, apply the same constants when training larger models

Putting it all Together: Training Algorithm

START with a randomly initialized small model M

FOR number of stages REPEAT :

T = Topt IF last stage ELSE **Tg**

WHILE slope of learning curve < **T** :

Train M for another step, keep track of #steps S

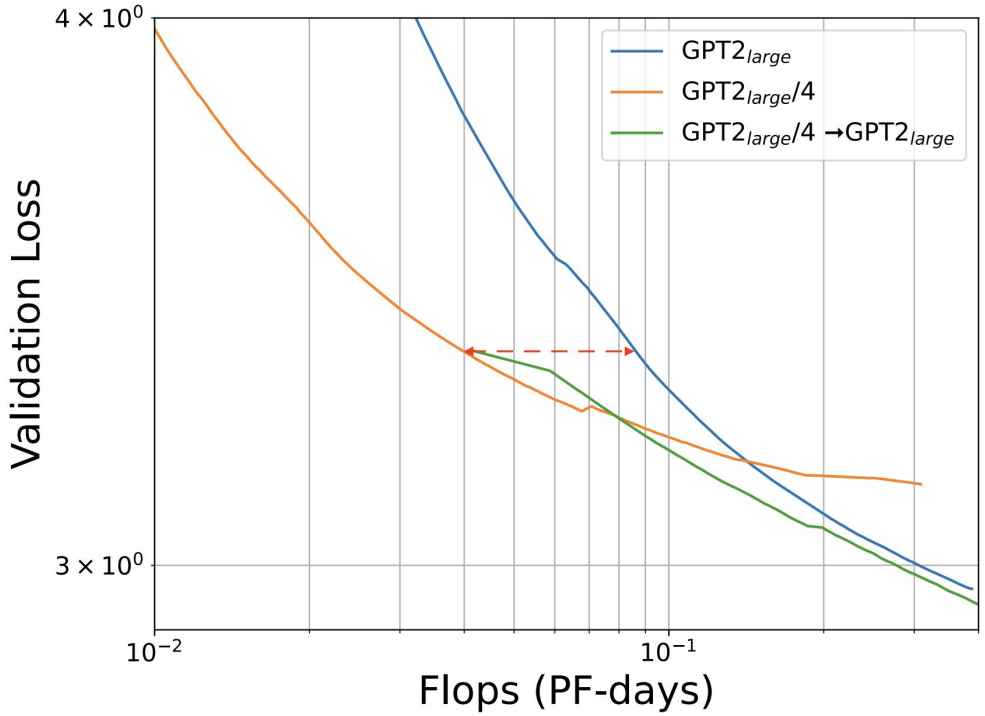
M = grow(M) // apply growth operator

S = S x ϱ_g // to set learning rate of the next stage

Staged Training

- Properties of growth operators
  - Loss preserving
    - Depth and Width operators
  - Training dynamics preserving
    - Optimizer and Learning rate
- Optimal Training Schedule
- Practical Training Schedule
- Evaluation

Evaluation - Pretraining loss



Evaluation - Pretraining loss

Percentage of compute saving

		$GPT2_{LARGE}$		$GPT2_{BASE}$	
		At OPT	After OPT	At OPT	After OPT
2 stage practical	2xW	7.3	5.2	20.2	19.7 .3
	4xW	5.3	3.8	8.6	5.5 .0
	2xD	11.0	6.1	20.4	19.8 .7
	4xD	7.3	5.2	10.1	6.4 .8
	2xDxW	5.4	3.8	9.5	6.83 .0
3 stage practical	2x2xW	10.9	7.8	17.9	11.4 .8
	2x2xD	14.5	10.4	21.4	15.9 .3

Evaluation - Zero-shot downstream

Percentage of compute saving

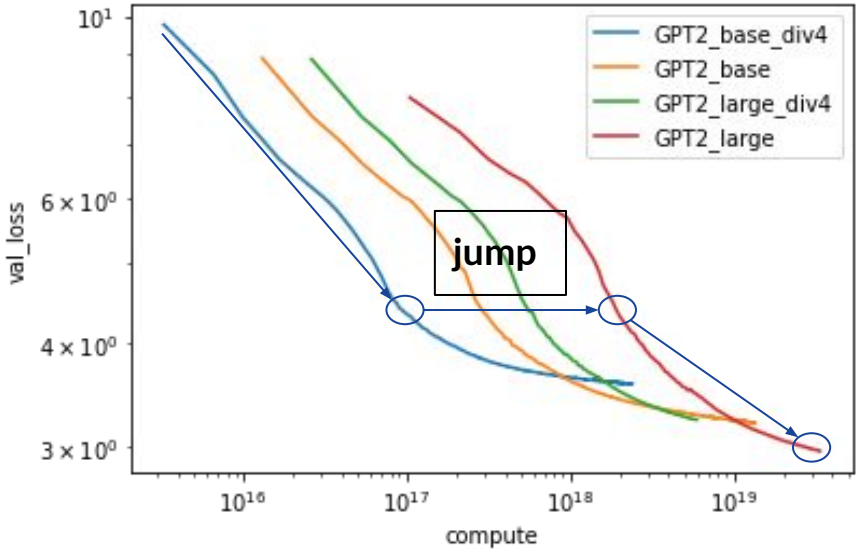
		Zero-shot Wikitext-103 (PPL)				Zero-shot LAMBADA (accuracy)			
		$GPT2_{LARGE}$		$GPT2_{BASE}$		$GPT2_{LARGE}$		$GPT2_{BASE}$	
		@OPT	AFTR OPT	@OPT	AFTR OPT	@OPT	AFTR OPT	@OPT	AFTR OPT
2 stage practical	2xwidth	-0.25	8.7	5.9	3.8	12.3	14.8	3.3	10.6
	4xwidth	1.3	7.4	1.1	6.4	18.0	18.2	10.1	8.6
	2xdepth	13.5	11.4	33.5	17.5	23.5	21.4	16.8	13.9
	4xdepth	17.5	9.5	7.9	3.1	20.5	14.6	9.4	7.8
	2xdepthxwidth	-0.3	1.6	3.6	4.5	-37.5	0.7	6.4	6.4
3 stage practical	2x2xwidth	6.3	9.3	5.4	8.0	13.4	20.3	1.8	11.8
	2x2xdepth	12.5	12.8	14.3	11.8	14.5	23.6	10.6	15.0

Conclusion and Future Work

- How to jump **effectively**

- How to identify the 3 points for **optimal** compute saving

- Saved up to **20%** compute



Future work:

Staged training using Depth + Width + Sequence Length + Batch Size

What Language Model to Train if You Have One Million GPU Hours?

Thomas Wang, Teven Le Scao, Adam Roberts, Daniel Hesslow, Stas Bekman, Lucile Saulnier, Hyung Won Chung, M Saiful Bari, Stella Biderman, Hady Elsahar, Jason Phang, Ofir Press, Colin Raffel, Victor Sanh, Sheng Shen, Lintang Sutawika, Jaesung Tae, Zheng Xin Yong, Julien Launay, Iz Beltagy

Big Science Architecture and Scaling Group

You have 1Million GPU hours - Now what?

1Million GPU (A100) hours ~= GPT3 compute budget

What language model to train?

(architectures, pretraining objectives, model sizes, activation functions, position embeddings, layernorms, training data … etc)

This talk will only discuss architectures and pretraining objectives for zero-shot evaluation. Check paper for more details

What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization?

Large language models started to exhibit zero-shot generalizations

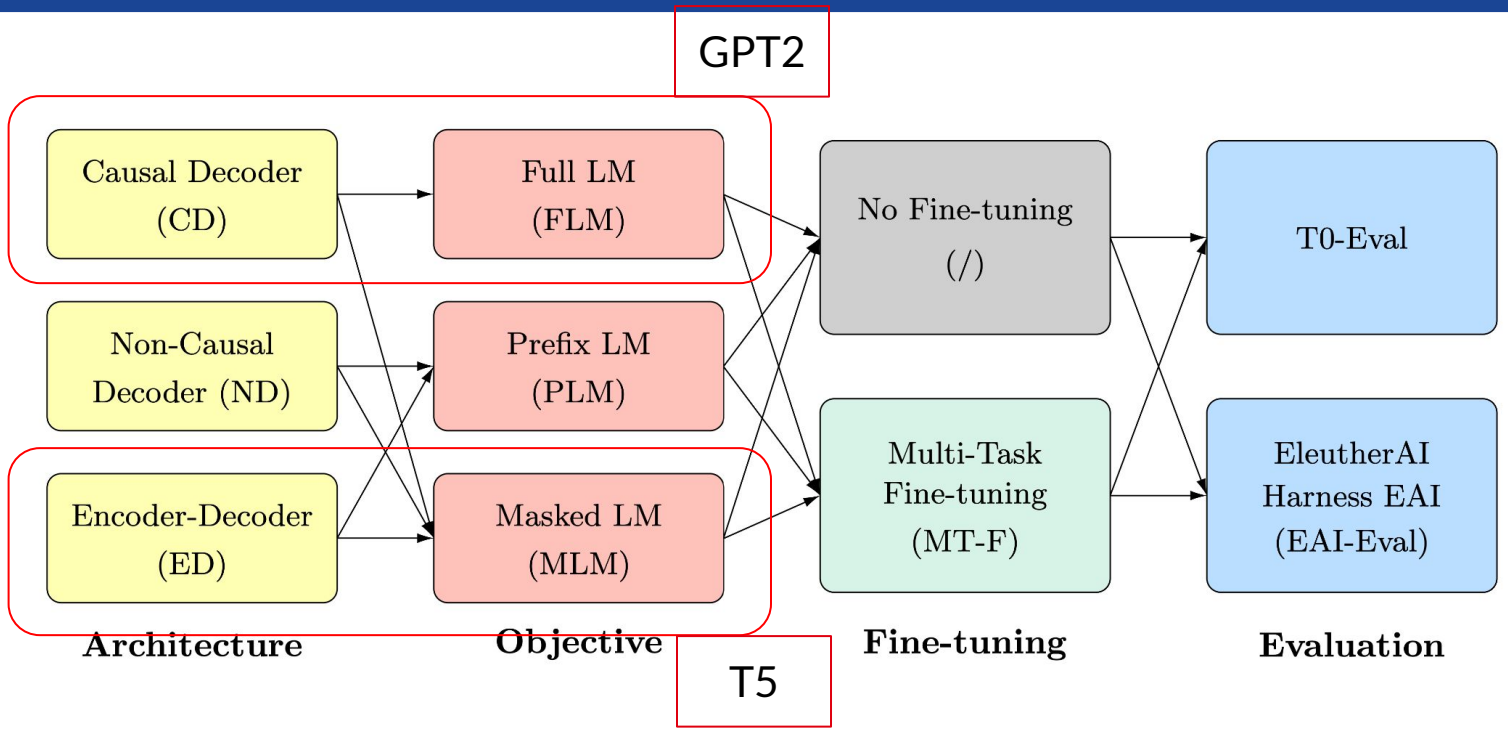
No systematic evaluation of **architectures** and **pretraining objectives** of SOTA models.

This is a large scale empirical evaluation of modeling choices and their impact on downstream **zero-shot generalization**

Two types of zero-shot generalization:

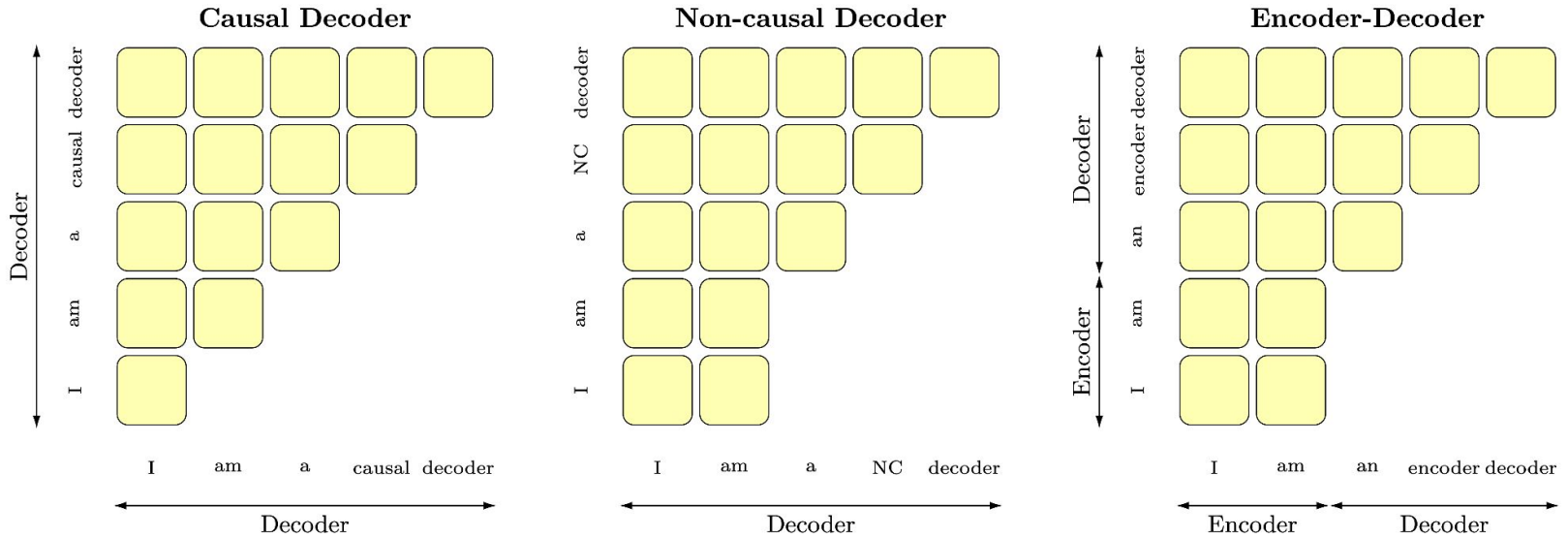
- Purely based on pretraining
- After multi-task supervised fine-tuning as in T0 [Sanh et. al.], FLAN [Wei et. al.]

Experimental Setup

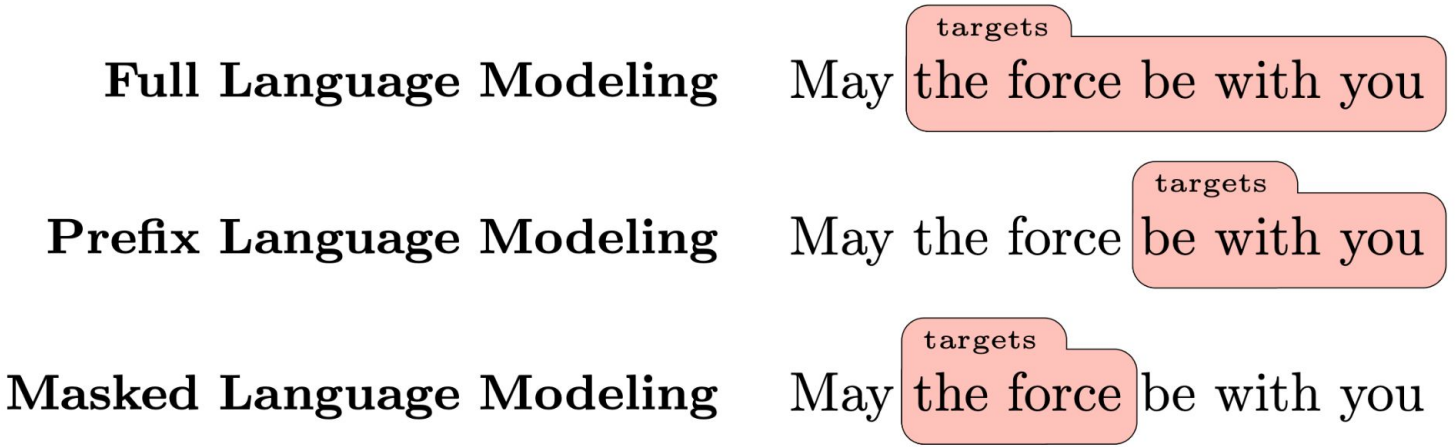


Experimental Setup - Architectures

Transformer models. All have decoders to support generation tasks. All seq2seq



Experimental Setup - Pretraining Objectives



Experimental Setup - Fine tuning

Supervised Multitask finetuning showed greatly improved zero-shot generalizations

e.g. T0 [Sanh et. al., 2021], FLAN [Wei et. al., 2021]

We use the T0 supervised multitask finetuning setup. Train for 10B tokens

Q: Do you expect the best architecture before and after T0 finetuning to be the same or different?

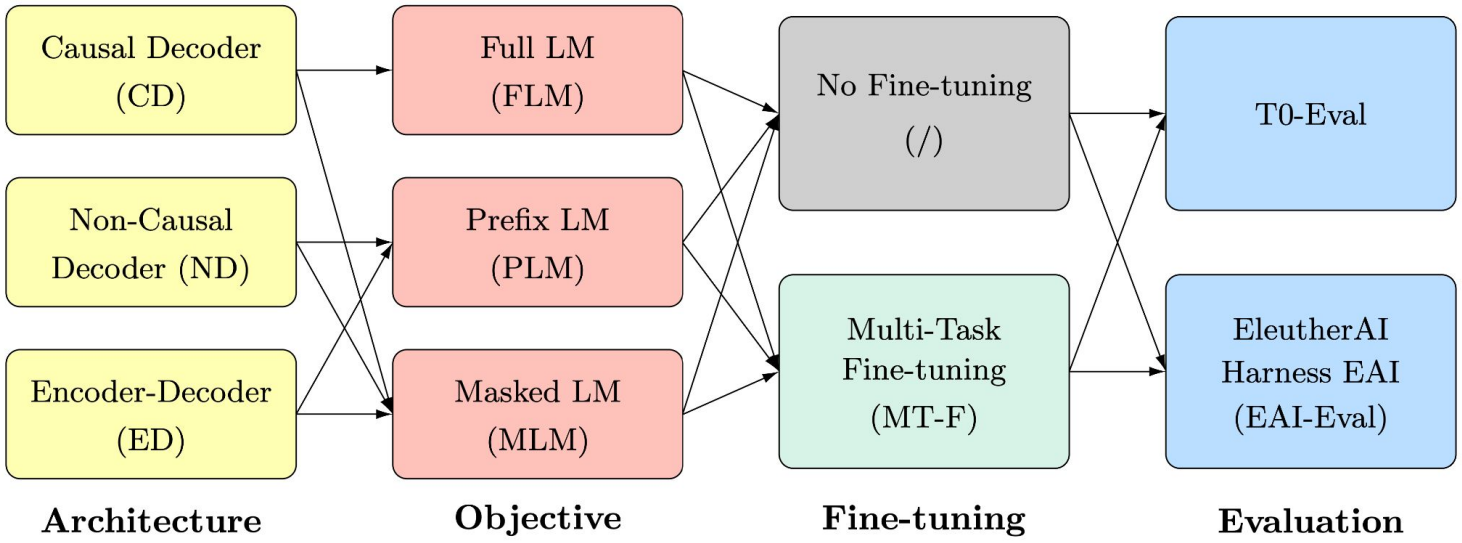
Experimental Setup - Evaluation Benchmarks

Focus on zero-shot evaluation

Two benchmarks:

- T0: using the T0 heldout evaluations
  - Zero-shot from task descriptions and prompt-engineering
- EAI: using the Eleuther AI evaluation benchmark
  - Zero-shot using prompt-engineering

Experimental Setup



Results - After Unsupervised Pretraining

	EAI-EVAL	T0-EVAL
FLM/PLM		
Causal decoder	**44.2**	**42.4**
Non-causal decoder	43.5	41.8
Encoder-decoder	39.9	41.7
MLM		
Causal decoder	random	
Non-causal decoder		
Encoder-decoder		

Causal decoder (GPT2-style) works the best

All MLM models are around random chance because they don't work for zero-shot

Results - After Supervised Multitask Finetuning

	EAI-EVAL	T0-EVAL
FLM/PLM		
Causal decoder	50.4	51.4
Non-causal decoder	48.9	54.0
Encoder-decoder	44.2	45.8
MLM		
Causal decoder	47.1	50.3
Non-causal decoder	51.0	55.2
Encoder-decoder	**51.3**	**60.6**

Encoder-decoder with MLM (T5-style) is the best after multitask finetuning

So which one to train?

Causal decoder with full LM (GPT2-style)

- Good for zero-shot, prompt-engineering, and generation

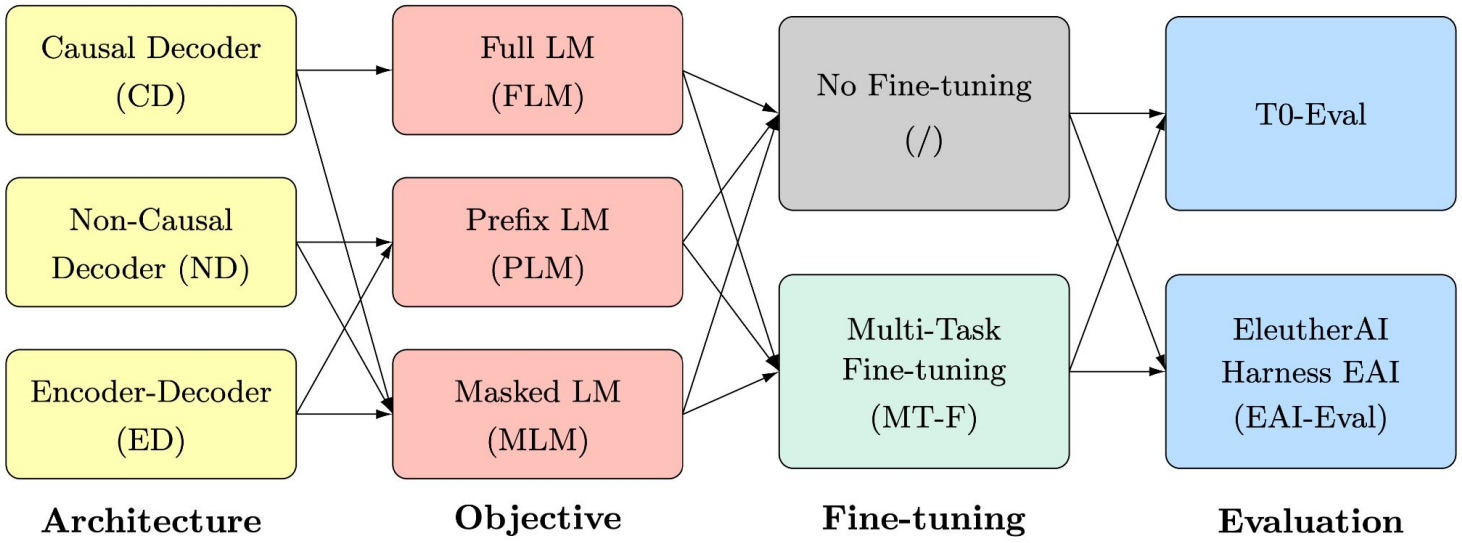
or Encoder-decoder with MLM (T5-style)

- Better zero-shot but only after supervised finetuning. Won't work for generation

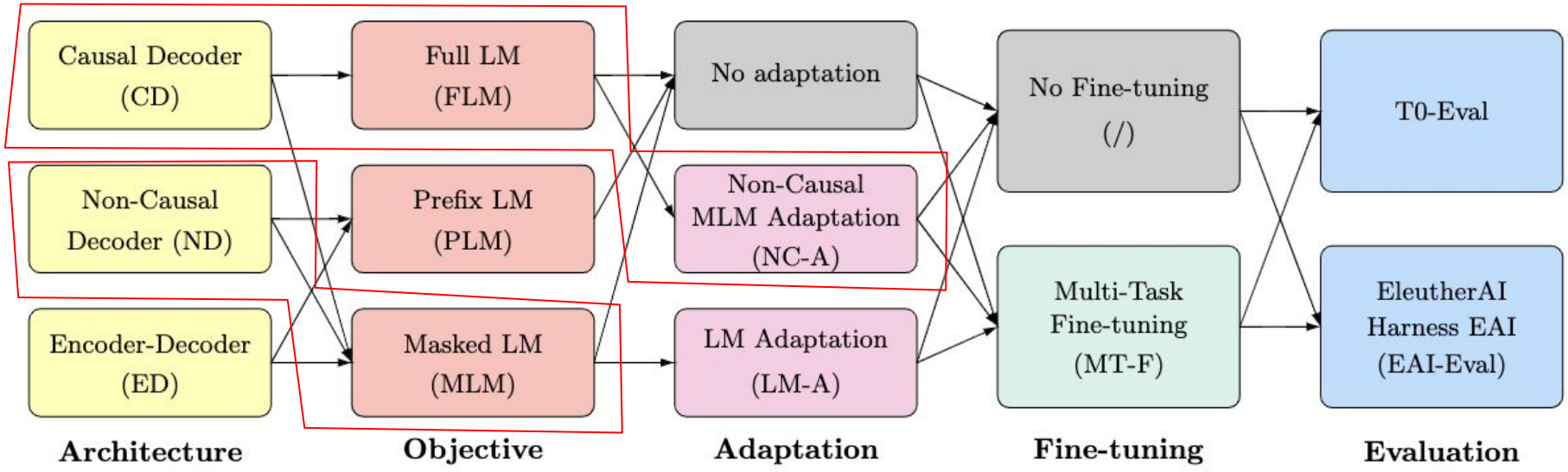
Can we find a compromise?

- Adaptation: pretrain on one objective then continue pretairning on another

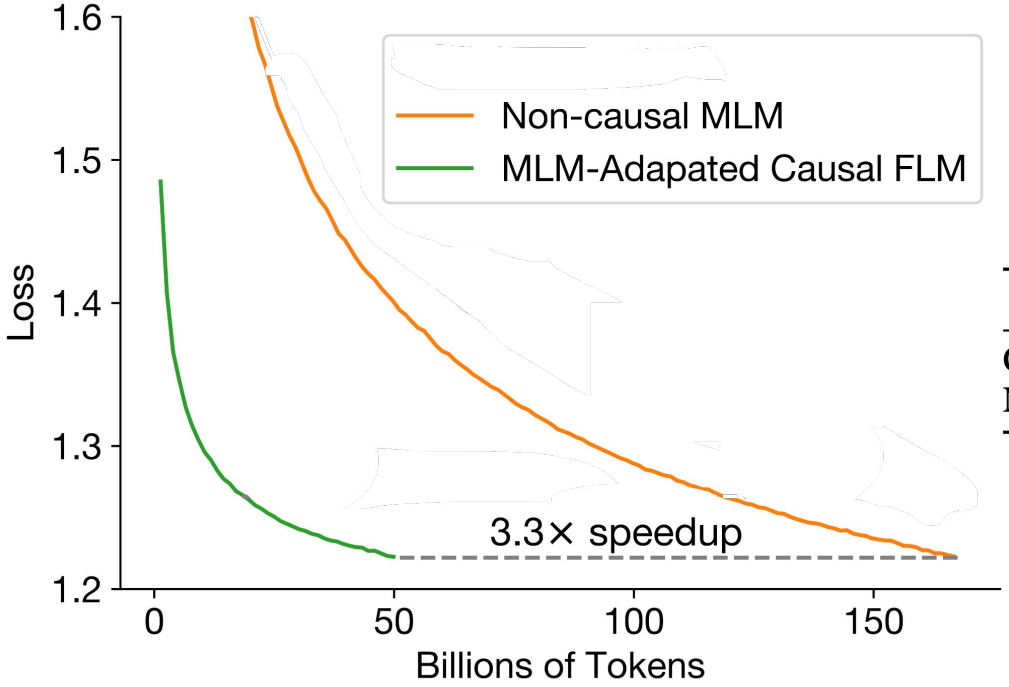
Language Model Adaptation



Language Model Adaptation



Language Model Adaptation



30% more compute for adaptation

	EAI-Eval	T0-Eval
Causal FLM	51.3	52.1
Non-causal MLM-Adapted	**52.3**	**54.9**

Conclusion and Future Work

Best for zero-shot: Causal decoder + full LM

Best for zero-shot after multitask finetuning: Encoder-decoder + MLM

Best compromise:

Causal decoder + full LM 70% of the compute, followed by

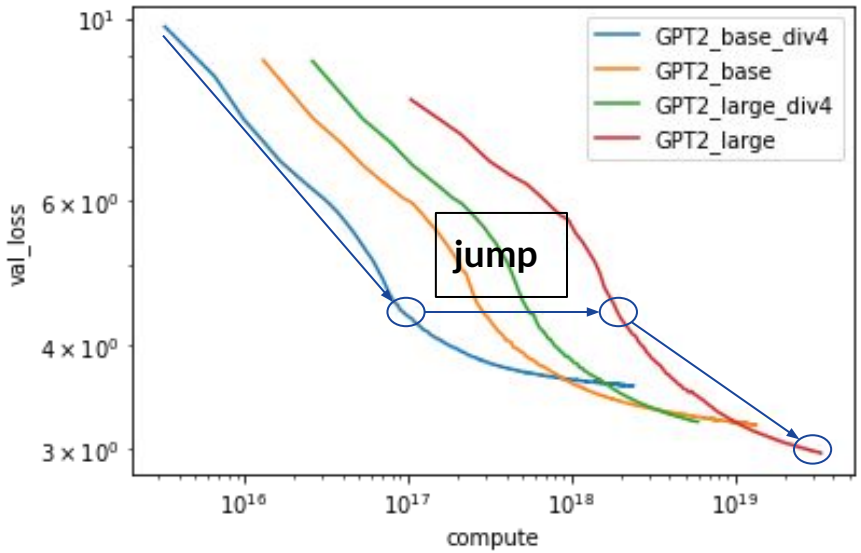
Non-causal decoder + MLM for 30% of the compute

Conclusion and Future Work

- Why best model changes after multi-task finetuning?
- What's an architecture and pretraining objective that would work well on both without an adaptation phase?
- We focused on zero-shot evaluation. Are the conclusions going to change if we use few-shot with/without finetuning?

Questions

Staged Training



What Language Model to Train if You Have One Million GPU Hours?

- Decoder-only model

- 70% on autoregressive LM, then

- 30% on MLM

Evaluation - Pretraining loss

		$GPT2_{LARGE}$			$GPT2_{BASE}$		
		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

Evaluation - Zero-shot downstream

		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

Experimental Setup

	Models architecture	
	Decoder-only	Encoder-decoder
Parameters	4.8B	11.0B
Vocabulary	32,128	
Positional embed.	T5 relative	
Embedding dim.	4,096	
Attention heads	64	
Feedforward dim.	10,240	
Activation	GEGLU [Shazeer, 2020]	
Layers	24	48
Logits via embed.	True	
Precision	`bfloat16`	

	Pretraining	Multitask finetuning
Dataset	C4	T0 train
Steps	131,072	10,000
Batch size in tokens	1,282,048	1,024
Optimizer	Adafactor(decay_rate=0.8)	
LR schedule	$\frac{1}{\sqrt{\max(n,10^4)}}$	fixed, 0.001
Dropout	0.0	0.1
z loss	0.0001	
Precision	`bfloat16`	