Sparse Models

Presenters: Vainateya Rangaraju, Vyas Narasimhan, Alex Ning

UNIVERSITY of VIRGINIA

Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

William Fedus, Barret Zoph, and Noam Shazeer

Introduction

- Large-scale training has been proven effective
  - Simple architectures with loads of computation can beat more complicated designs
- Previous work has scaled at a great computational cost
- Switch Transformer is a sparsely-activated mixture-of-expert model which aims to change this
  - Sparsity refers to activating only a subset of weights for each incoming token

Mixture-of-Experts Models

- Replace FFN in transformer model with many “expert” FFNs and a (trained) gating function to route tokens to various ones
- MoE models have seen success in machine translation
  - However, they have issues with complexity, communication cost, and training instabilities
- Current libraries and hardware accelerators cater to dense matrix operations
- This paper aims to simplify the MoE model to achieve stability and computational efficiency

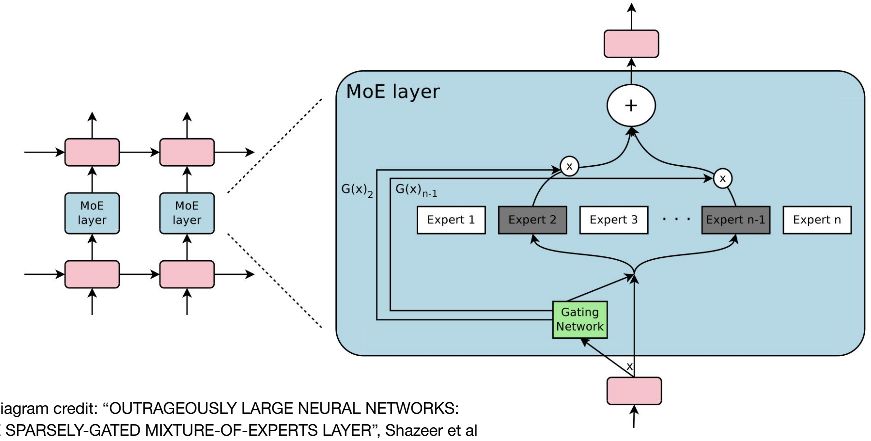


Diagram credit: “OUTRAGEOUSLY LARGE NEURAL NETWORKS: THE SPARSELY-GATED MIXTURE-OF-EXPERTS LAYER”, Shazeer et al

What We Will Show

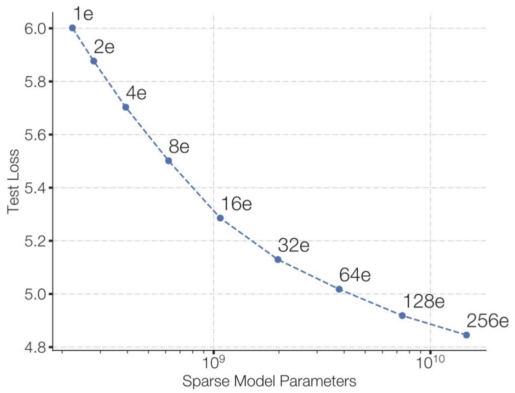
- This approach addresses previous MoE issues
- This approach is broadly valuable in NLP
- Superior scaling in:
    - Pre-training
    - Fine-tuning
    - Multi-task training
- Switch Transformer architecture is not only useful at scale but also with few computational resources (at small scale)
- Large, sparse model can be distilled by 99% and still preserve 30% of gains

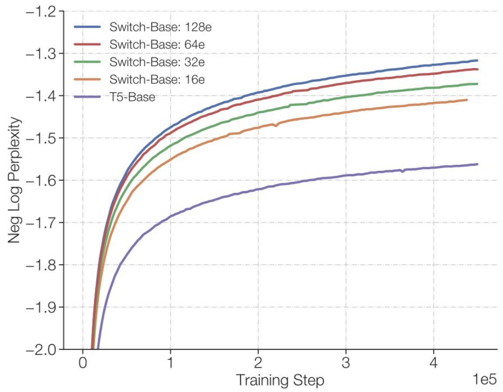
A Fourth Axis

Three traditional axes:

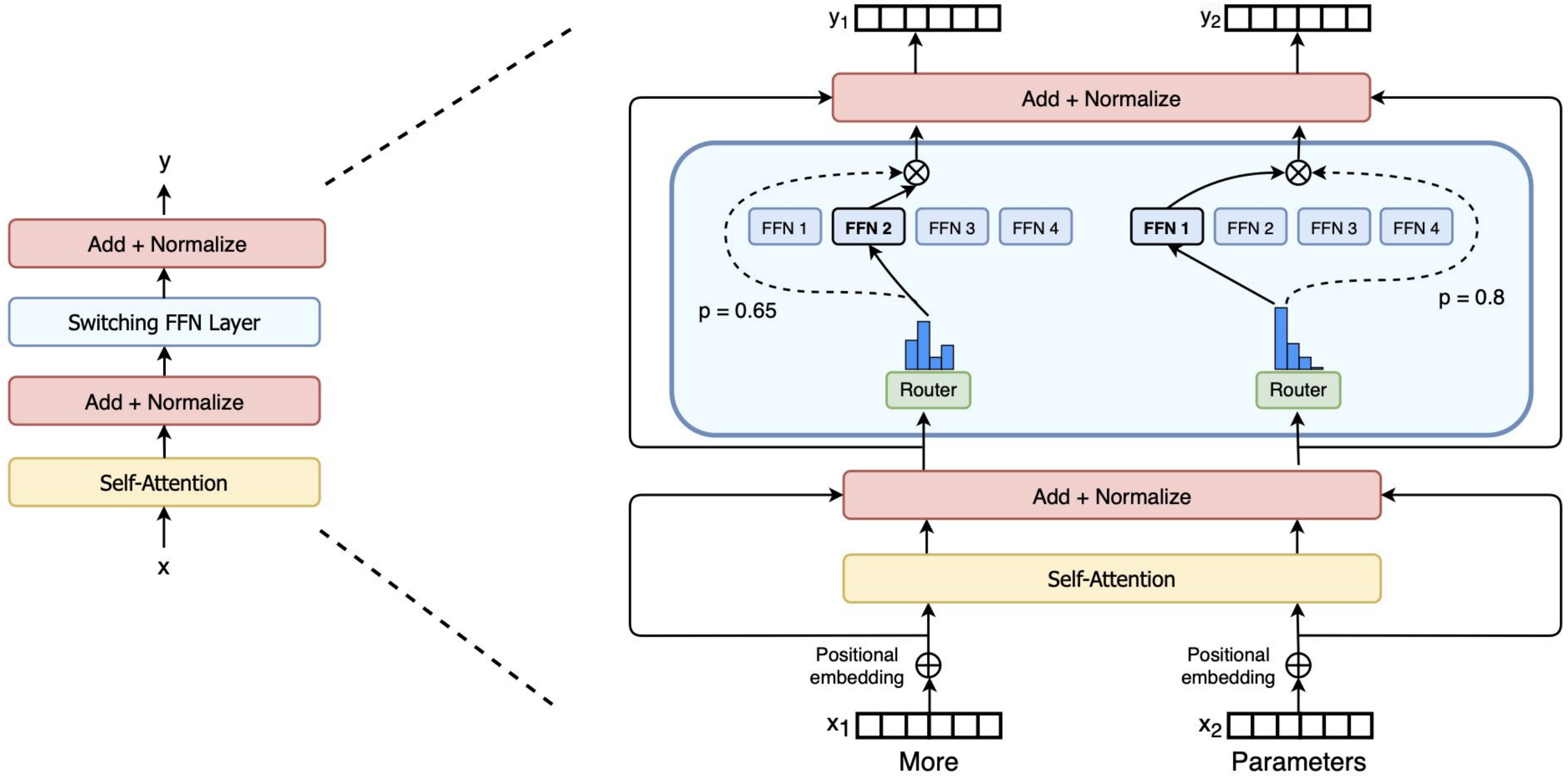
- Model size
- Dataset size
- Computational budget

Fourth axis: Parameter count independent from computational budget.





Switch Transformer Encoder Diagram



Top 1 Routing

- Previous work (Shazeer et al.) conjectured that the number of experts routed to must be more than 1 to have non-trivial gating in routing function
  - If you only route to top 1 expert, how is the routing function going to be able to compare experts and figure out how to use them?
- In stark contrast, this paper uses top 1 routing, and demonstrates that it:
  - Preserves model quality
  - Reduces routing computation
  - Performs better
- The top 1 routing function is also called “Switch layer”
- Computational benefits
  - Single expert reduces computational cost
  - Batch size (expert capacity) of each expert can be halved (further explained later)
  - Simpler implementation and less communication cost

Distributed Switch Implementation

- One unique expert per device
- Tensor shapes statically determined at compilation time
- Computation is *dynamic* due to routing
- Ideally, routing function would route to experts evenly. This is not always true
- Expert capacity: Limit for number of tokens computed by each expert

$$\text{expert capacity} = \left(\frac{\text{tokens per batch}}{\text{number of experts}}\right) \times \text{capacity factor}.$$

- Capacity factor (>1) provides extra buffer/overhead in-case tokens are not evenly spread between experts

Expert Capacity

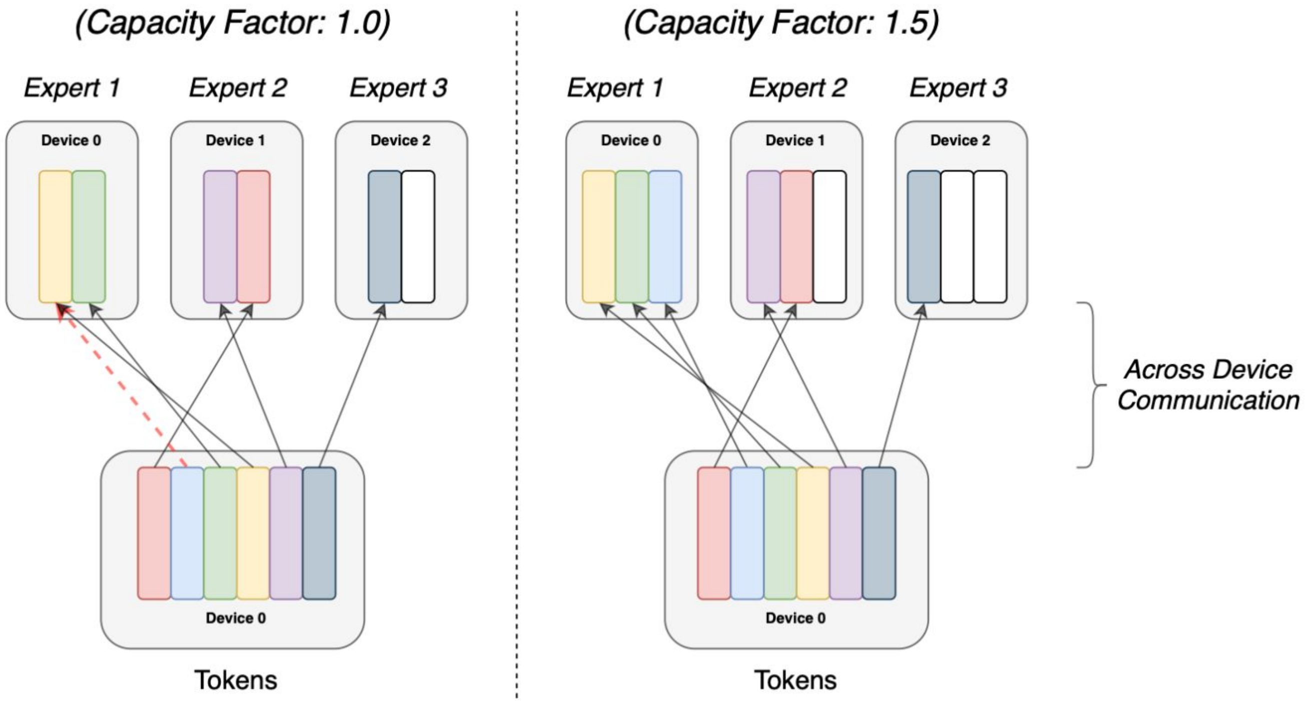
- If too many tokens are routed to an expert, computation is skipped and token is passed through skip connection
    - These tokens are called “dropped tokens”
- Higher capacity factor can prevent dropped tokens, but wastes computation and memory (especially due to statically declared sizes)
- Empirically, lower rates of dropped tokens is important for scaling
- No dependency between number of experts and number of tokens dropped

$$\text{expert capacity} = \left(\frac{\text{tokens per batch}}{\text{number of experts}}\right) \times \text{capacity factor}.$$

Token Routing Dynamics

Terminology

- **Experts:** Split across devices, each having their own unique parameters. Perform standard feed-forward computation.
- **Expert Capacity:** Batch size of each expert. Calculated as
- (tokens_per_batch / num_experts) * capacity_factor
- **Capacity Factor:** Used when calculating expert capacity. Expert capacity allows more buffer to help mitigate token overflow during routing.



Differentiable Load Balancing Loss

- Auxiliary loss added to total loss
- Promote evenly allocating tokens between experts
- Given N experts, a batch $\mathcal{B}$ with T tokens
- Ideally, both f_i and $P_i = 1/N$

$$\text{loss} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i \cdot P_i$$

$$f_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{\operatorname{argmax} p(x) = i\}$$

$$P_i = \frac{1}{T} \sum_{x \in \mathcal{B}} p_i(x).$$

f_i = fraction of tokens allocated to ith expert

P_i = fraction of router probability allocated to ith expert

Differentiable Load Balancing Loss - Ideally

$$loss = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i \cdot P_i = \alpha \cdot N \cdot \sum_{i=1}^{N} \left(\frac{1}{N} \cdot \frac{1}{N} \right)$$

$$= \alpha \cdot N \cdot \sum_{i=1}^{N} \frac{1}{N^2} = \alpha \cdot N \cdot \frac{1}{N} = \alpha$$

Empirically, $\alpha = 10^{-2}$ has worked well

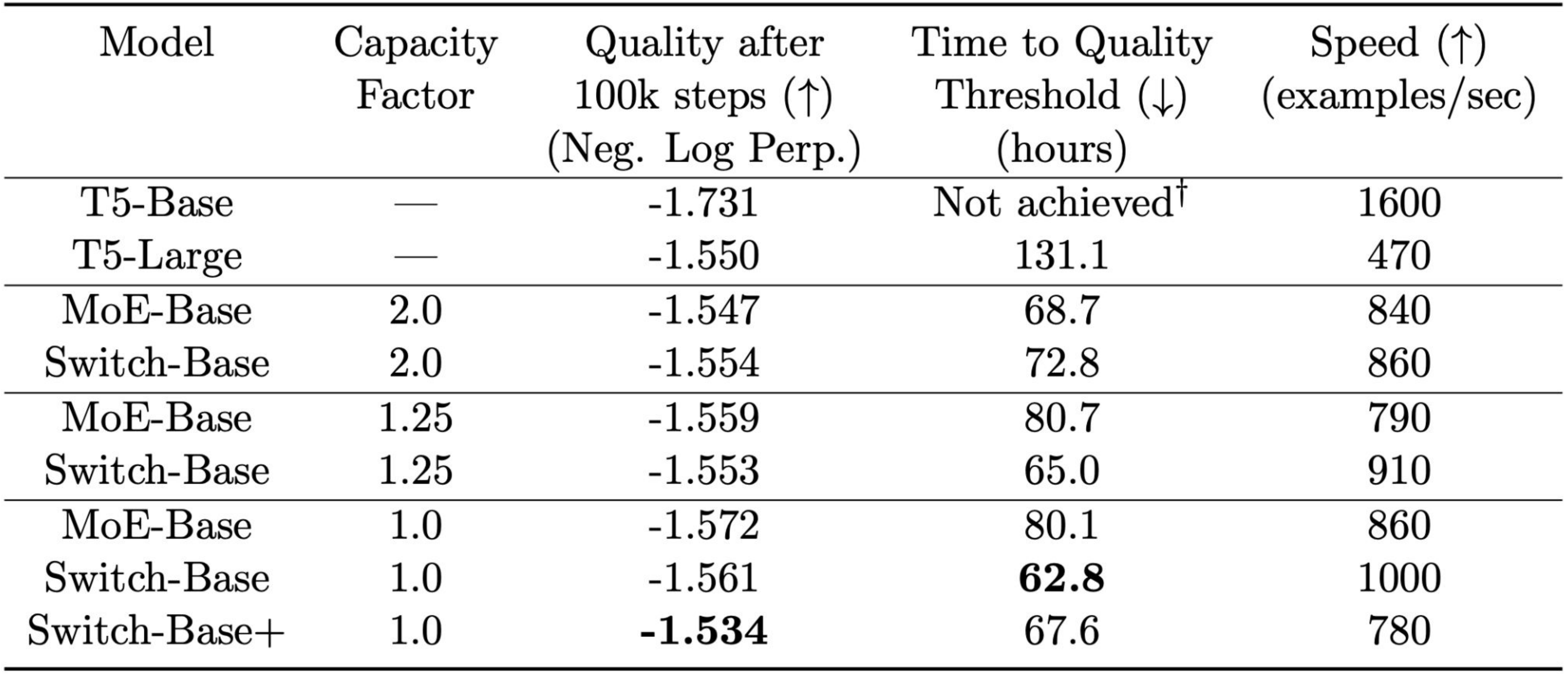
Putting It All Together: The Switch Transformer

- Compared with regular model (T5 base) and MoE Transformer (top 2 routing)
  - All models trained identically

Switch transformers outperform:

1. Speed quality: given fixed computations and amount of time
2. Smaller computational footprint than MoE Transformer
   a. If the size of Switch Transformer is increased to match training speed of MoE Transformer, then Switch Transformer outperforms MoE Transformer

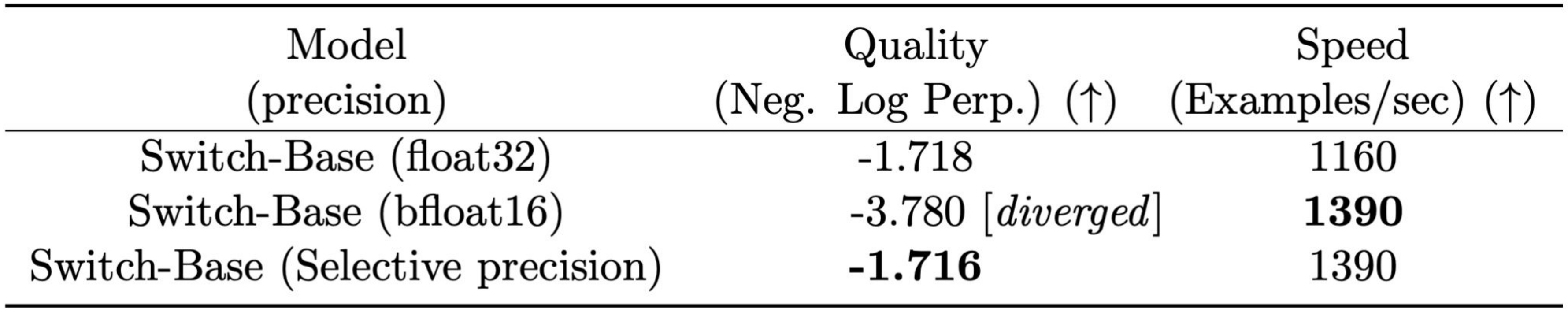
Capacity Factor: Switch Transformer vs MoE Transformer



Model	Capacity Factor	Quality after 100k steps (↑) (Neg. Log Perp.)	Time to Quality Threshold (↓) (hours)	Speed (↑) (examples/sec)
T5-Base	—	-1.731	Not achieved[†]	1600
T5-Large	—	-1.550	131.1	470
MoE-Base	2.0	-1.547	68.7	840
Switch-Base	2.0	-1.554	72.8	860
MoE-Base	1.25	-1.559	80.7	790
Switch-Base	1.25	-1.553	65.0	910
MoE-Base	1.0	-1.572	80.1	860
Switch-Base	1.0	-1.561	**62.8**	1000
Switch-Base+	1.0	**-1.534**	67.6	780

Selective Precision

- float32 precision is used within routing function; bfloat16 elsewhere
  - When data enters the routing function, it is cast to float32
  - When it leaves, it is cast back to bfloat16



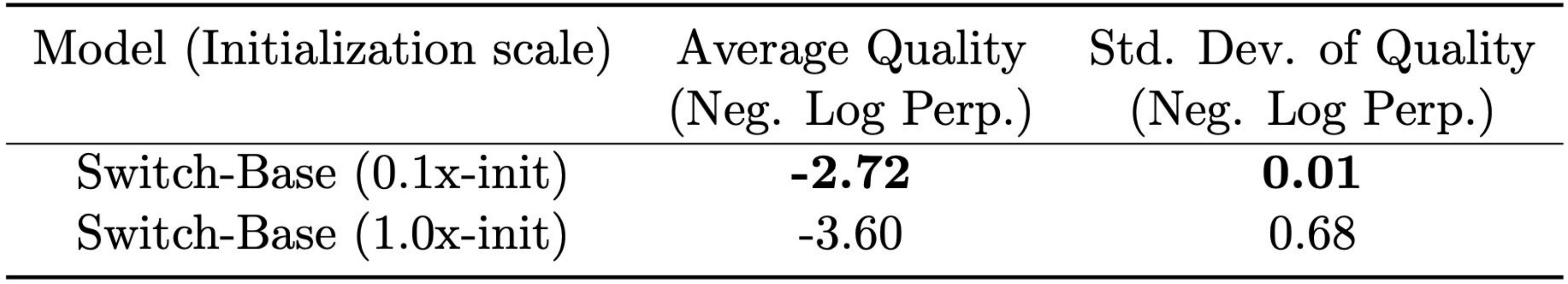
Model (precision)	Quality (Neg. Log Perp.) (↑)	Speed (Examples/sec) (↑)
Switch-Base (float32)	-1.718	1160
Switch-Base (bfloat16)	-3.780 [*diverged*]	**1390**
Switch-Base (Selective precision)	**-1.716**	1390

Small Parameter Initialization for Stability

Weights are initialized with $\mu = 0$ and $\sigma = \sqrt{s/n}$ where s is a scale hyper-parameter and n is the number of input units to the weight tensor.

Typically for transformers $s = 1.0$

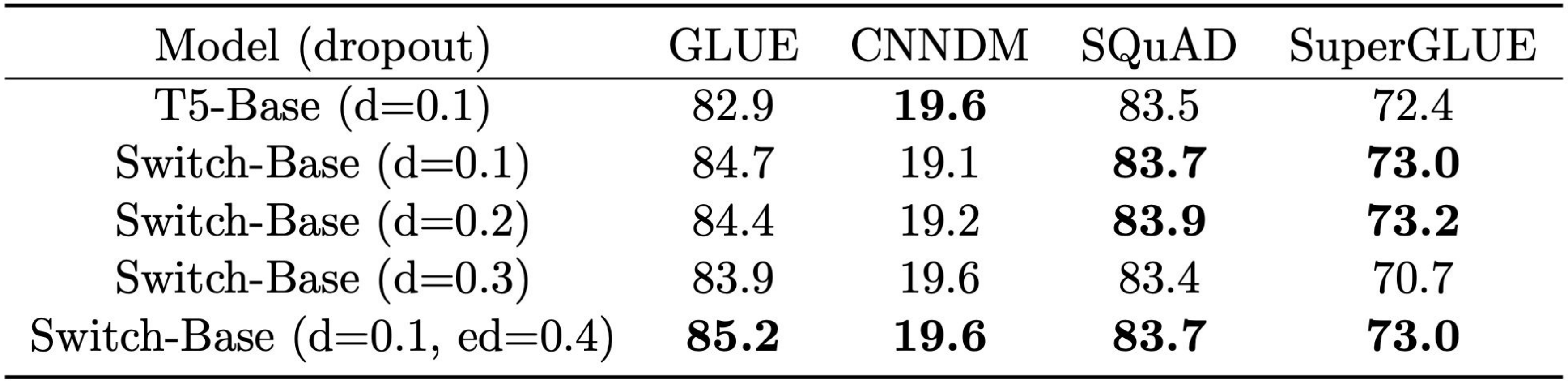
However, we find that $s = 0.1$ greatly improves stability. This is true for both a 223M parameter model and a model with > 1T parameters



Model (Initialization scale)	Average Quality (Neg. Log Perp.)	Std. Dev. of Quality (Neg. Log Perp.)
Switch-Base (0.1x-init)	**-2.72**	**0.01**
Switch-Base (1.0x-init)	-3.60	0.68

Regularizing Large Sparse Models

- Overfitting is a greater concern given more parameters of Switch Transformer
- Thus, we implement a higher dropout rate specifically for the expert layers
  - *ed = expert dropout*



Model (dropout)	GLUE	CNNDM	SQuAD	SuperGLUE
T5-Base (d=0.1)	82.9	**19.6**	83.5	72.4
Switch-Base (d=0.1)	84.7	19.1	**83.7**	**73.0**
Switch-Base (d=0.2)	84.4	19.2	**83.9**	**73.2**
Switch-Base (d=0.3)	83.9	19.6	83.4	70.7
Switch-Base (d=0.1, ed=0.4)	**85.2**	**19.6**	**83.7**	**73.0**

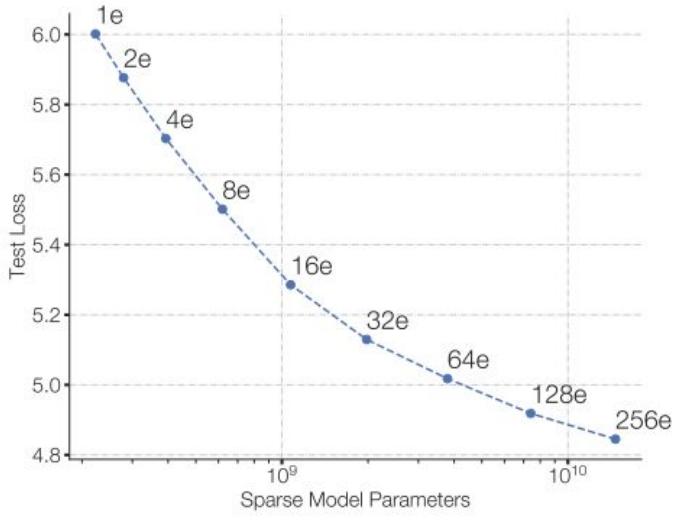
Scaling

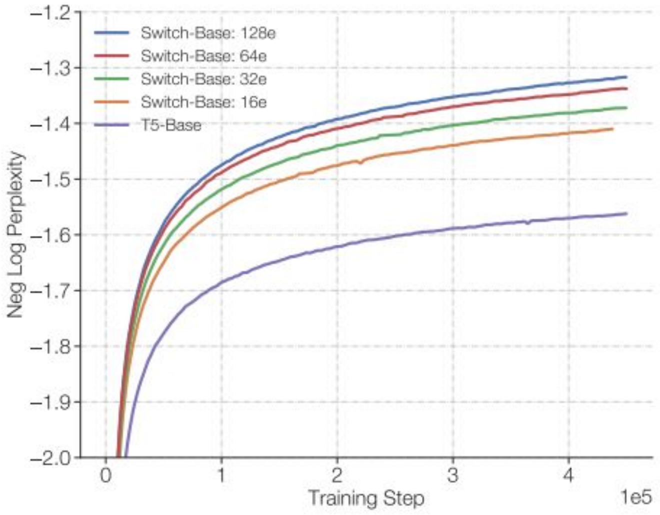
- The number of experts were scaled with a fixed computational budget
- Adding additional experts marginally increases the compute
    - The router now evaluates over more experts

Fixed number of steps

Overall trend: having more experts speeds up training

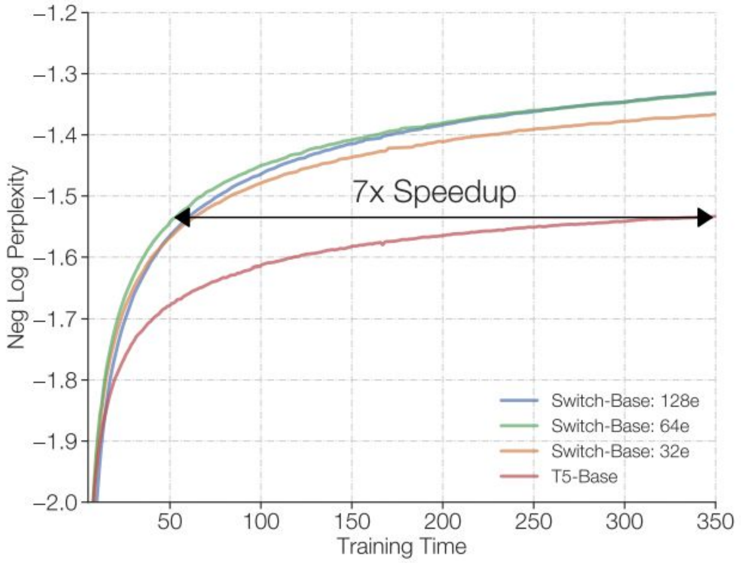
- More experts correlate with smaller loss and more sample-efficient models





Fixed training duration

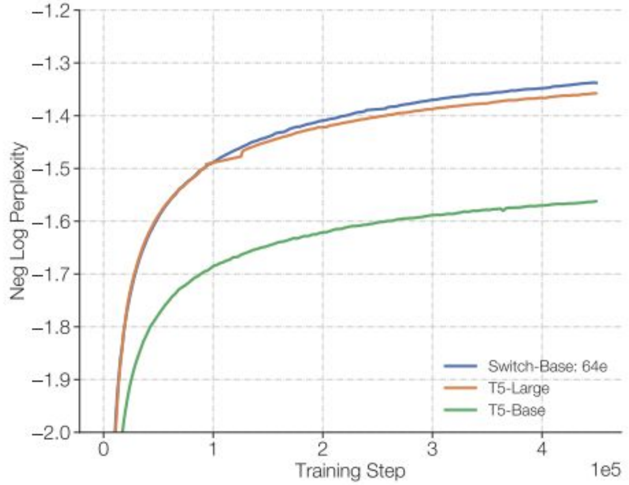
Sparse models achieve the same quality as dense models with the same computational budget in **1/7** of the time

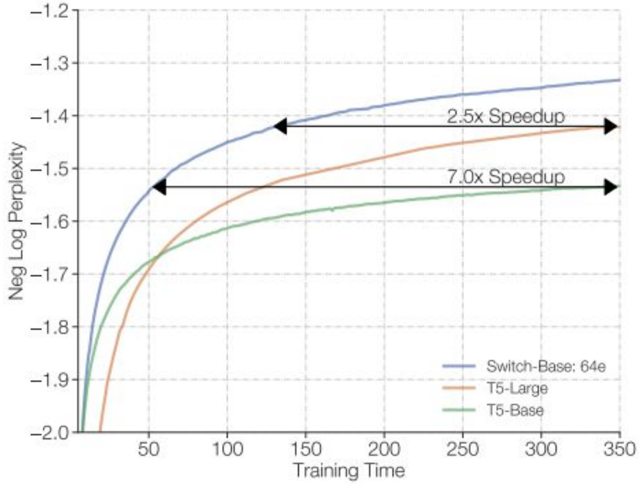


Compared with larger dense models?

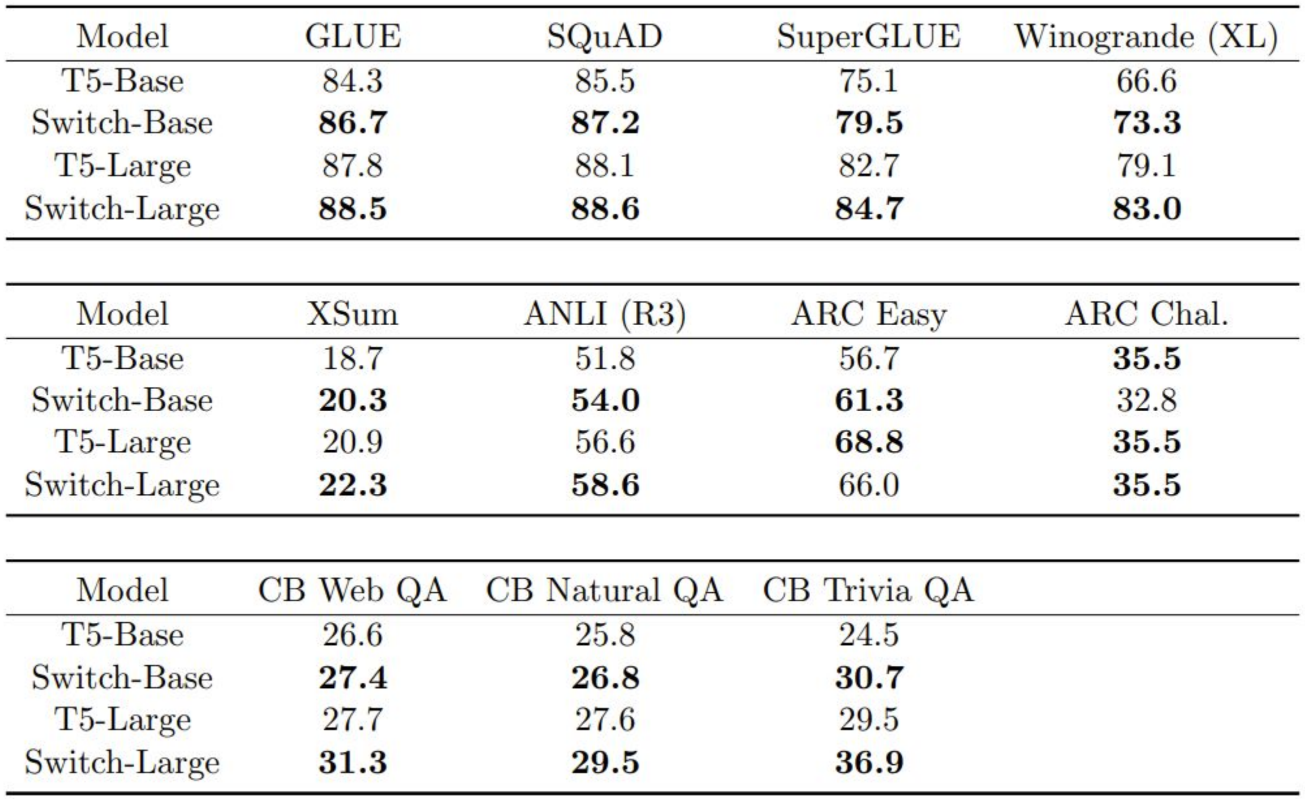
Switch-Base still outperforms larger, more computationally expensive models

- T5-Large applies 3.5x more FLOPS per token
- Switch-Base takes fewer training steps and less training time overall





Fine-Tuning



Model	GLUE	SQuAD	SuperGLUE	Winogrande (XL)
T5-Base	84.3	85.5	75.1	66.6
Switch-Base	**86.7**	**87.2**	**79.5**	**73.3**
T5-Large	87.8	88.1	82.7	79.1
Switch-Large	**88.5**	**88.6**	**84.7**	**83.0**

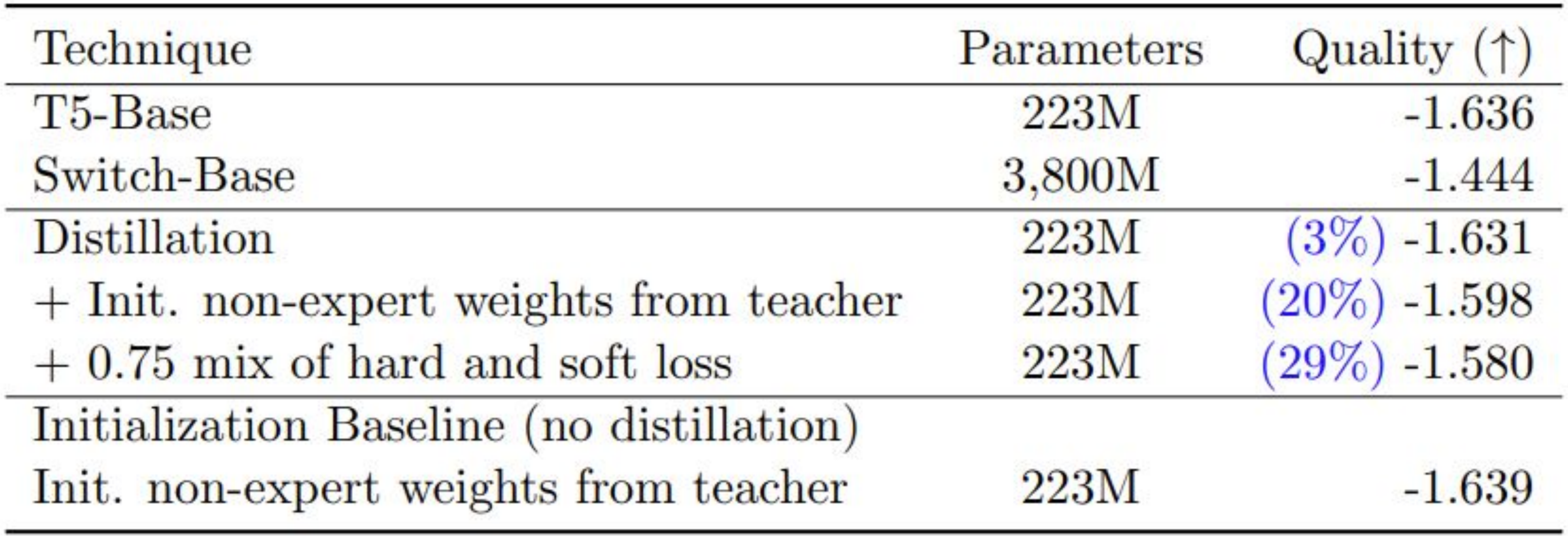
Model	XSum	ANLI (R3)	ARC Easy	ARC Chal.
T5-Base	18.7	51.8	56.7	**35.5**
Switch-Base	**20.3**	**54.0**	**61.3**	32.8
T5-Large	20.9	56.6	**68.8**	**35.5**
Switch-Large	**22.3**	**58.6**	66.0	**35.5**

Model	CB Web QA	CB Natural QA	CB Trivia QA
T5-Base	26.6	25.8	24.5
Switch-Base	**27.4**	**26.8**	**30.7**
T5-Large	27.7	27.6	29.5
Switch-Large	**31.3**	**29.5**	**36.9**

Distillation

Can we distill a large Switch model into a smaller T5-model?

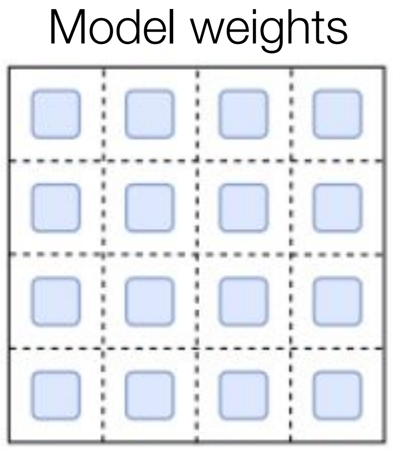
- Initialize T5 with non-expert weights from Switch
- Weight teacher’s output at .25 and ground truth label at .75

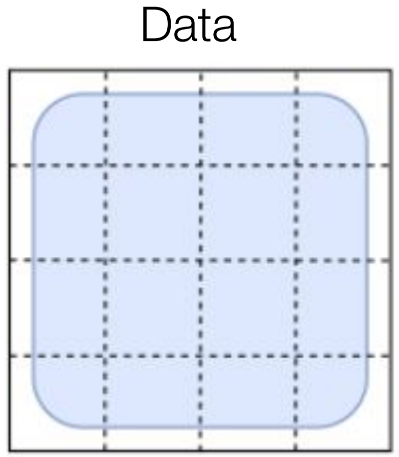


Technique	Parameters	Quality (↑)
T5-Base	223M	-1.636
Switch-Base	3,800M	-1.444
Distillation	223M	(3%) -1.631
+ Init. non-expert weights from teacher	223M	(20%) -1.598
+ 0.75 mix of hard and soft loss	223M	(29%) -1.580
Initialization Baseline (no distillation)		
Init. non-expert weights from teacher	223M	-1.639

Data Parallelism

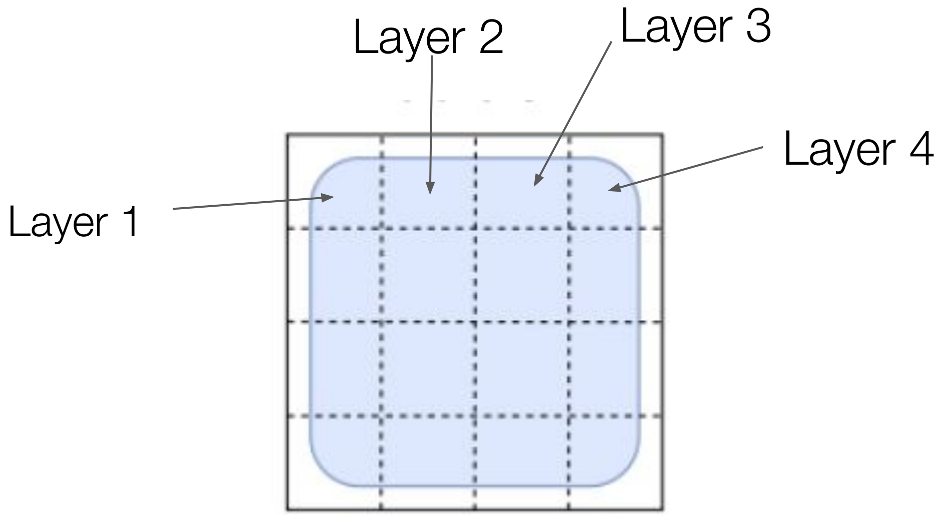
- Generally the default
- Each core has the same weights
- Training data is shared across each core
- Result is concatenated at the end

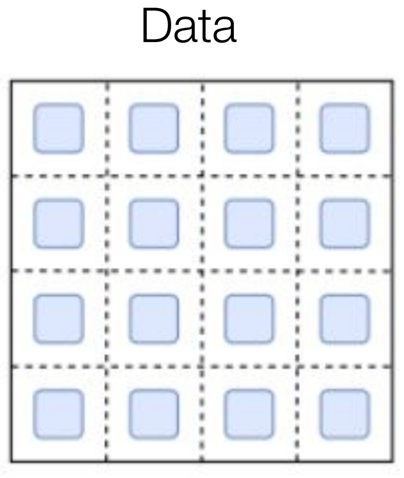




Model Parallelism

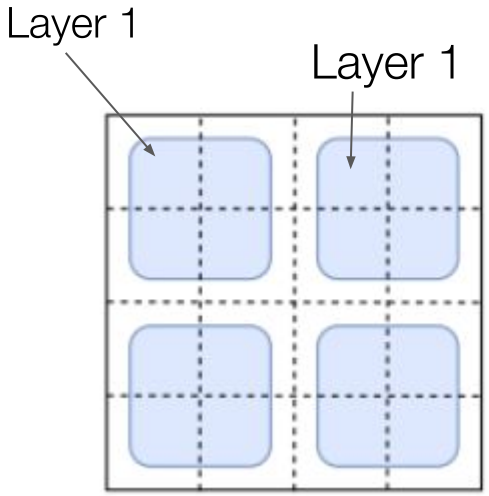
- The model weights are split up amongst the cores
- Data is sent sequentially through each core
  - Each core needs to communicate during forward and backpropagation
- Inefficient: want to avoid!

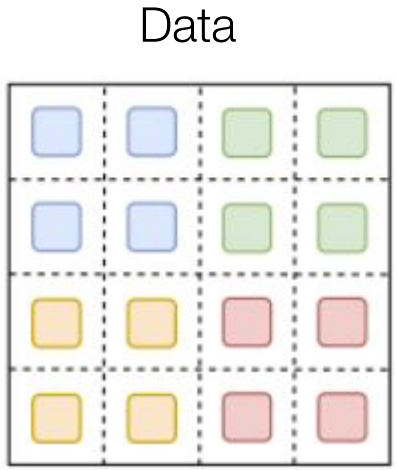




Model and Data Parallelism

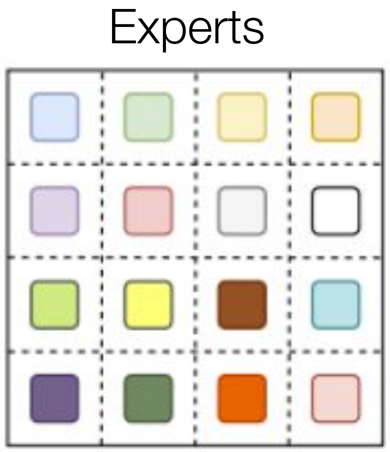
- Split up the model weights across 4 cores, and copy them to the remaining 12
- Data batch is split into 4 smaller batches: each batch fed to a different copy

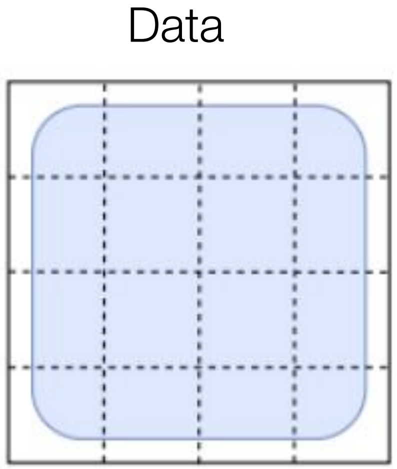




Expert and Data Parallelism

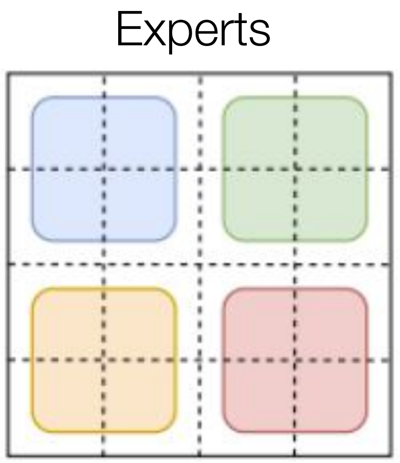
- Used by Switch Transformer
- Each expert is given to a different core
- Data is then split up across the 16 cores - each expert is given distinct data

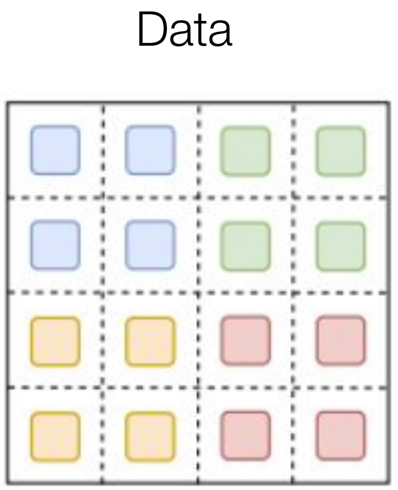




Expert, Model, and Data Parallelism

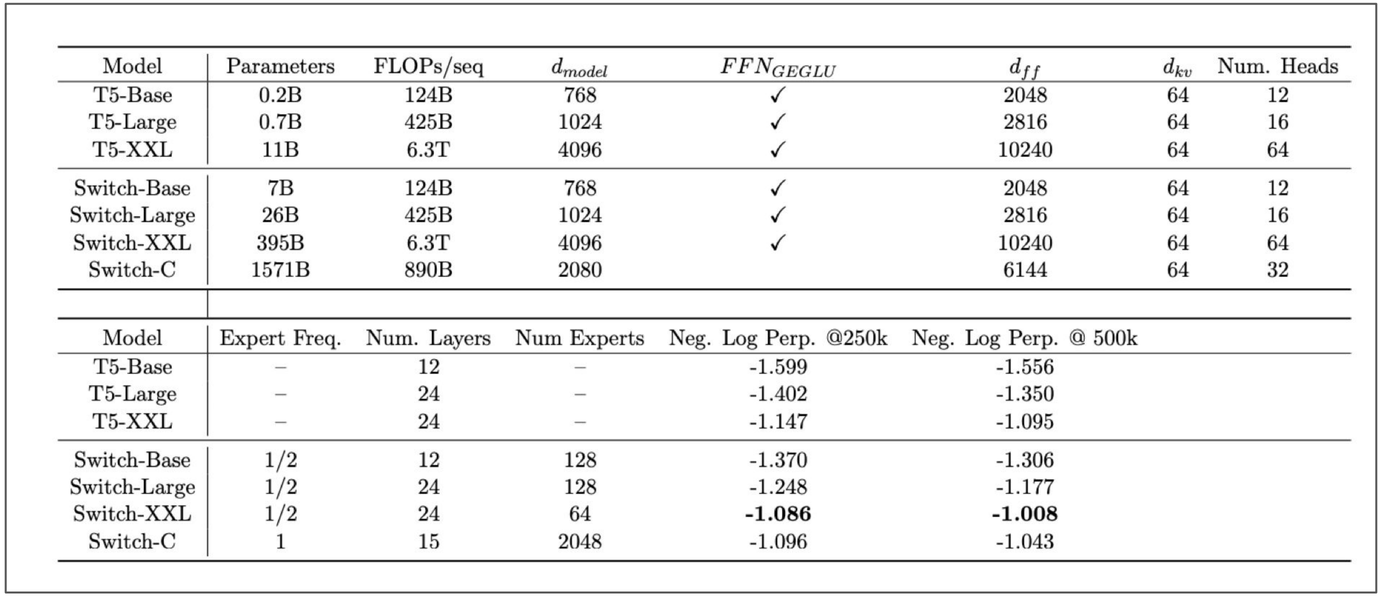
- Extreme case where one expert doesn't fit in a core
  - Switch Transformer with 1 trillion parameters
- Each expert is split up across multiple cores
- Data is split up accordingly - each expert gets different data





Towards a Trillion Parameter Models

- Two large switch transformers (365 billion, 1.6 trillion)
  - Switch-C model is designed with only expert-parallelism (hyperparameters smaller than T5-XXL)
  - Switch-XXL model FLOP matched to T5-XXL → larger hyperparameters with communication cost



Model	Parameters	FLOPs/seq	d_{model}	FFN_{GEGLU}	d_{ff}	d_{kv}	Num. Heads
T5-Base	0.2B	124B	768	✓	2048	64	12
T5-Large	0.7B	425B	1024	✓	2816	64	16
T5-XXL	11B	6.3T	4096	✓	10240	64	64
Switch-Base	7B	124B	768	✓	2048	64	12
Switch-Large	26B	425B	1024	✓	2816	64	16
Switch-XXL	395B	6.3T	4096	✓	10240	64	64
Switch-C	1571B	890B	2080		6144	64	32

Model	Expert Freq.	Num. Layers	Num Experts	Neg. Log Perp. @250k	Neg. Log Perp. @ 500k
T5-Base	–	12	–	-1.599	-1.556
T5-Large	–	24	–	-1.402	-1.350
T5-XXL	–	24	–	-1.147	-1.095
Switch-Base	1/2	12	128	-1.370	-1.306
Switch-Large	1/2	24	128	-1.248	-1.177
Switch-XXL	1/2	24	64	**-1.086**	**-1.008**
Switch-C	1	15	2048	-1.096	-1.043

Towards a Trillion Parameter Models

- Sample efficiency vs T5-XXL
    - We consider negative log perplexity on C4 after 250 and 500 steps
    - Both Switch Transformers improve over T5-XXL by over 0.061
    - To put that in perspective: T5-XXL trained for an additional 250k steps to increase 0.052

- Training Instability
    - We see inconsistent results as we increase the scale of our models
    - Larger Switch-C model shows no training instability but the Switch-XXL version (10x FLOPs per sequence) is sometimes unstable

Towards a Trillion Parameter Models

- Reasoning Fine Tuning Performance
  - Switch-XXL, pre-trained on 503B tokens, w/ multi-task training for efficiency, learning all tasks jointly.
  - SQuAD: Achieved 89.7 accuracy, close to SOTA of 91.3.
  - SuperGLUE Score: Recorded an average of 87.5, slightly below T5's 89.3 and SOTA of 90.0.
  - ANLI Achievement: Improved to 65.7 accuracy, surpassing the previous best of 49.4.
  - Observation: Despite leading in Neg. Log Perp. on pre-training, Switch-XXL's downstream performance hasn't fully reached state-of-the-art levels.

Towards a Trillion Parameter Models

- Knowledge-Based Fine Tuning Performance
  (Early tests on closed-book tasks w/o additional pre-training.)
  - Natural Questions: Reached 34.4 exact match, surpassing the previous 32.8.
  - WebQuestions: Improved to 41.0, over the earlier 37.2.
  - TriviaQA: Increased to 47.5, above the former 42.9.
  - Model Efficiency: Achieved high quality with less than half the training data of counterparts.
  - Task Suitability: Better translation of gains to knowledge-based tasks than reasoning tasks.
  - Future Potential: Pre-training perplexity suggests room for further fine-tuning enhancements.

Related Work

Scaling Neural Networks:

- Model parallelism has been shown to scale models to billions of parameters
  - Pipeline based model parallelism
- Product Key networks

Conditional Computation:

- Early methods adapted weight selection from hidden-state bit patterns.
- Expert layers demonstrated improvements in tasks like jittered MNIST and speech.

Related Work

Mixture of Experts:

- MOEs between LSTM layers → SOTA on language modeling and machine translation benchmarks
- Integration in Transformer where MOEs were subbed for feed forward layers (no NLP results)
- GShard, used MOE transformers to achieve large-scale machine translation successes
- Fan et al.'s (2021) deterministic MoE for parameter segmentation among language groups.

Attention Pattern Sparsity:

- Techniques for reducing attention complexity from $O(L^2)$, facilitating longer sequence learning
  - Where L is the sequence length dimension
- Identified as complementary to current models, with potential for future integration.

Discussion

Q1	**Does the Switch Transformer perform better just because it has more parameters?**	● While larger models do perform better, the Switch Transformer also increases sample efficiency and speed using equivalent computational resources.
Q2	**Can I benefit from this model without supercomputer access?**	● Yes, even models with as few as two experts see performance gains and can run on commonly available GPUs or TPUs.
Q3	**Do sparse models have an edge over dense models on the speed-accuracy Pareto curve?**	● Controlled experiments confirm that sparse models outperform dense models for the same computational budget and time.

Discussion

Q3

Is it possible to deploy smaller versions of these large models?

- Quality is not fully preserved, but we can distill sparse models into dense ones, retaining about 30% of the quality gain with 10 to 100x compression

Q4

Why opt for a Switch Transformer over a model-parallel dense model?

- Switch Transformers are more time-efficient and can still employ model parallelism to balance FLOPs per token with conventional parallelism trade-offs.

Q5

If sparse models are so efficient, why haven't they been widely adopted?

- The Switch Transformer tackles the hurdles of complexity, training, and communication in sparse models, which held back sparse models amidst the dense model scaling trend.

Limitations + Future Work

- Addressing training stability for extremely large models like Switch-XXL remains a challenge despite some progress with stability techniques and regularizers
- Uncovered anomalies in performance despite similar pre-training results highlight a complex relationship between fine-tuning quality, FLOPs per token, and parameter count
- Advocates for a detailed analysis of scaling relationships to better inform the design of future architectures and hardware, considering data, model, and expert parallelism

Limitations + Future Work

- Suggests exploring systems with diverse expert types for more adaptable computation, enhancing performance especially for more complex tasks

- Proposes investigating the integration of expert layers outside the traditional FFN layer of Transformers, noting potential improvements and challenges like training instabilities

- Encourages examining the benefits of model sparsity in various modalities beyond language, suggesting potential advantages in multi-modal networks

Longformer: The Long-Document Transformer

Iz Beltagy, Matthew E. Peters, and Arman Cohan

Motivation

Transformers take a fixed-size input

- 512-2048 tokens for most LLMs

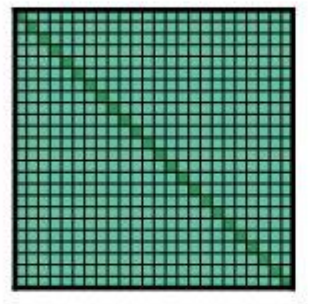
For long documents, they have to split up the text and do self-attention on each part

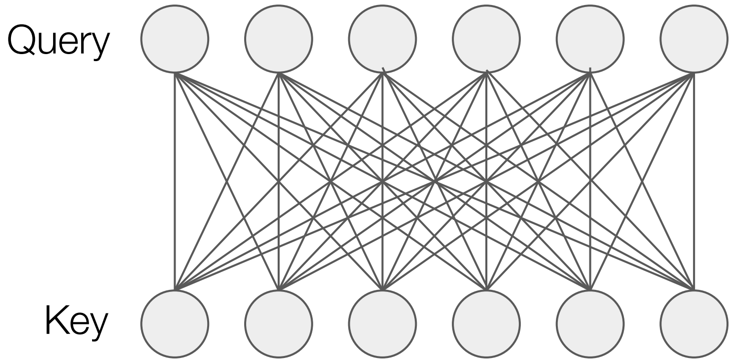
- Prevents tokens from the beginning and end of a document from attending to one another

Previous Solution (naive)

- Why not just increase the input size to allow for more tokens to attend to each other?
  - This is highly inefficient!

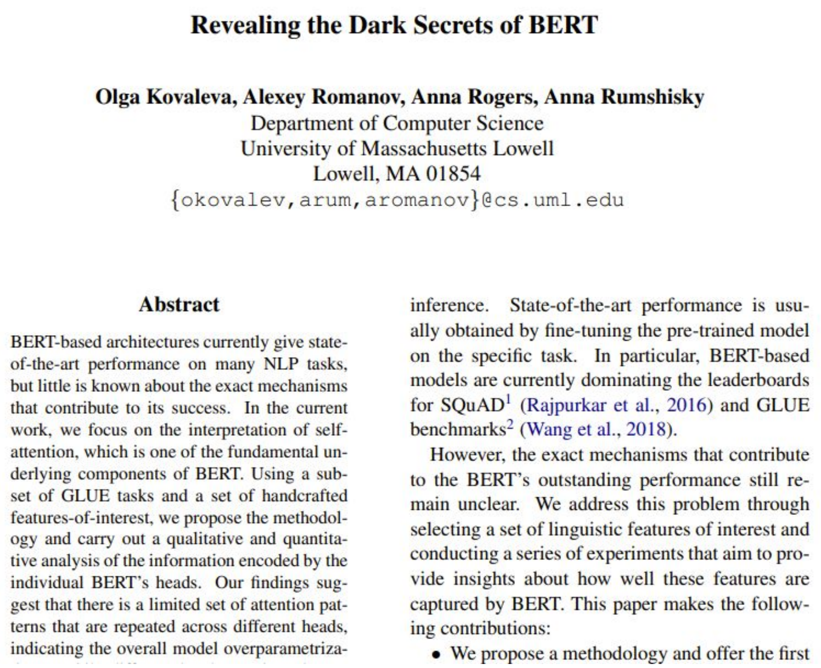
- Memory complexity of self-attention is O(n^2)
  - As input size increases, memory usage increases **quadratically** - not good!


(a) Full n^2 attention



Revealing the Dark Secrets of BERT

- Paper written by Kovaleva et al., in 2019
- Conducted an intensive study on self-attention in BERT and how information was encoded in each head
- **Discussed the importance of local context in sentences**



Revealing the Dark Secrets of BERT

Olga Kovaleva, Alexey Romanov, Anna Rogers, Anna Rumshisky
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854
{okovalev,arum,aromanov}@cs.uml.edu

Abstract

BERT-based architectures currently give state-of-the-art performance on many NLP tasks, but little is known about the exact mechanisms that contribute to its success. In the current work, we focus on the interpretation of self-attention, which is one of the fundamental underlying components of BERT. Using a subset of GLUE tasks and a set of handcrafted features-of-interest, we propose the methodology and carry out a qualitative and quantitative analysis of the information encoded by the individual BERT's heads. Our findings suggest that there is a limited set of attention patterns that are repeated across different heads, indicating the overall model overparametriza-

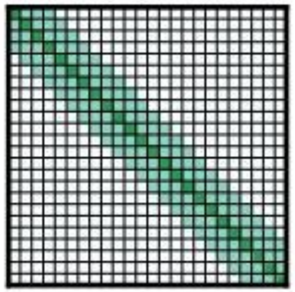
inference. State-of-the-art performance is usually obtained by fine-tuning the pre-trained model on the specific task. In particular, BERT-based models are currently dominating the leaderboards for SQuAD[1] (Rajpurkar et al., 2016) and GLUE benchmarks[2] (Wang et al., 2018).

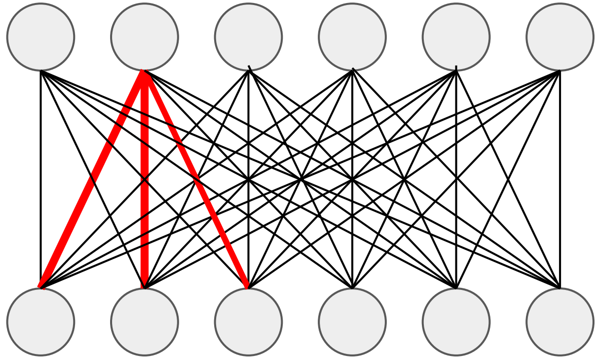
However, the exact mechanisms that contribute to the BERT's outstanding performance still remain unclear. We address this problem through selecting a set of linguistic features of interest and conducting a series of experiments that aim to provide insights about how well these features are captured by BERT. This paper makes the following contributions:

- We propose a methodology and offer the first

Sliding window attention

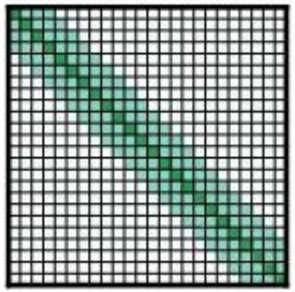
- Convolutional kernel: do attention on a few tokens at a time and slide the window over
- Memory becomes O(n*k) for a kernel size of k


(b) Sliding window attention

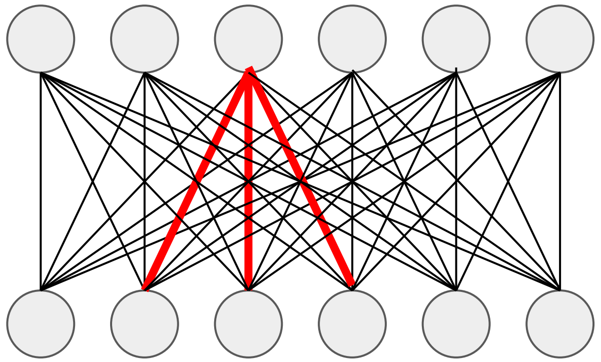


Sliding window attention

- Convolutional kernel: do attention on a few tokens at a time and slide the window over
- Memory becomes O(n*k) for a kernel size of k

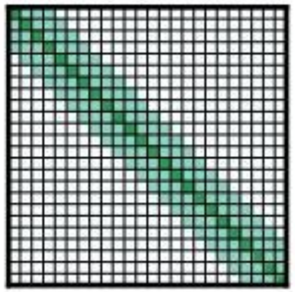


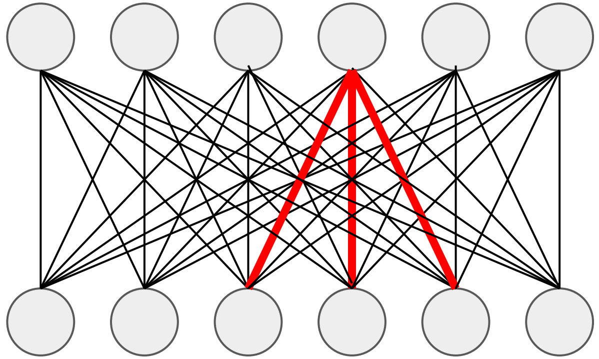
(b) Sliding window attention



Sliding window attention

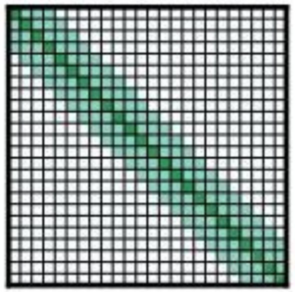
- Convolutional kernel: do attention on a few tokens at a time and slide the window over
- Memory becomes O(n*k) for a kernel size of k

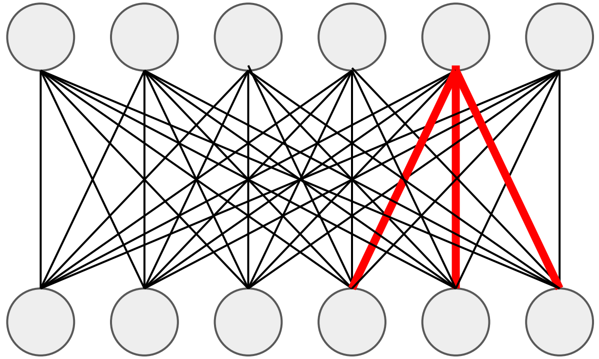

(b) Sliding window attention



Sliding window attention

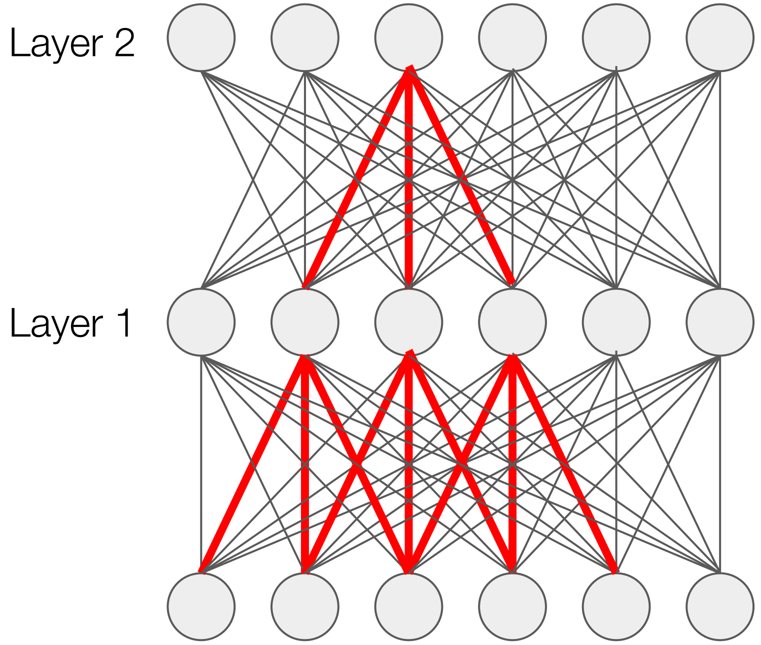
- Convolutional kernel: do attention on a few tokens at a time and slide the window over
- Memory becomes O(n*k) for a kernel size of k


(b) Sliding window attention



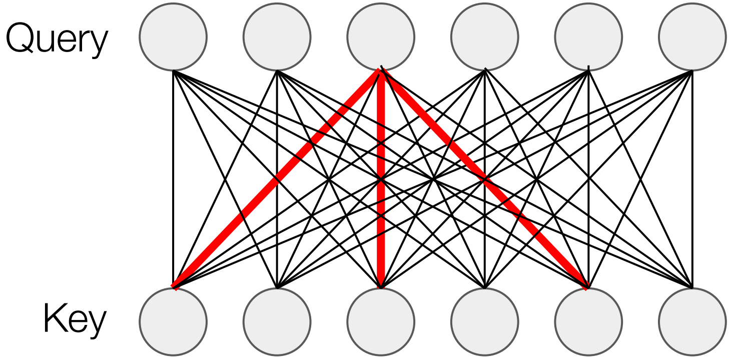
Sliding window attention

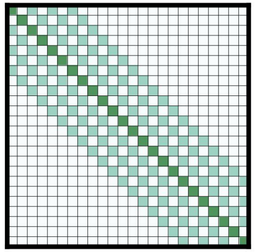
- Convolutional kernel: do attention on a few tokens at a time and slide the window over
- Memory becomes O(n*k) for a kernel size of k
- Obvious cost in performance is minimized by stacking attention layers
    - Tokens in higher layers will eventually attend to further tokens, forming a cone



Dilated sliding window

- Still maintain the same number of elements in the window, but alternate
- Requires fewer layers to cover an entire sequence
- Contradicts previous assertion about local context
- Used along with traditional sliding window:
    - Traditional sliding window used in lower layers
    - Dilated sliding window used in higher layers
- Memory is still O(n*k)



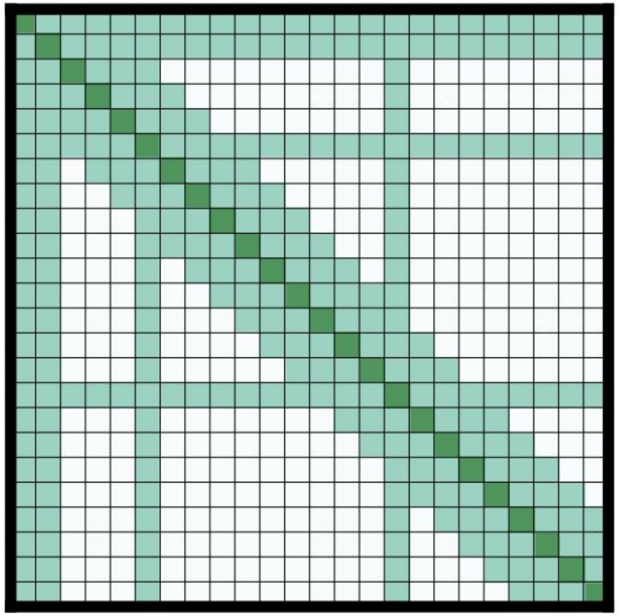

(c) Dilated sliding window

Global Attention

- Utilize some tokens to attend to everything
- The rest of the tokens will attend to a sliding window
- Example: [CLS] token in classification
- Memory: O(n*k + 2*s*n)

Sliding window

Each special token has to attend to every token, and the other way around

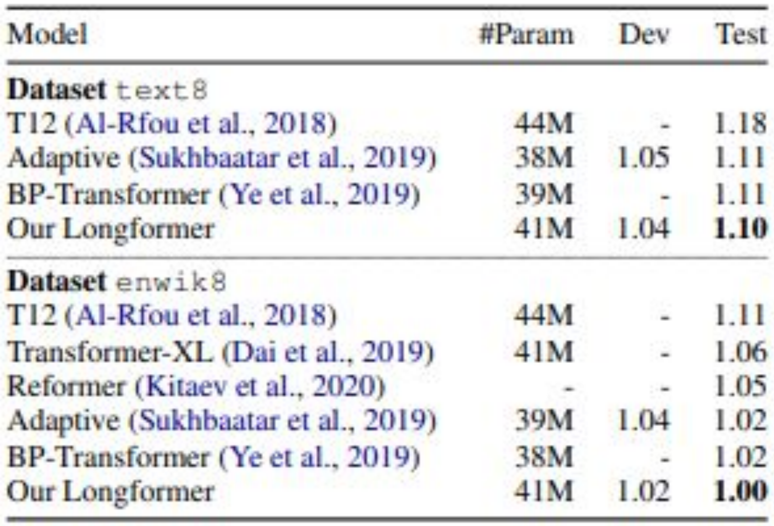


Linear Projections

- Utilize separate projections for sliding-window attention and global attention
  - Q_S, K_S, V_S for sliding-window
  - Q_G, K_G, V_G for global attention
- Separate projections provide flexibility to model both kinds of attention
- Global attention projections are initialized with sliding window projections

Autoregressive Language Modeling

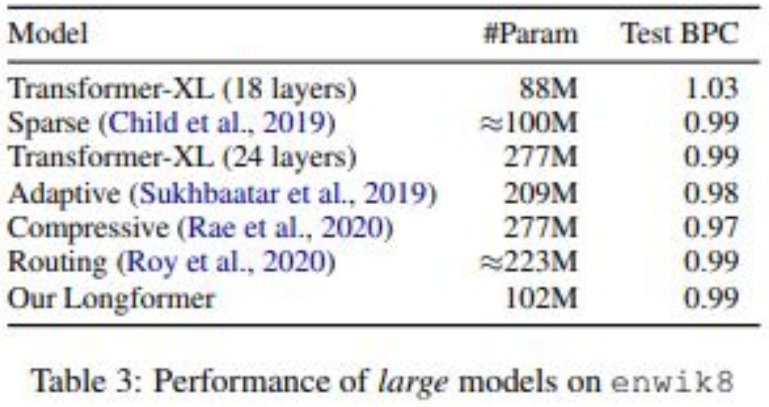
- Utilized dilated sliding window
- Added dilations in 2 highest layers to understand larger context
- No dilation in lower layers to understand local context



Model	#Param	Dev	Test
Dataset `text8`			
T12 (Al-Rfou et al., 2018)	44M	-	1.18
Adaptive (Sukhbaatar et al., 2019)	38M	1.05	1.11
BP-Transformer (Ye et al., 2019)	39M	-	1.11
Our Longformer	41M	1.04	**1.10**
Dataset `enwik8`			
T12 (Al-Rfou et al., 2018)	44M	-	1.11
Transformer-XL (Dai et al., 2019)	41M	-	1.06
Reformer (Kitaev et al., 2020)	-	-	1.05
Adaptive (Sukhbaatar et al., 2019)	39M	1.04	1.02
BP-Transformer (Ye et al., 2019)	38M	-	1.02
Our Longformer	41M	1.02	**1.00**

Autoregressive Language Modeling

- Also had good performance compared to much larger models
- Matches several models with double the parameters

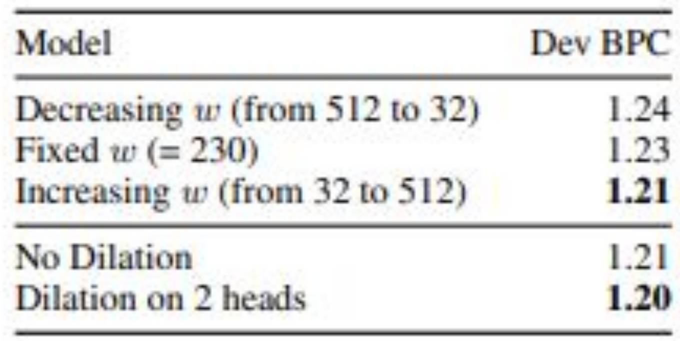


Model	#Param	Test BPC
Transformer-XL (18 layers)	88M	1.03
Sparse (Child et al., 2019)	≈100M	0.99
Transformer-XL (24 layers)	277M	0.99
Adaptive (Sukhbaatar et al., 2019)	209M	0.98
Compressive (Rae et al., 2020)	277M	0.97
Routing (Roy et al., 2020)	≈223M	0.99
Our Longformer	102M	0.99

Table 3: Performance of *large* models on `enwik8`

Ablation study

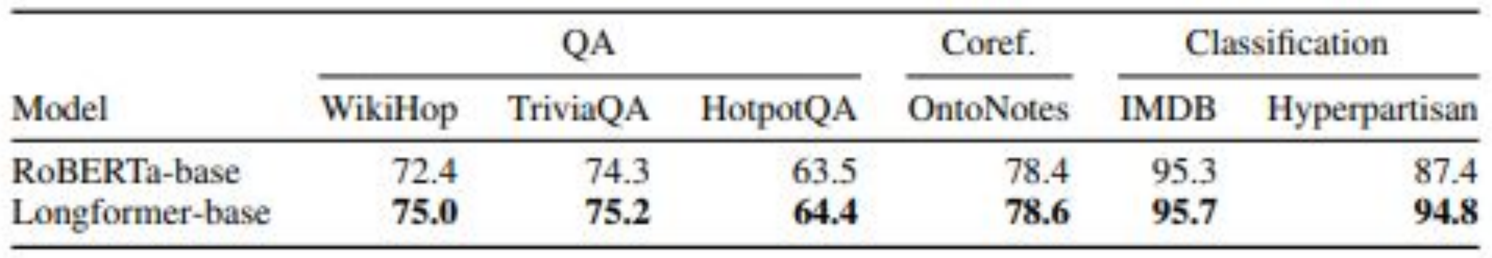
- Increasing the window size from the bottom to top layer has the best performance
- Adding dilations has better performance than not



Model	Dev BPC
Decreasing w (from 512 to 32)	1.24
Fixed w (= 230)	1.23
Increasing w (from 32 to 512)	**1.21**
No Dilation	1.21
Dilation on 2 heads	**1.20**

Pretraining/Fine-tuning on tasks

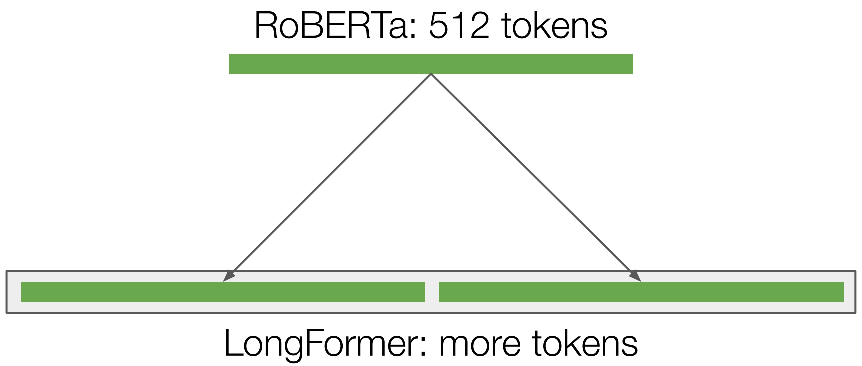
- Started from the RoBERTa released checkpoint - replacing self-attention with sliding window attention
  - Window size of 512
  - Added additional positional embeddings for the larger document size
- Utilized global attention to fine-tune on a task



	QA			Coref.	Classification	
Model	WikiHop	TriviaQA	HotpotQA	OntoNotes	IMDB	Hyperpartisan
RoBERTa-base	72.4	74.3	63.5	78.4	95.3	87.4
Longformer-base	**75.0**	**75.2**	**64.4**	**78.6**	**95.7**	**94.8**

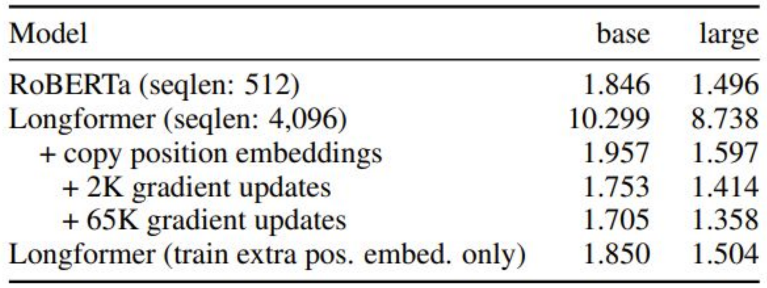
Position Embeddings

- LongFormer supports up to 4096 tokens, while RoBERTa only supports 512
- The RoBERTa position embeddings were copied over (8 times)
    - Why? BERT's positional embeddings have already been trained for local context
    - Issue at boundaries (every 512 tokens) resolved during further pretraining



MLM Pretraining

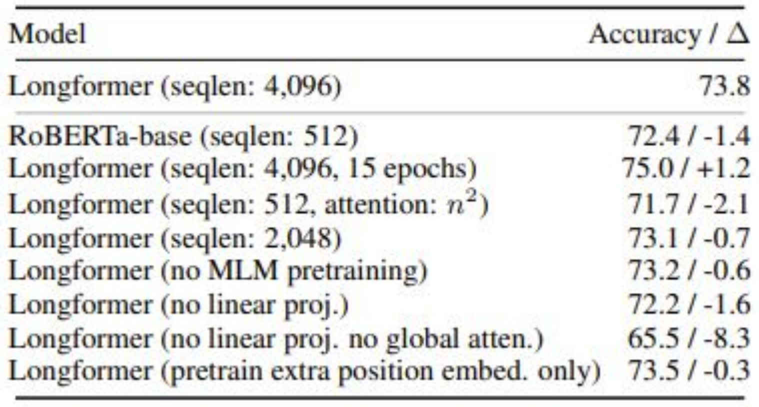
- From the RoBERTa checkpoint, LongFormer was pretrained on a corpus of long documents
- Copy initialization had the largest impact on performance



Model	base	large
RoBERTa (seqlen: 512)	1.846	1.496
Longformer (seqlen: 4,096)	10.299	8.738
+ copy position embeddings	1.957	1.597
+ 2K gradient updates	1.753	1.414
+ 65K gradient updates	1.705	1.358
Longformer (train extra pos. embed. only)	1.850	1.504

Ablations on WikiHop

- In general, LongFormer benefits from global attention, longer sequences, MLM pretraining, separate linear projections
- LongFormer with same sequence length and attention as RoBERTa performs slightly worse
    - Shows that performance gains are not from additional pretraining



Model	Accuracy / Δ
Longformer (seqlen: 4,096)	73.8
RoBERTa-base (seqlen: 512)	72.4 / -1.4
Longformer (seqlen: 4,096, 15 epochs)	75.0 / +1.2
Longformer (seqlen: 512, attention: n^2)	71.7 / -2.1
Longformer (seqlen: 2,048)	73.1 / -0.7
Longformer (no MLM pretraining)	73.2 / -0.6
Longformer (no linear proj.)	72.2 / -1.6
Longformer (no linear proj. no global atten.)	65.5 / -8.3
Longformer (pretrain extra position embed. only)	73.5 / -0.3

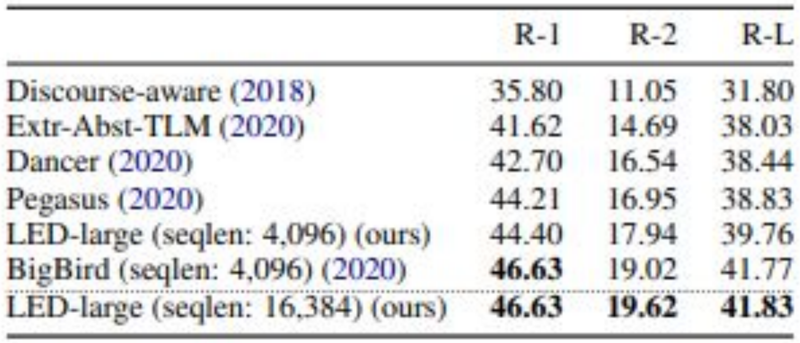
Table 10: WikiHop development set ablations

Memory efficiency

- Global attention = O(n*k+2*s*n) = O(n)
- With k = 512 (same as RoBERTa)
    - O(n*k + 2*s*n) = O(n*512 + 2*s+n) = O(n(512+2*s))
    - Since n > 512, this is more memory-intensive than RoBERTa
- However, the memory usage increases **linearly** as the document size or the window size changes
- Memory usage increases **quadratically** for self-attention

Longformer-Encoder-Decoder

- Same architecture/initialized parameters as BART, but with more positional embedding tokens
- Encoder uses local+global attention
- Decoder uses full self-attention on previous tokens as well as encoder tokens
- Evaluated on arXiv summarization dataset



	R-1	R-2	R-L
Discourse-aware (2018)	35.80	11.05	31.80
Extr-Abst-TLM (2020)	41.62	14.69	38.03
Dancer (2020)	42.70	16.54	38.44
Pegasus (2020)	44.21	16.95	38.83
LED-large (seqlen: 4,096) (ours)	44.40	17.94	39.76
BigBird (seqlen: 4,096) (2020)	**46.63**	19.02	41.77
LED-large (seqlen: 16,384) (ours)	**46.63**	**19.62**	**41.83**

Contributions

- Devised a new method for doing attention on a larger input size while maintaining a similar amount of memory
  - Sliding window attention
  - Dilated sliding window
  - Global attention

- Developed a state-of-the-art LLM utilizing local+global attention

Limitations + Future Work

- Memory usage isn’t exactly reduced compared to other LLMs
  - Often utilizes more memory
  - Key idea is that memory usage scales **linearly**, not **quadratically**
- Study the impact of increasing the sequence length
- Study additional pretraining objectives for Longformer Encoder Decoder

Efficient Streaming Language Models with Attention Sinks

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis

Theme: Extending Transformers to Greater Lengths

Length Extrapolation

Enable a model to handle sequences longer than those it was trained on

Context Window Expansion

Strategies enlarging the context windows of LLMs

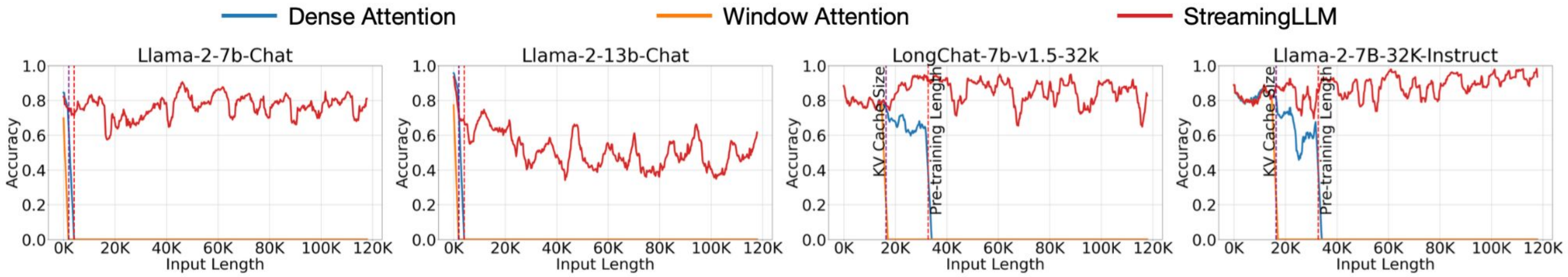
Improving LLMs' Utilization of Long Text

Just because a model can handle longer sequences doesn't mean it uses them effectively

This paper focuses on length extrapolation.

Motivation

- Transformer models have difficulty generalizing to sequence lengths longer than those they were trained on
  - We can train on longer sequence lengths (Context Window Expansion), but fundamentally the context length remains finite
    - Plus training on longer sequence lengths is very expensive

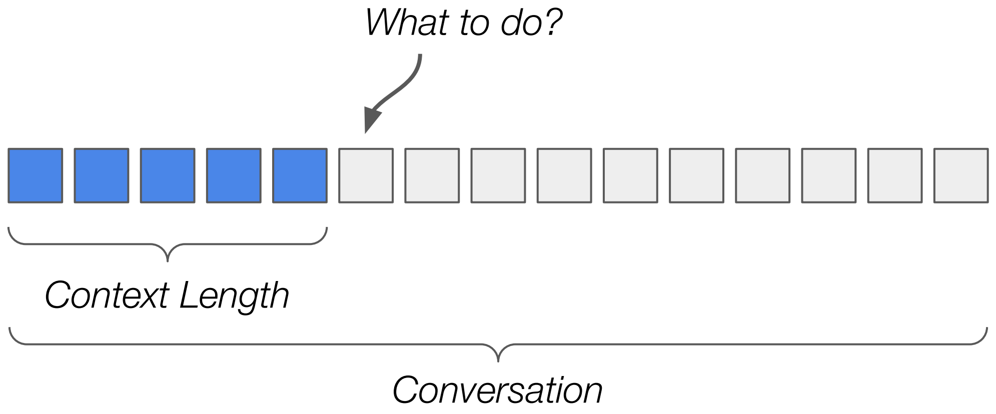


Performance of transformer models ("Dense Attention") plummet if we give it a sequence longer than those it was trained on

Why care about length extrapolation?

Being able to generalize to an infinite length of text is important for many real-world applications, such as AI chat assistants.

Imagine we could not generalize. Then, if you were talking with an AI assistant, it might suddenly forget all previous context once the conversation reaches the context length and it is forced to “reset”. Or, we will need to expensively re-compute all tokens’ KVs on a rolling basis.



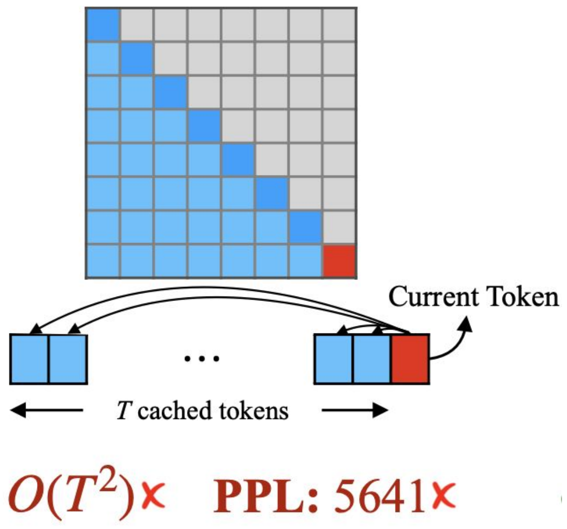
Previous Solutions: Dense Attention

Just give the transformer a sequence longer than those it was trained on!

Results

- Performance is awful ❌
- Scales quadratically with sequence length ❌

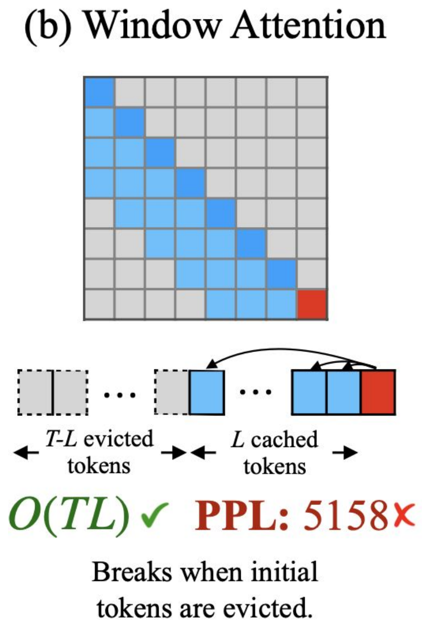
(a) Dense Attention

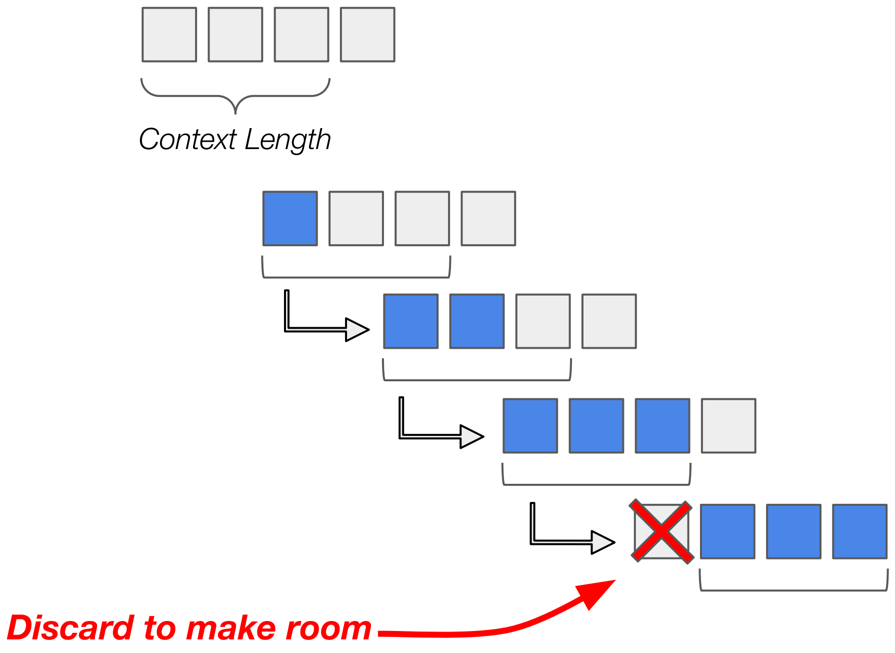


Has poor efficiency and performance on long text.

Previous Solutions: Window Attention

If we are already at the context length of the transformer, discard the earliest cached KV tokens to make way for new ones.





Previous Solutions: Window Attention

Results:

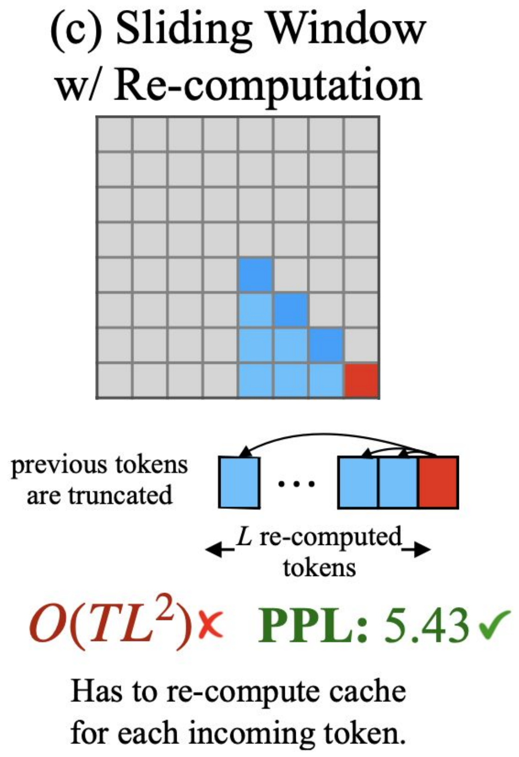
- Scaling: Linear ✅
- Once we begin losing initial tokens, performance degrades substantially ❌
  - Reason: We’ll get there!

Previous Solution: Sliding Window w/ Recomputation

For each new token, re-compute all tokens' KVs!
No caching at all!

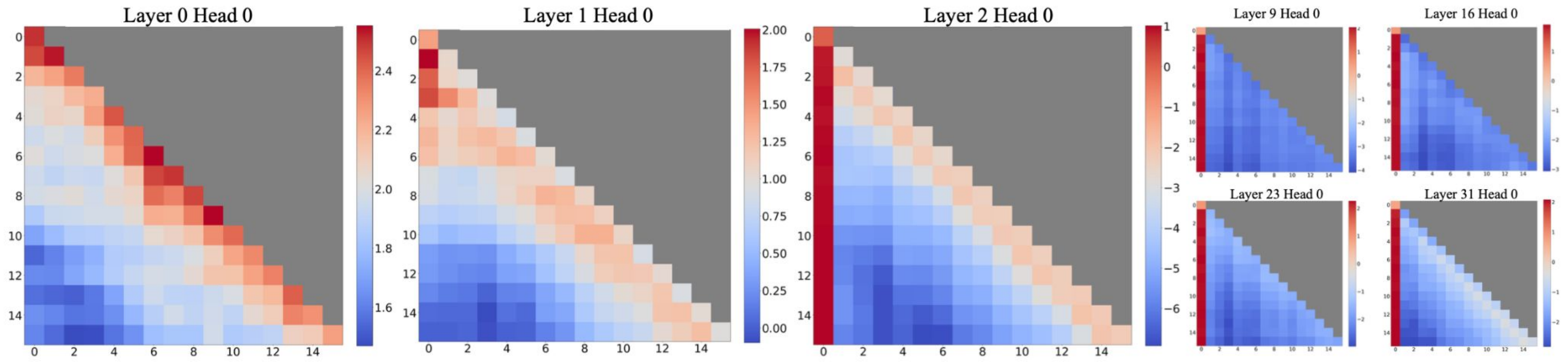
Results

- Good performance ✅
- Quadratic scaling with new tokens ❌
  - Seriously, no caching is really inefficient


(c) Sliding Window w/ Re-computation
previous tokens are truncated
L re-computed tokens
$O(TL^2)$ ✗
PPL: 5.43 ✓
Has to re-compute cache for each incoming token.

There’s a better solution! But first, **attention sinks**!

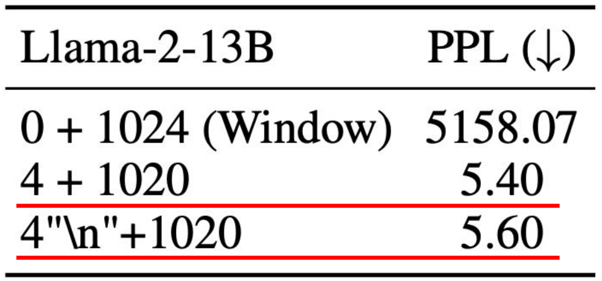
Observe the average attention logits in Llama-2-7b over 256 sentences, each of length 16



For later layers, notice how the initial token consistently has the highest values.

Even when the initial token is semantically insignificant

We can replace the first four tokens with line break tokens, and the attention sink phenomenon persists



Llama-2-13B	PPL (↓)
0 + 1024 (Window)	5158.07
4 + 1020	5.40
4"\n"+1020	5.60

Full understanding of this table requires some context covered later; the important thing here is that the attention sink phenomenon remains true even when four line break tokens are inserted at the front

So what’s going on?

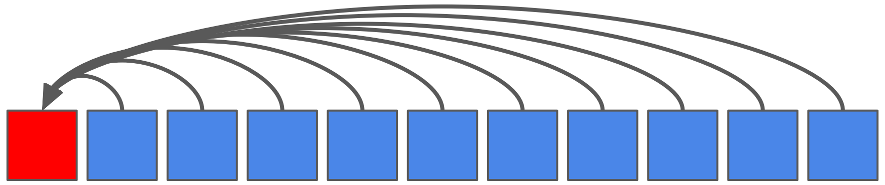
- Even when there isn’t a strong match between the query and any key, the softmax operation ensures attention scores must sum to one.
- So the model just sends all of that “extra” attention to the attention sink!

Image: https://www.americanstandard-us.com/pedestal-bathroom-sinks/townsend-8-inch-widespread-pedestal-sink-top-and-leg-combination/white-0328800020

Why the initial token as attention sink?

Simple; it’s visible to every token due to the autoregressive nature of decoders.

Making it an easy candidate to act as the sink.



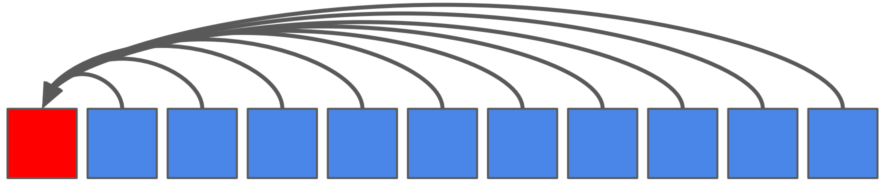
Recall Window Attention: Why does it fail?

Removal of attention sink causes a significant shift in the distribution of attention scores; the sink is now gone, so we have to pour the excess attention elsewhere.

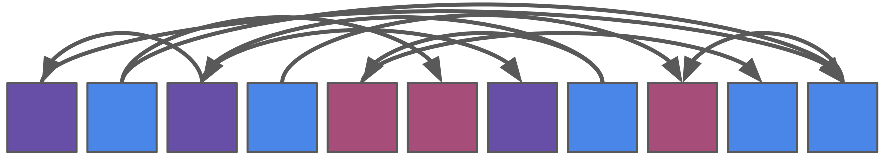
$$\text{SoftMax}(x)_i = \frac{e^{x_i}}{\underline{e^{x_1}} + \sum_{j=2}^{N} e^{x_j}}, \quad \underline{x_1} \gg x_j, j \in 2, \ldots, N$$

Initial token logit, x_1, is far greater than other logits

With attention sink

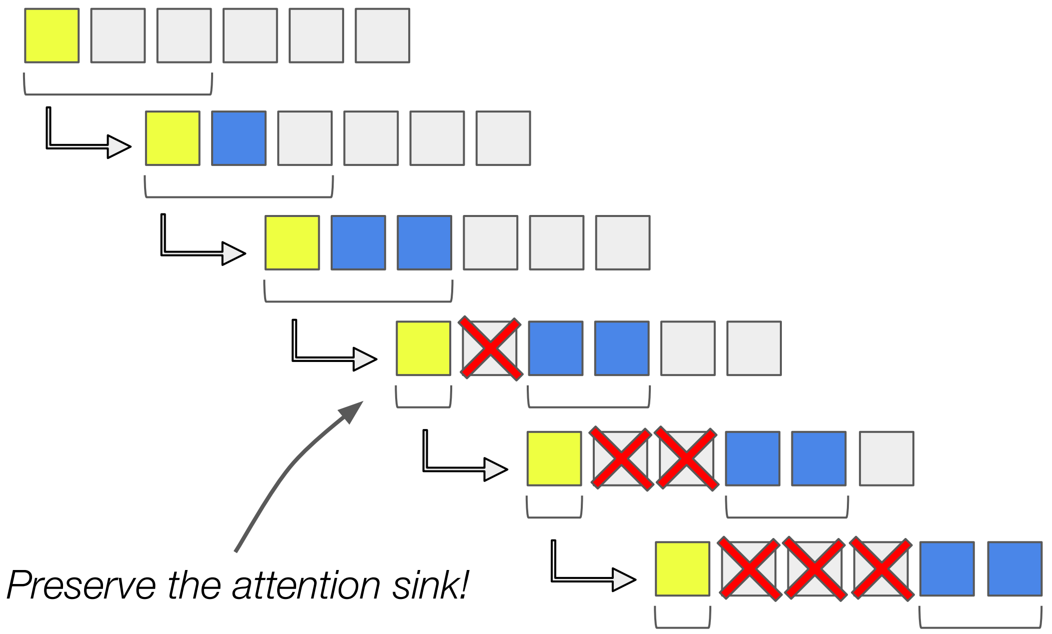


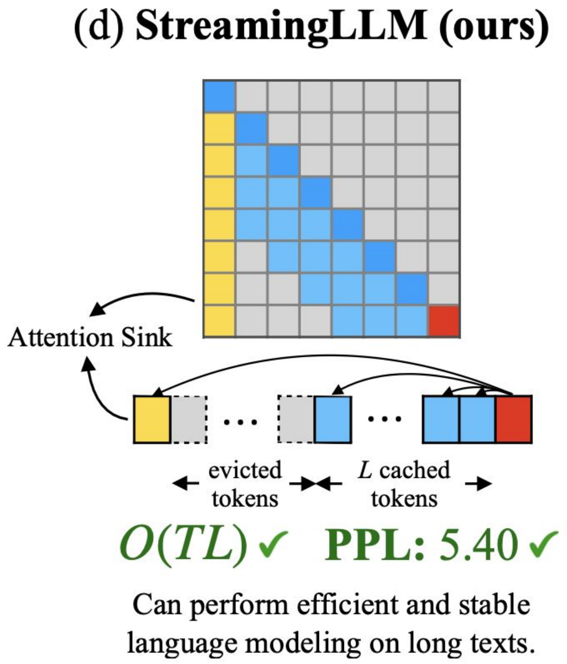
Attention sink removed



A Solution: StreamingLLM

If the removal of initial tokens causes window attention to fail, then let's just keep those tokens to continue acting as attention sinks!

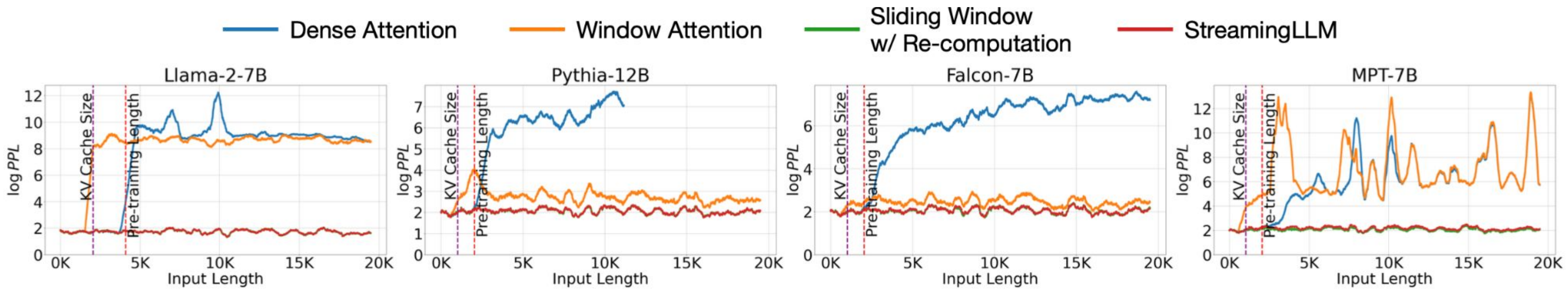




Comparing Length Extrapolation Methods

Graphs Show Log Perplexity vs Input Length

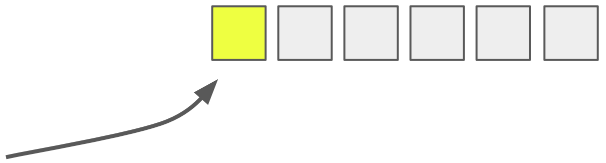
- **Dense Attention**
  - Fails when input length exceeds pre-trained length
- **Window Attention**
  - Fails when input length exceeds KV Cache Size (ie. initial token is discarded)
- **Sliding Window w/ Recomputation**
  - Maintains good performance throughout
- **StreamingLLM**
  - Nearly identical performance to Sliding Window w/ Recomputation



Pre-Training with Attention Sinks

The beauty of StreamingLLM is that it can be applied to pre-trained language models without further fine-tuning.

But we can also pre-train an LLM from scratch with a **trainable** “Sink Token” as the initial token.


Trainable Sink Token as initial token

A Different Solution: Softmax-off-by-One (Zero-Sink)

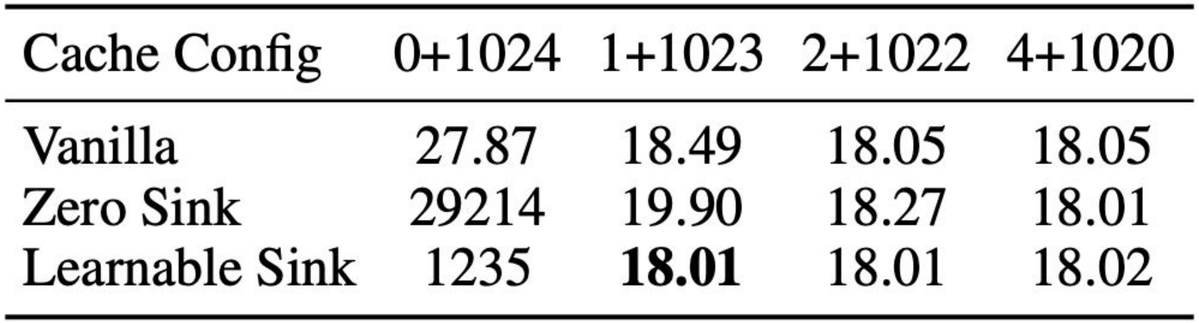
By adding a 1 into the denominator of the softmax function, the attention scores no longer need to sum to 1

$$\mathbf{SoftMax}_1(x)_i = \frac{e^{x_i}}{\underline{1} + \sum_{j=1}^{N} e^{x_j}}$$

This is conceptually equivalent to using a token with all-zero Key and Value features. Hence, it is also called “Zero-Sink”

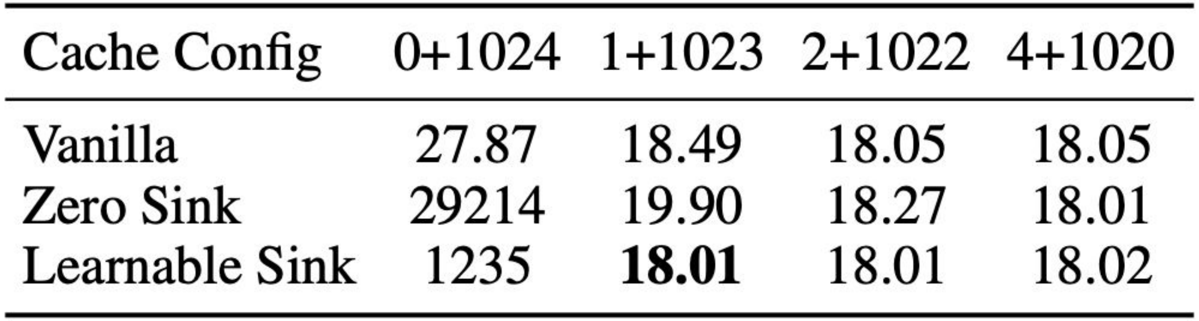
How many tokens as attention sinks?

- “Initial token” was a slight simplification; models may use the first *few* tokens as attention sinks.
- Here, we see three 160-million parameter language models pre-trained from scratch. One is a vanilla model (no modifications), one uses Softmax-off-by-One (Zero Sink), and one was pre-trained with a trainable Sink Token.
- Model is then tested on sequences beyond its context length on PG19 dataset via rolling Window Attention method using between 0-4 initial tokens as attention sinks:



Cache Config	0+1024	1+1023	2+1022	4+1020
Vanilla	27.87	18.49	18.05	18.05
Zero Sink	29214	19.90	18.27	18.01
Learnable Sink	1235	**18.01**	18.01	18.02

Vanilla vs Zero-Sink vs Learnable Sink

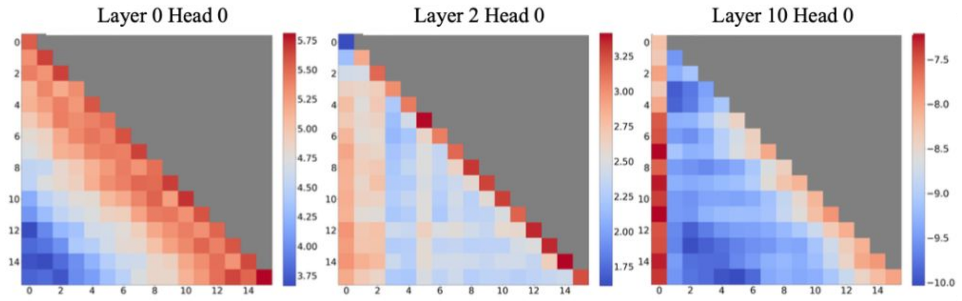


Cache Config	0+1024	1+1023	2+1022	4+1020
Vanilla	27.87	18.49	18.05	18.05
Zero Sink	29214	19.90	18.27	18.01
Learnable Sink	1235	**18.01**	18.01	18.02

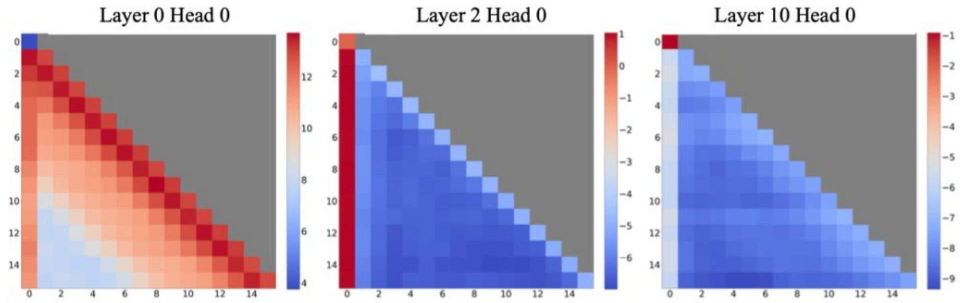
- **Vanilla:** Having first 2 initial tokens as sink results in lower perplexity than just 1
- **Zero Sink:** Having first 4 initial tokens as sink results in lowest perplexity
- **Learnable Sink:** Perplexity does not decrease further when using more than 1 initial tokens as attention sink

Attention Visualization

Model pre-trained with Sink Token utilizes it far more clearly and in earlier layers



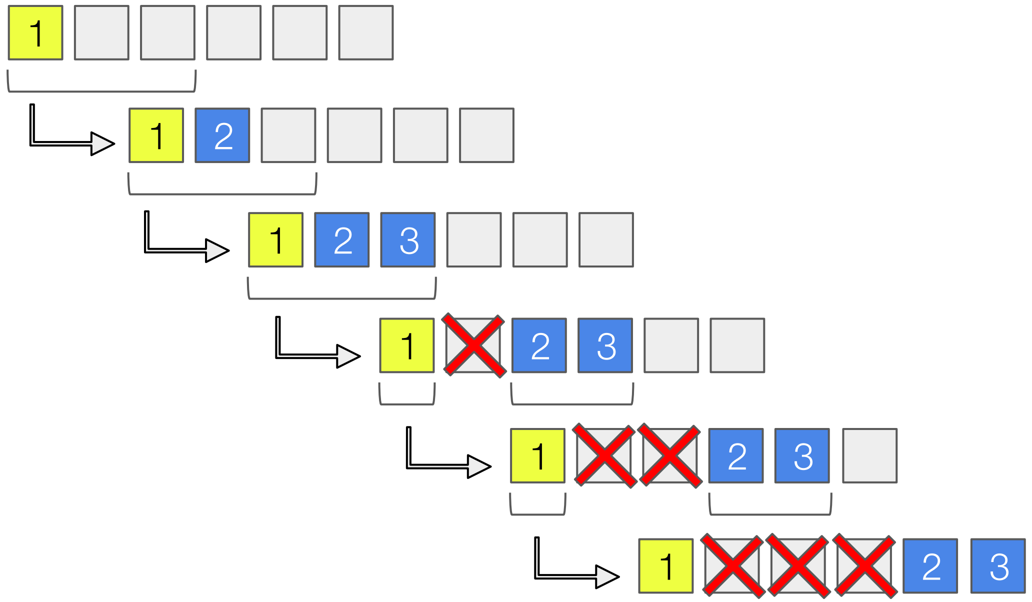
Pre-Trained without Sink Token



Pre-Trained with Sink Token

StreamingLLM: Position Encoding

SteamingLLM focuses on positions *within* the rolling cache instead of those in the text



Compatible with Any Relative Positional Encoding Scheme

RoPE

Because each iteration the position of each token within the cache will decrease (except attention sink tokens), StreamingLLM caches KVs *before* the RoPE rotary transformation is applied. Thus, the rotary transformation corresponding to a different position can be applied to the KVs later on.

ALiBi

In ALiBi, the bias is applied to the attention scores directly - so we can cache KVs without worry.

Experiments

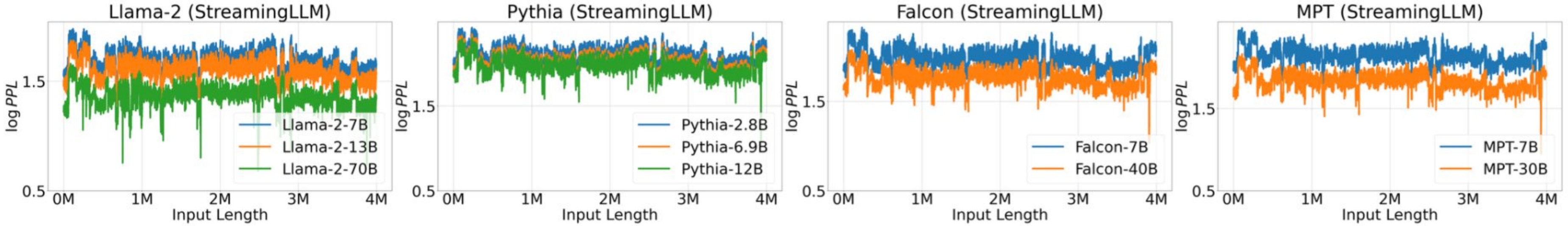
- Perplexity of Common Models Implementing StreamingLLM On Super-Long Texts
- Sink Token Pre-Training Results
- Comparison of Methods on Instruction-Tuned Models

Perplexity of Common Models Implementing StreamingLLM On Super-Long Texts

Graphs Show Log Perplexity vs Input Length

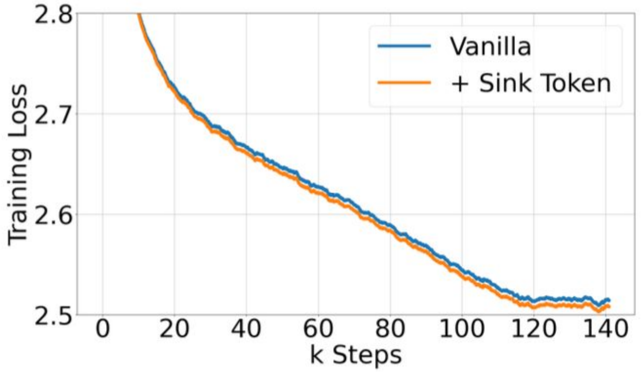
Text length of 4 million tokens.
From concatenated test set of PG19 (100 books).

As we can see, performance remains consistent. Fluctuations attributed to transitions between books



Sink Token Pre-Training Results

Compare two 160m parameter models, one pre-trained w/ Sink Token and one not (vanilla)



Convergence
Pre-training loss curves nearly identical

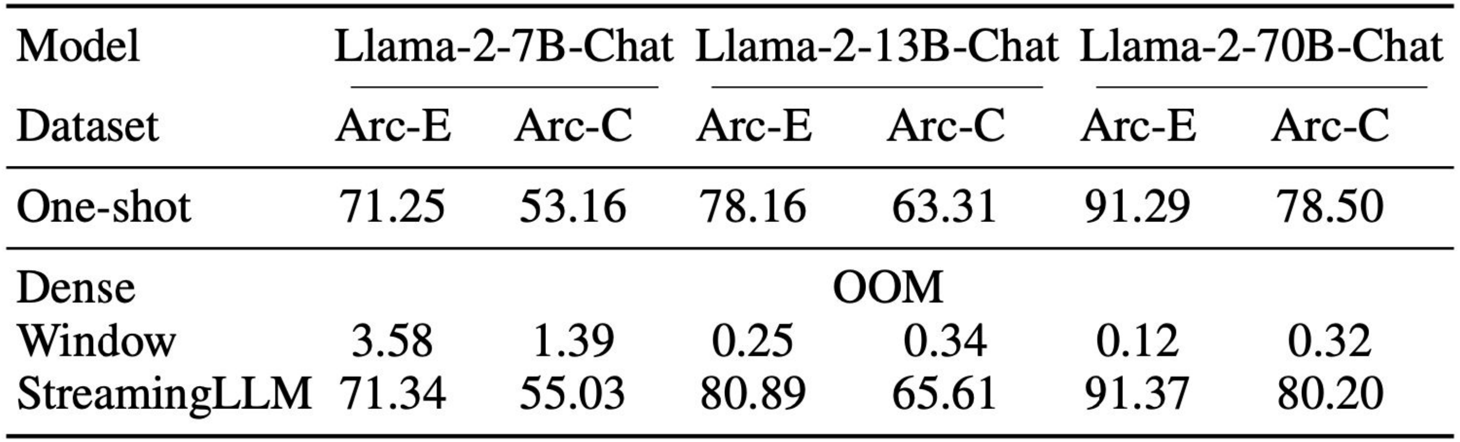
Methods	ARC-c	ARC-e	HS	LBD	OBQA	PIQA	WG
Vanilla	18.6	45.2	29.4	39.6	16.0	62.2	50.1
+Sink Token	**19.6**	**45.6**	**29.8**	**39.9**	**16.6**	**62.6**	**50.8**

Accuracy (%)
Zero-shot performance across multiple benchmarks. Sink token pre-training does not harm accuracy

Comparison of Methods on Instruction-Tuned Models

Accuracy (%) on ARC-[Easy, Challenge] datasets. Questions concatenated and streamed

StreamingLLM has comparable performance with one-shot sample-by-sample



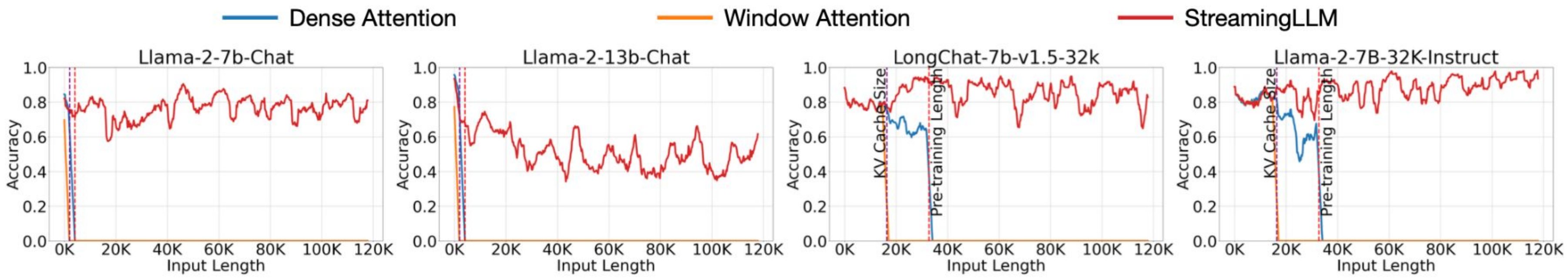
Model	Llama-2-7B-Chat		Llama-2-13B-Chat		Llama-2-70B-Chat	
Dataset	Arc-E	Arc-C	Arc-E	Arc-C	Arc-E	Arc-C
One-shot	71.25	53.16	78.16	63.31	91.29	78.50
Dense	OOM					
Window	3.58	1.39	0.25	0.34	0.12	0.32
StreamingLLM	71.34	55.03	80.89	65.61	91.37	80.20

OOM = Out of Memory

StreamEval

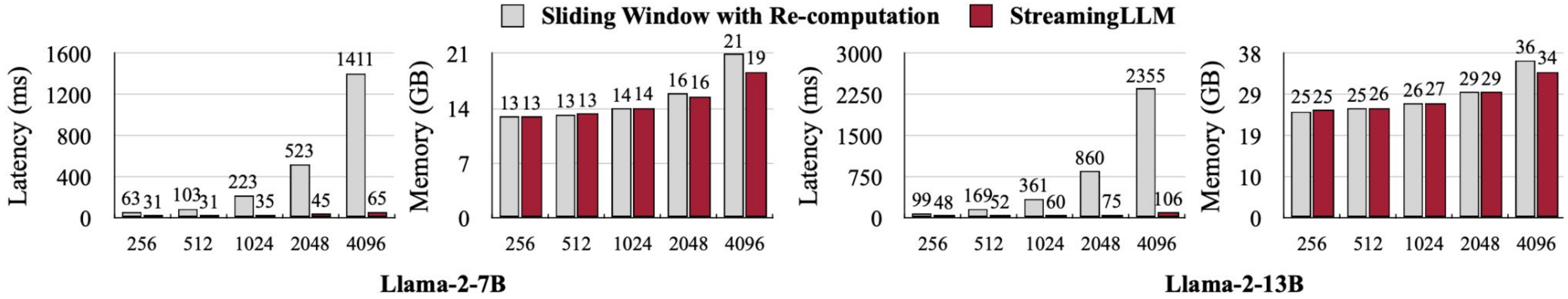
A dataset inspired by LongEval

- LongEval had a single query over a long sequence of text
- StreamEval has a query every 10 lines
- Each query's answer is 20 lines prior
    - Reflects real-world scenarios where questions pertain to recent information



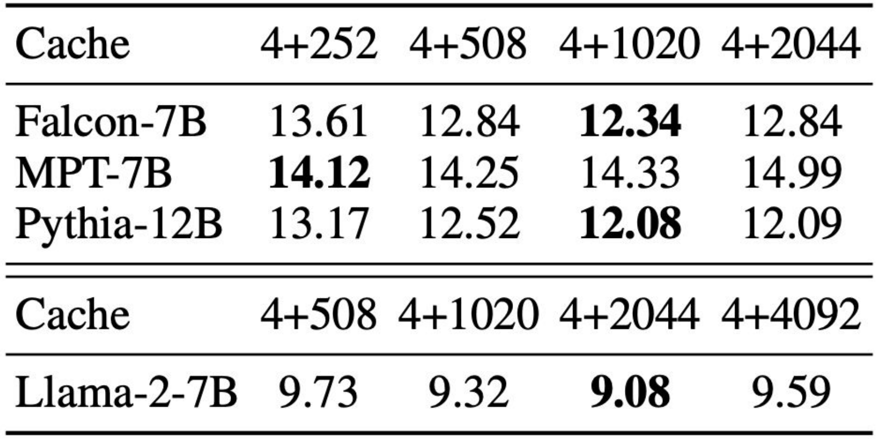
Efficiency

Using Sliding Window w/ Recomputation as baseline, StreamingLLM possesses similar memory footprint while achieving up to 22.2x speedup per token



Limitation - Increasing Cache Sizes Does Not Always Improve

- Increasing cache size does not consistently yield a decrease in perplexity across all models for StreamingLLM.
- Perplexities in table evaluated on concatenated PG19 dataset with 400k tokens.
- Indicates potential difficulties with proper **utilization** of long text
  - Just because there's a bunch of text doesn't mean the LLM can use it well!



Cache	4+252	4+508	4+1020	4+2044
Falcon-7B	13.61	12.84	**12.34**	12.84
MPT-7B	**14.12**	14.25	14.33	14.99
Pythia-12B	13.17	12.52	**12.08**	12.09

Cache	4+508	4+1020	4+2044	4+4092
Llama-2-7B	9.73	9.32	**9.08**	9.59

SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot

Elias Frantar and Dan Alistarh

Table of Contents:

Motivation: The What Behind Pruning

The democratization and usability of LLM's warrants research to reduce the computational cost of the model without sacrificing much on performance

Current research areas include:

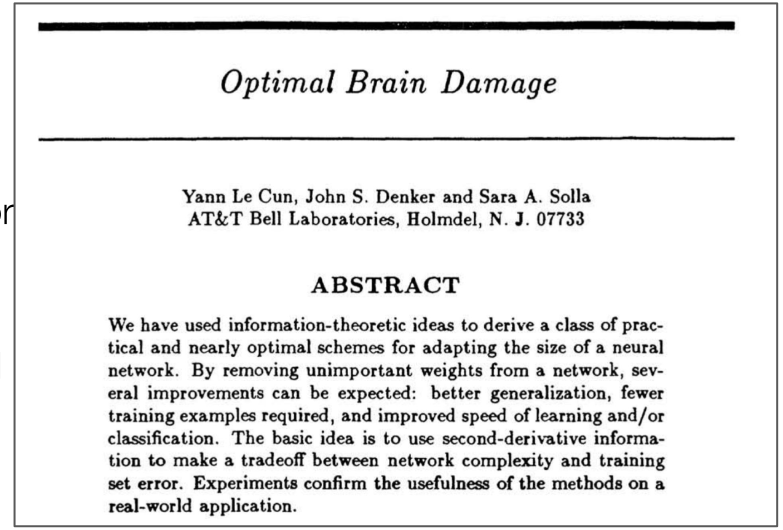
Technique		Description
1	Pruning	• Removing neural connections to create smaller sparse models • Two Scopes: Structured and Unstructured •
2	Quantization	• Reducing precision of parameters to save cost with less space • Weight Quantization → saves memory, but not faster • Activation Quantization → faster, but calibration required
3	Knowledge Distillation	• A student model learns to predict outputs of teacher model • It's more efficient for student to learn the teacher's distribution

Motivation: The Why Behind Pruning

Pruning has been well documented a method for deploying smaller, more efficient models since Le-Cun's 1990 paper.

Generally, LLMs are very overparameterized

- Pruning = cutting down these connections containing unused or relatively "unimportant" parameters
- We expect this to result in faster inference and training time and lesser capacity for storage
- Obviously pruning comes at the expense of performance and generalizability
    - So we prune to optimize both cost and performance



Optimal Brain Damage

Yann Le Cun, John S. Denker and Sara A. Solla
AT&T Bell Laboratories, Holmdel, N. J. 07733

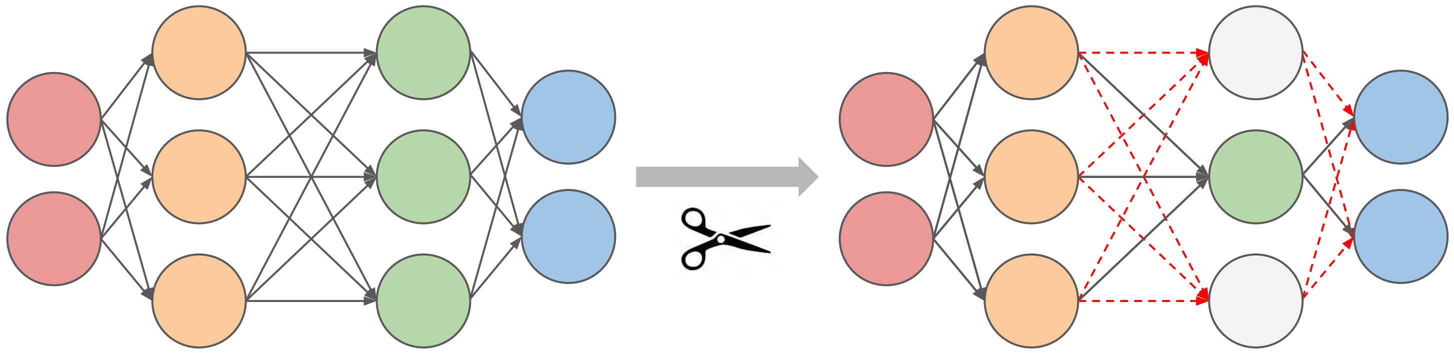
ABSTRACT

We have used information-theoretic ideas to derive a class of practical and nearly optimal schemes for adapting the size of a neural network. By removing unimportant weights from a network, several improvements can be expected: better generalization, fewer training examples required, and improved speed of learning and/or classification. The basic idea is to use second-derivative information to make a tradeoff between network complexity and training set error. Experiments confirm the usefulness of the methods on a real-world application.

Background: The Intuition Behind Pruning

There are generally three scopes of pruning methods:

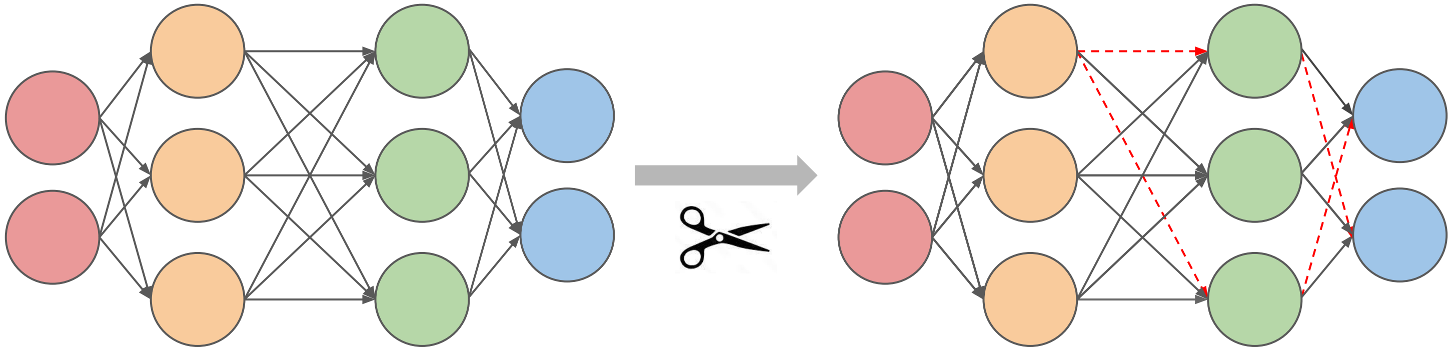
1. Structured Pruning
    a. Systematically removing groups of connections (weights), significantly reducing the size of the model and saving on compute
    b. However, may come at a bigger cost of performance, and limits "maximum sparsity"



Background: The Intuition Behind Pruning

There are generally three scopes of pruning methods:

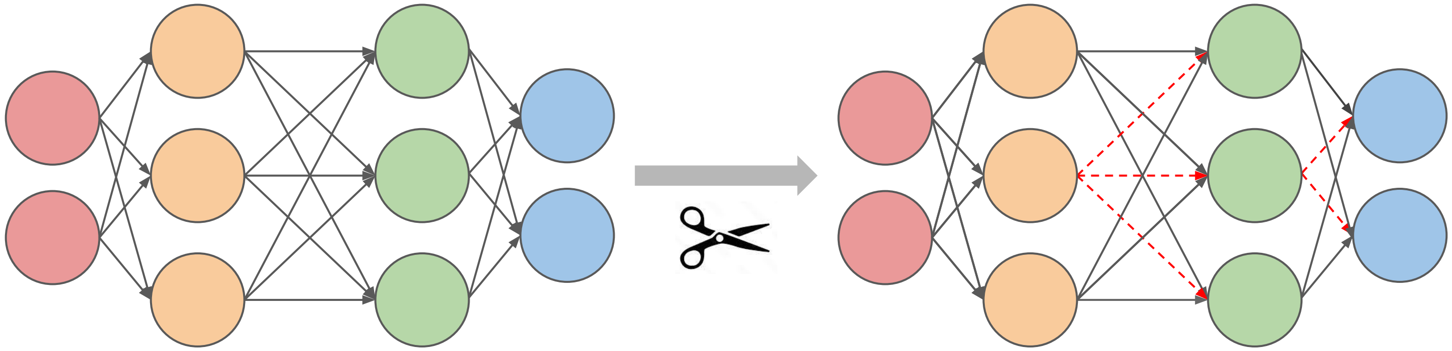
2. Unstructured Pruning
    a. Systematically removing individual connections (weights) based on a certain criteria
    b. Might be less abruptly changing the size of the model, but more granular, meaning that it's more apt to approaching maximum sparsity
    c. e.g Magnitude based pruning; let x = pruning factor, set least x% of weights to 0!



Background: The Intuition Behind Pruning

There are generally three scopes of pruning methods:

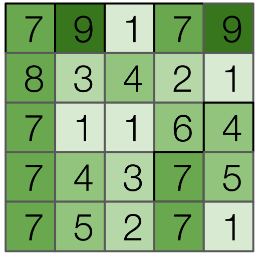
3. Semi-Structured Pruning
   a. Attempts to balance the benefits of the previous types of pruning
   b. Removes a group of connections (weights) that are not as large scale as structured pruning, but still more substantial than the individual weights removed in unstructured pruning
   c. Usually presented with a ratio K:N, where for every block of N weights, K are pruned!



Background: Additional Notes on Pruning

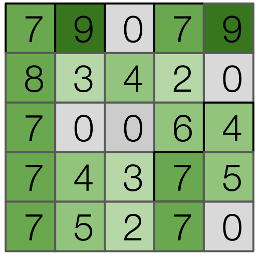
- One-shot pruning
  - The removal of connections (weights) happen in one pass (they happen at once)
  - This is independent of the scope of pruning (that is to say each of the three can occur in one shot)

- For an attention matrix, the process of pruning is very similar
  - Filling in 0's in the matrix doesn't mean it's a no-op, it won't prevent the operation from occurring
  - Therefore, we'll need a sparse Matmul algorithm to ignore these!



7	9	1	7	9
8	3	4	2	1
7	1	1	6	4
7	4	3	7	5
7	5	2	7	1

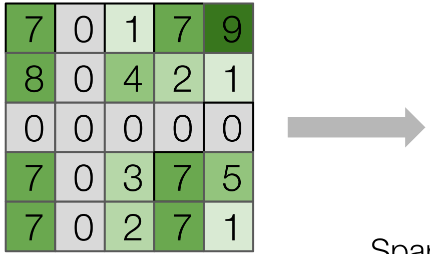
Attention Matrix (before pruning)



7	9	0	7	9
8	3	4	2	0
7	0	0	6	4
7	4	3	7	5
7	5	2	7	0

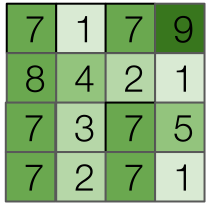
Unstructured Pruning

(This is also an example of Magnitude Based Pruning)



7	0	1	7	9
8	0	4	2	1
0	0	0	0	0
7	0	3	7	5
7	0	2	7	1

Structured Pruning



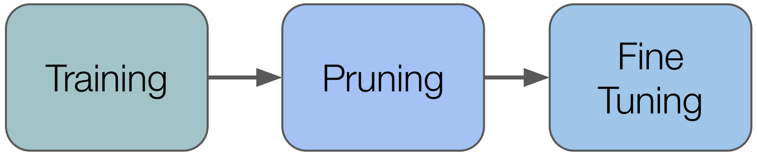
7	1	7	9
8	4	2	1
7	3	7	5
7	2	7	1

Sparse Matmul designed to skip 0's (Row 3, Col 2 are thus omitted)

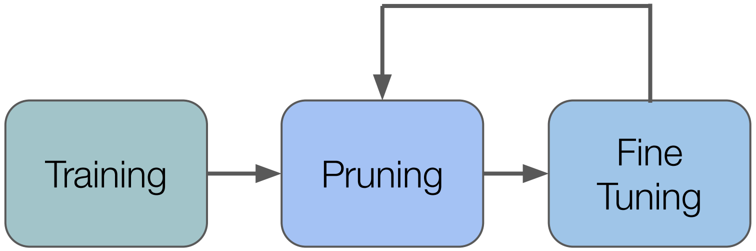
Background: Post Training Pruning

Post Training Pruning:

- Pruning a well-trained and well-optimized model (usually with calibration data) without the need for any retraining afterwards
  - While post-training quantization is well documented, it has been shown to work with pruning too
  - This is practical, as having to retrain LLMs can be very resource consuming



One-Shot Pruning



Iterative Pruning

Background: Layer Wise Pruning

- Selectively removing weights, considering one layer at a time
  - Justification: some layers may have more “unimportant” weights

- How do we know if our pruning is optimal (balances sparsity and accuracy)?
  - Layer-wise pruning → “layer-wise” subproblems → then recombining the compressed layers
  - Then we can find L2 error between outputs before and after pruning
  - (Hubara et al., 2021a) specifically defined this as the following optimization problem:

For a layer l, let M_l be the sparsity mask and $\hat{W}_l$ be the possibly updated weights

$$\text{argmin}_{\text{mask } \mathbf{M}_\ell, \widehat{\mathbf{W}}_\ell} ||\mathbf{W}_\ell \mathbf{X}_\ell - (\mathbf{M}_\ell \odot \widehat{\mathbf{W}}_\ell)\mathbf{X}_\ell||_2^2$$

Find the values of M_l and $\hat{W}_l$ that minimize the following:

The original inputs to l times the original weights for l

Element wise multiply M_l and $\hat{W}_l$. Re-multiply by X_l for new output

After subtracting outputs, we find the L2 Norm

Background: Layer-wise Pruning Problem

- One thing to note is that optimizing the mask and the weights jointly is np-hard
  - This is why similar works approach this problem with approximations

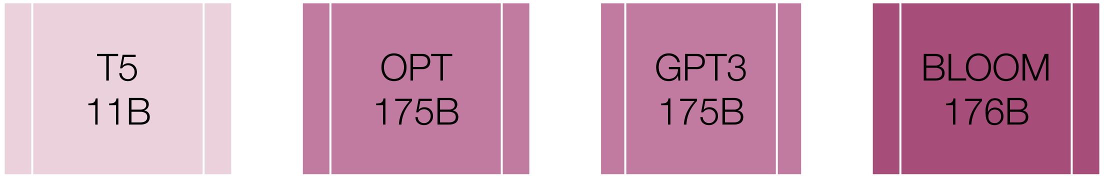
- One work-around: split problem into mask selection and weight reconstruction
  - “Once a mask is fixed, optimizing the remaining unpruned weights is a linear squared error problem”

- Noble Past Efforts
  - Ada Prune as introduced by (Hubara et al., 2021a)
    - featured magnitude based pruning for mask selection and SGD for weight reconstruction
  - Iterative AdaPrune as previously introduced by the authors of this paper
  - OBC as introduced by (Frantar et al., 2022b)

Background: Difficulty of Scaling to 100B parameters

- Difficulty of Scaling to 100+ Billion Parameters
  - The existing work only has the ability to compress models ranging in the several 100's of millions of parameters (and to do this they may even take up to a couple hours)
  - This does not scale well for GPT family models whose sizes sit comfortably around 175 Billion parameters

- Reminder: We wanted a way to compress GPT family LLMs
  - Existing top performing GPT models have a lot of demand for deployment, but are too large
    - 175 Billion Parameters → 320 GB in FP16 format → needs at least 5 A100 GPUs!
    - If only there was a way…



Introduction: Sparse-GPT

SparseGPT: A new pruning method

- “GPT family models can be pruned to at least 50% sparsity”
  - One-shot
  - No retraining, so this is “post training pruning”
  - Minimal loss of accuracy (perplexity, 0-shot accuracy)

- Effectiveness shown on open source LLM’s (OPT-175B, BLOOM-176B,)
  - <4.5 hours to compress
  - 60% unstructured sparsity
  - >100 billion weights ignored at inference

Introduction: Prior Work

- Researchers claim that before Sparse-GPT almost all work on compressing GPT family models have been in *quantization*

- Most of the effective pruning methods explored are not post-train pruning!
    - The very few effective one-shot pruning methods don't scale well to 175B parameters
        - E.g: Ada Prune as introduced by (Hubara et al., 2021a)

- Thus, the researchers comfortably claim that there is no work showing success with pruning LLM's on the scale of 100B parameters

Introduction: Sparse GPT at a glance

- "First accurate one shot pruning method for models 10-100+ billion parameters"
    - Takes only a couple of hours for the largest models, and can be used alongside quantization
    - Larger the model → the more compressible it is

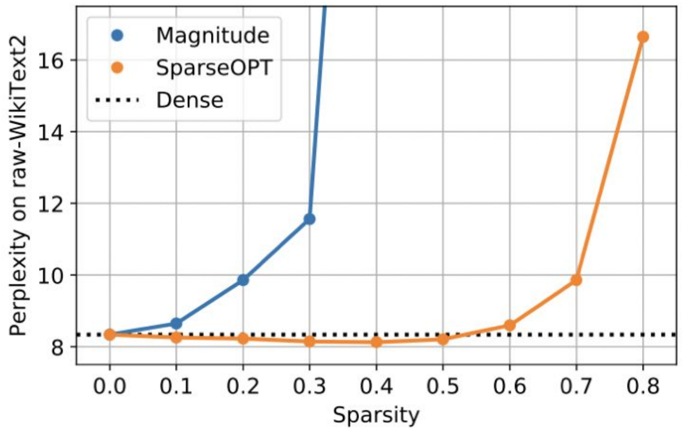


Figure 1. Sparsity-vs-perplexity comparison of `SparseGPT` against magnitude pruning on OPT-175B, when pruning to different *uniform* per-layer sparsities.

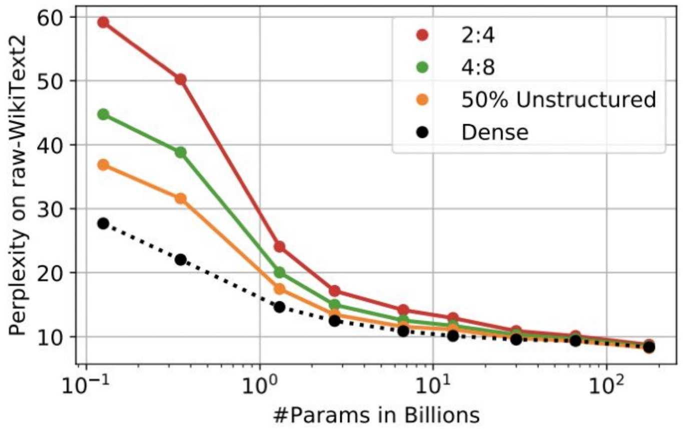


Figure 2. Perplexity vs. model and sparsity type when compressing the entire OPT model family (135M, 350M, . . . , 66B, 175B) to different sparsity patterns using `SparseGPT`.

The Row Hessian Challenge

- In order to explain the Sparse GPT's algorithm, we must first understand how the authors approach the layer-wise pruning problem discussed
  - Surely, they intend to fix the sparse mask first and then proceed layer by layer
  - However, now they must address the subproblem of weight reconstruction

- Let's consider the sparse reconstruction problem for a matrix
  - $\mathbf{w^i}_{\mathbf{M_i}} = (\mathbf{X}_{\mathbf{M_i}}\mathbf{X}_{\mathbf{M_i}}^\top)^{-1}\mathbf{X}_{\mathbf{M_i}}(\mathbf{w}_{\mathbf{M_i}}\mathbf{X}_{\mathbf{M_i}})^\top$ this is very similar to the previous equation!

  - $\mathbf{w^i}_{\mathbf{Mi}}$ represents the optimal weight values for a fixed mask in layer i
  - $\mathbf{X}_{\mathbf{Mi}}$ is a subset of inputs whose weights are not pruned
  - $\mathbf{w}_{\mathbf{Mi}}$ is a subset of weights corresponding to X_{Mi}
  - $(\mathbf{X}_{\mathbf{Mi}}\mathbf{X^T}_{\mathbf{Mi}})^{-1}$ is the inverse Hessian, $(\mathbf{H}_{\mathbf{Mi}})^{\mathbf{-1}}$, shows impact of pruning weight on loss function
  - $\mathbf{H}_{\mathbf{Mi}}$ is applied to a product of the values and inputs $\mathbf{X}_{\mathbf{Mi}}(\mathbf{w}_{\mathbf{Mi}}\mathbf{X}_{\mathbf{Mi}})^{\mathbf{T}}$

The Row Hessian Challenge

- Notice how each layer 'i' demands a recalculation of the hessian, $(X_{Mi}X^T_{Mi})^{-1}$, H_{Mi}
  - As it turns out this calculation is very computationally heavy and demanding
  - Without finding a way to hasten this process, achieving a feasible algorithm is difficult
  - This forms the basis for what's called the *Row Hessian Challenge*

- Why does this recalculating need to occur?
  - Because each row needs a unique hessian, why?
  - Because each row has a unique mask
    - and we know that $(H_{Mi})^{-1} \neq (H^{-1})_{Mi}$
  - Why can't we fix 1 row-mask for all rows?
    - compromises model accuracy

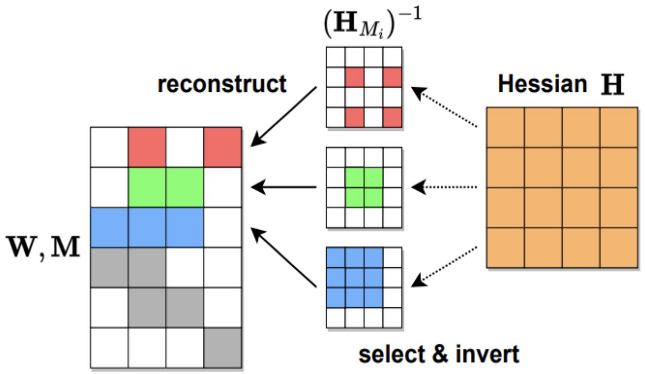


Figure 3. Illustration of the row-Hessian challenge: rows are sparsified independently, pruned weights are in white.

Solving The Row Hessian Challenge

- Enter: the Sparse GPT algorithm
  - We need a way to reuse Hessians row-wise while still enabling distinct pruning masks!
  - We proceed by column; distinct layer-wise masks, but reuse the same hessian across rows
  - Moving through each column, the algorithm takes an iterative approach, stopping at a target sparsity

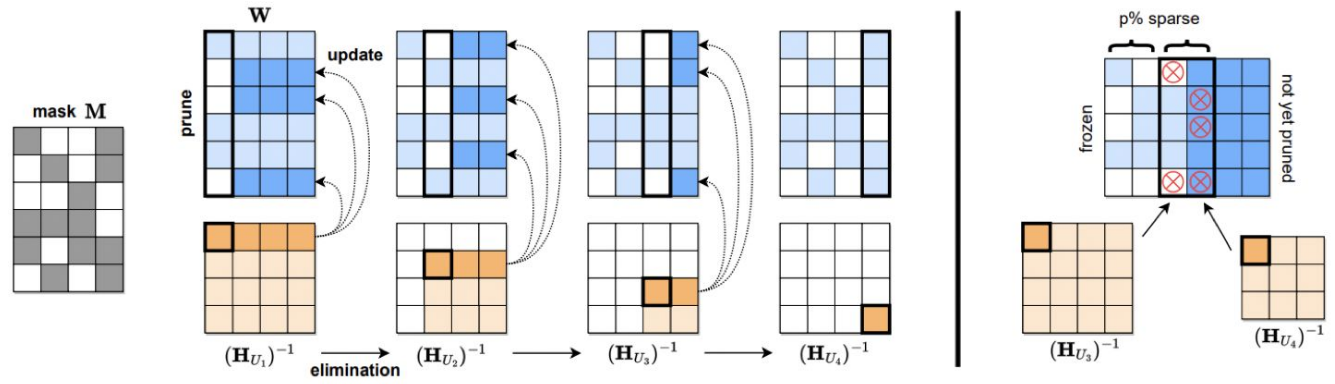
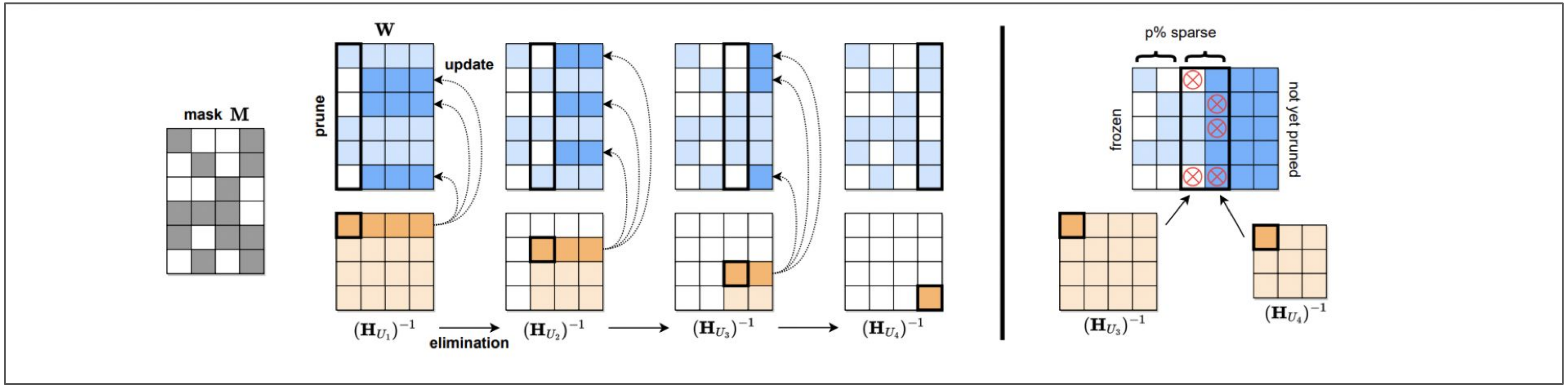

mask $\mathbf{M}$
W
prune
update
elimination
$(\mathbf{H}_{U_1})^{-1}$
$(\mathbf{H}_{U_2})^{-1}$
$(\mathbf{H}_{U_3})^{-1}$
$(\mathbf{H}_{U_4})^{-1}$
p% sparse
frozen
not yet pruned
$(\mathbf{H}_{U_3})^{-1}$
$(\mathbf{H}_{U_4})^{-1}$

Figure 4. [Left] Visualization of the `SparseGPT` reconstruction algorithm. Given a fixed pruning mask $\mathbf{M}$, we incrementally prune weights in each column of the weight matrix $\mathbf{W}$, using a sequence of Hessian inverses $(\mathbf{H}_{U_j})^{-1}$, and updating the remainder of the weights in those rows, located to the "right" of the column being processed. Specifically, the weights to the "right" of a pruned weight (dark blue) will be updated to compensate for the pruning error, whereas the unpruned weights do not generate updates (light blue). [Right] Illustration of the adaptive mask selection via iterative blocking.

Solving The Row Hessian Challenge

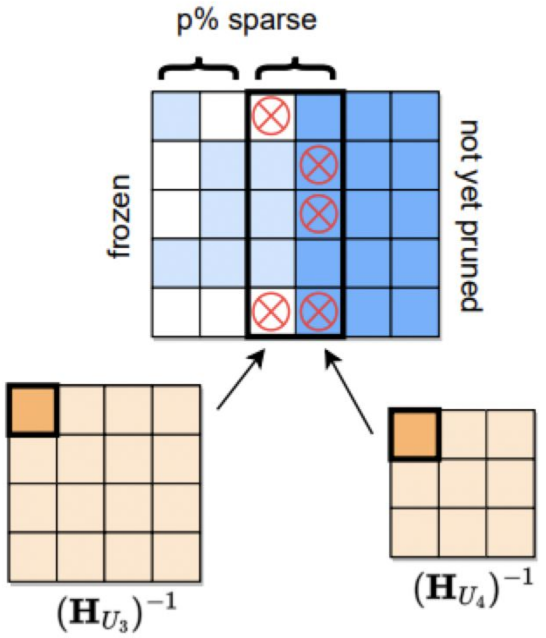
- Hessian Synchronization
  - Each column gets its own hessian inverse matrix (created via gaussian elimination of 1 row and column)
  - After a mask prunes weights in a column, $(H_{Ui})^{-1}$ is used to update all weights to the right of the pruned
  - This is to minimize the pruning error (remember $(H_{Ui})^{-1}$ contains relevant information about loss)
- Computational Complexity
  - As it turns out, this new approach is efficient enough to support an effective algorithm!


mask M
W
prune
update
$(\mathbf{H}_{U_1})^{-1}$
elimination
$(\mathbf{H}_{U_2})^{-1}$
$(\mathbf{H}_{U_3})^{-1}$
$(\mathbf{H}_{U_4})^{-1}$
p% sparse
frozen
not yet pruned
$(\mathbf{H}_{U_3})^{-1}$
$(\mathbf{H}_{U_4})^{-1}$

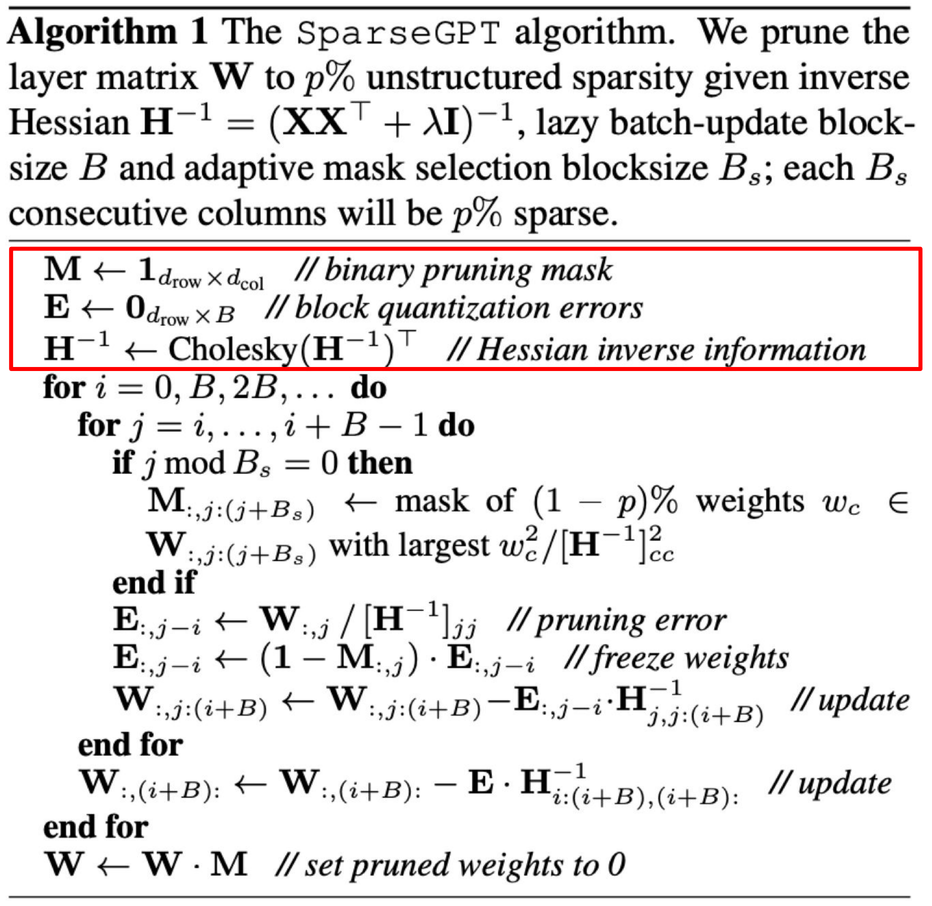
Algorithm: Adaptive Mask Selection

- Problems with fixed mask selection
  - The weight updates during reconstruction can give insight into creating a new mask
  - Research shows that using this capitalizing on this information has better results
- Problems with a column-wise heuristic
  - (e.g, choosing p% optimal weights in each column   to prune, that way overall sparsity is capped at p%)
  - However, we want non-uniform selection!
- Solution: Iterative Blocking
  - Mask preset blockwise, (Bs = 128 columns)
  - Mask generated using error formula & diagonals of hessian
  - Then the next block's weights are updated → repeat!
  - Non-uniform selection!

works for semi-structured pruning, n:m format! Let Bs = m, and choose the n weights with lowest error



Algorithm: Full Algorithm Pseudocode

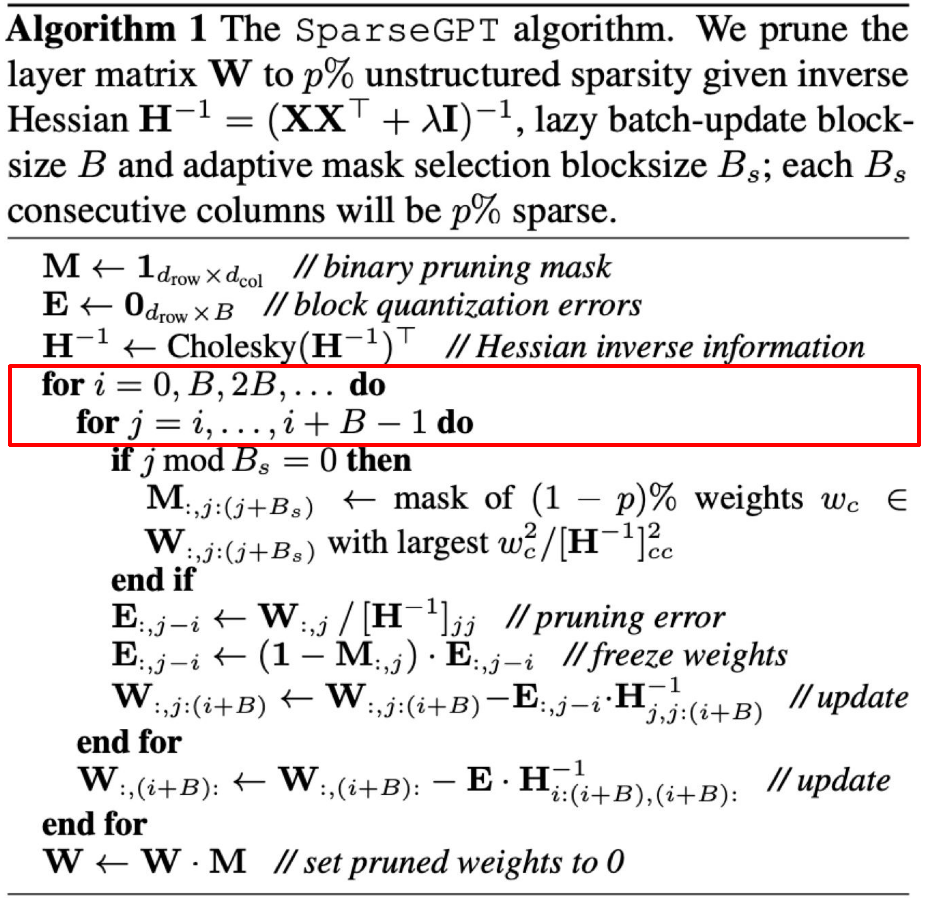


Algorithm 1 The `SparseGPT` algorithm. We prune the layer matrix $\mathbf{W}$ to $p\%$ unstructured sparsity given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$, lazy batch-update blocksize B and adaptive mask selection blocksize B_s; each B_s consecutive columns will be $p\%$ sparse.

```
$\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$ // binary pruning mask
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ // block quantization errors
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ // Hessian inverse information
for $i = 0, B, 2B, \ldots$ do
    for $j = i, \ldots, i + B - 1$ do
        if $j \bmod B_s = 0$ then
            $\mathbf{M}_{:,j:(j+B_s)} \leftarrow$ mask of $(1-p)\%$ weights $w_c \in \mathbf{W}_{:,j:(j+B_s)}$ with largest $w_c^2 / [\mathbf{H}^{-1}]_{cc}^2$
        end if
        $\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j} \,/\, [\mathbf{H}^{-1}]_{jj}$ // pruning error
        $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{1} - \mathbf{M}_{:,j}) \cdot \mathbf{E}_{:,j-i}$ // freeze weights
        $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ // update
    end for
    $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ // update
end for
$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}$ // set pruned weights to 0
```

- Build the pruning mask (0’s or 1’s)
- Store the block quantization errors
- Calculate the inverse Hessians
    - They use Cholesky Decomposition

Algorithm: Full Algorithm Pseudocode

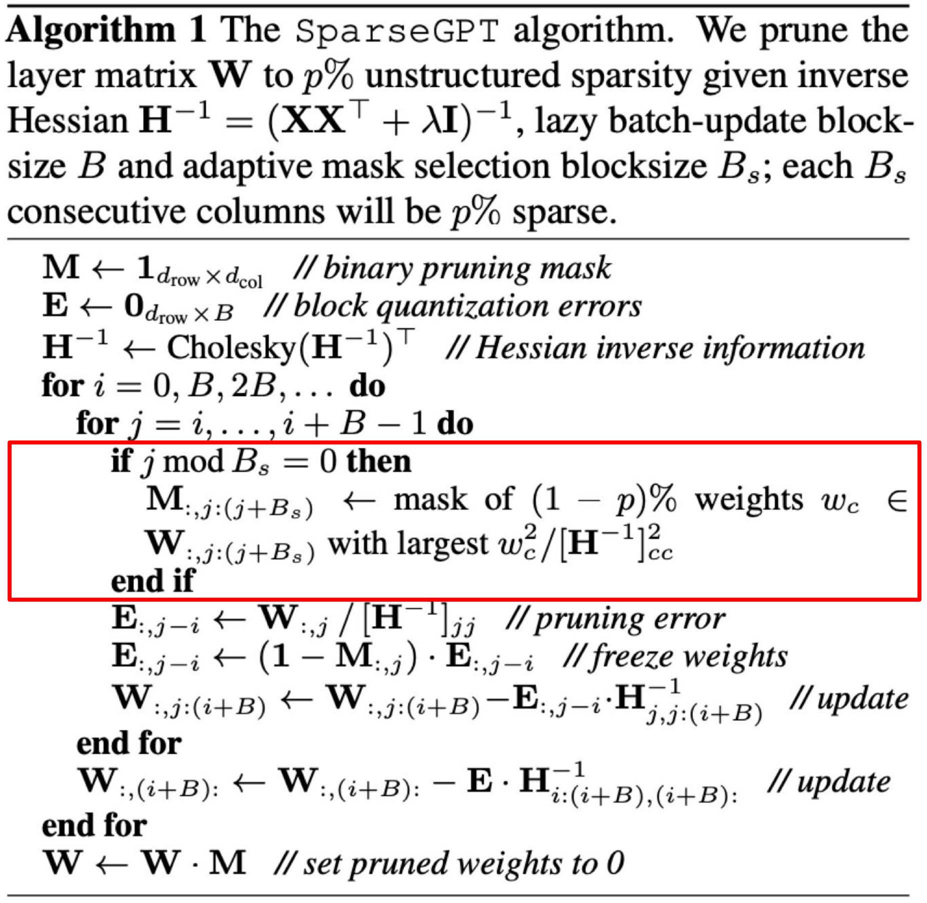


Algorithm 1 The `SparseGPT` algorithm. We prune the layer matrix $\mathbf{W}$ to $p\%$ unstructured sparsity given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$, lazy batch-update blocksize B and adaptive mask selection blocksize B_s; each B_s consecutive columns will be $p\%$ sparse.

```
$\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$   // binary pruning mask
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$   // block quantization errors
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$   // Hessian inverse information
for $i = 0, B, 2B, \ldots$ do
  for $j = i, \ldots, i + B - 1$ do
    if $j \bmod B_s = 0$ then
      $\mathbf{M}_{:,j:(j+B_s)} \leftarrow$ mask of $(1 - p)\%$ weights $w_c \in \mathbf{W}_{:,j:(j+B_s)}$ with largest $w_c^2 / [\mathbf{H}^{-1}]_{cc}^2$
    end if
    $\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j} \,/\, [\mathbf{H}^{-1}]_{jj}$   // pruning error
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{1} - \mathbf{M}_{:,j}) \cdot \mathbf{E}_{:,j-i}$   // freeze weights
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$   // update
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$   // update
end for
$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}$   // set pruned weights to 0
```

- The nested loop controls a lot of the algorithm
- Outer loop iterates through blocks
- Inner loop iterates through a block’s columns

Algorithm: Full Algorithm Pseudocode

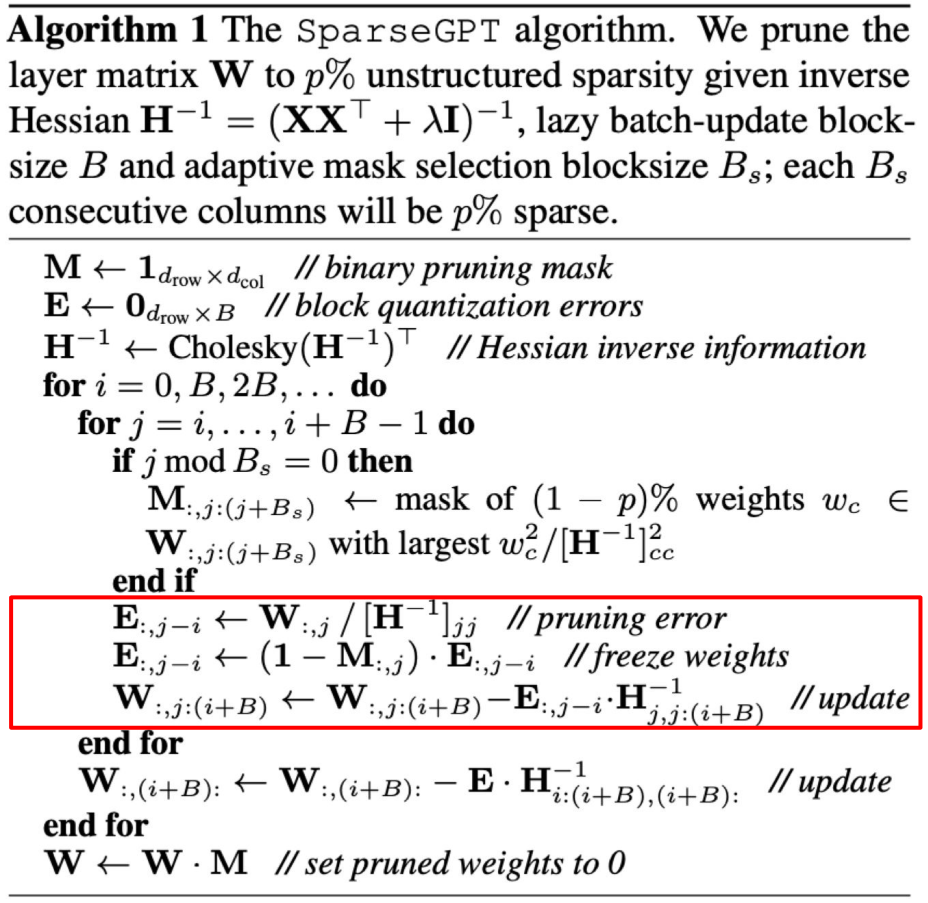


Algorithm 1 The `SparseGPT` algorithm. We prune the layer matrix $\mathbf{W}$ to $p\%$ unstructured sparsity given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$, lazy batch-update blocksize B and adaptive mask selection blocksize B_s; each B_s consecutive columns will be $p\%$ sparse.

```
$\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$   // binary pruning mask
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$   // block quantization errors
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$   // Hessian inverse information
for $i = 0, B, 2B, \ldots$ do
  for $j = i, \ldots, i + B - 1$ do
    if $j \bmod B_s = 0$ then
      $\mathbf{M}_{:,j:(j+B_s)} \leftarrow$ mask of $(1-p)\%$ weights $w_c \in \mathbf{W}_{:,j:(j+B_s)}$ with largest $w_c^2 / [\mathbf{H}^{-1}]_{cc}^2$
    end if
    $\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j} \,/\, [\mathbf{H}^{-1}]_{jj}$   // pruning error
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{1} - \mathbf{M}_{:,j}) \cdot \mathbf{E}_{:,j-i}$   // freeze weights
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$   // update
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$   // update
end for
$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}$   // set pruned weights to 0
```

- Checks if current column is start of new block
- If so, then updates the pruning mask for the next block of columns
- The pruning mask has the top (1-p)% weights set to 1 and rest 0 (for p% sparsity)
  - Selects weights by importance, according to the inverse hessian

Algorithm: Full Algorithm Pseudocode



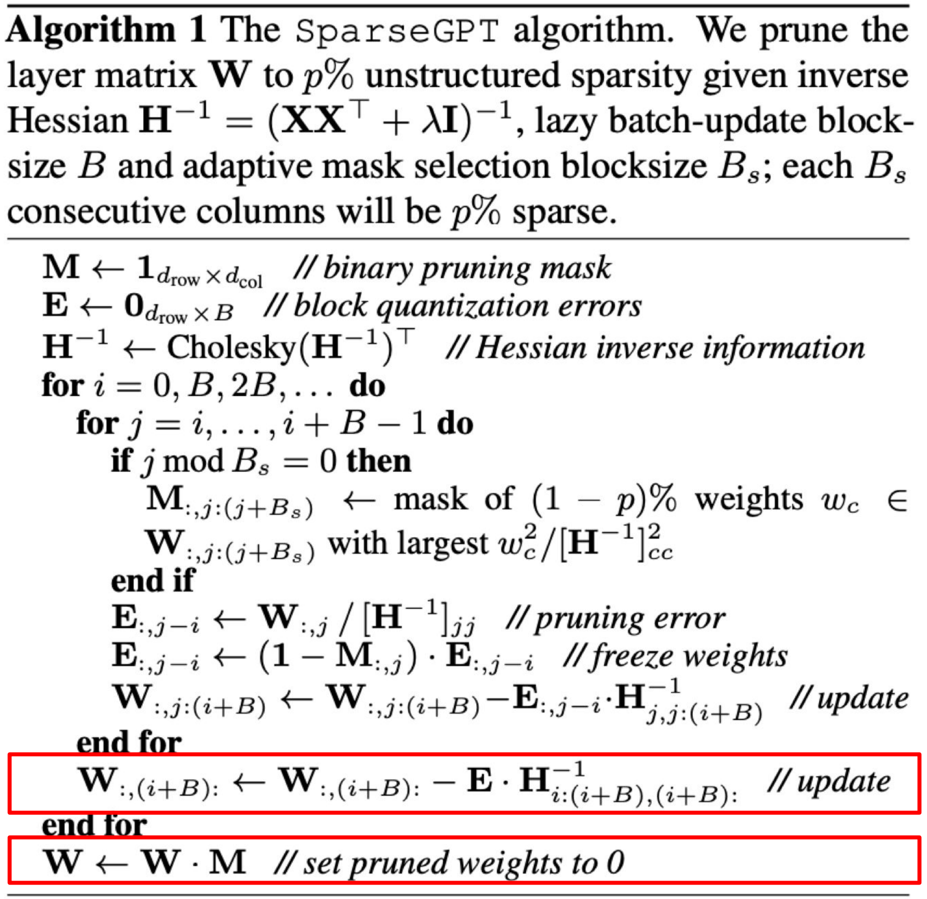
Algorithm 1 The `SparseGPT` algorithm. We prune the layer matrix $\mathbf{W}$ to $p\%$ unstructured sparsity given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$, lazy batch-update blocksize B and adaptive mask selection blocksize B_s; each B_s consecutive columns will be $p\%$ sparse.

```
$\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$   // binary pruning mask
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$   // block quantization errors
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$   // Hessian inverse information
for $i = 0, B, 2B, \ldots$ do
  for $j = i, \ldots, i + B - 1$ do
    if $j \bmod B_s = 0$ then
      $\mathbf{M}_{:,j:(j+B_s)} \leftarrow$ mask of $(1-p)\%$ weights $w_c \in \mathbf{W}_{:,j:(j+B_s)}$ with largest $w_c^2 / [\mathbf{H}^{-1}]_{cc}^2$
    end if
    $\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j} \,/\, [\mathbf{H}^{-1}]_{jj}$   // pruning error
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{1} - \mathbf{M}_{:,j}) \cdot \mathbf{E}_{:,j-i}$   // freeze weights
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$   // update
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$   // update
end for
$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}$   // set pruned weights to 0
```

- Find the pruning error for each of the columns and store it in the E matrix
- Freeze the weights that were pruned in the E matrix, so they can't contribute to error
- Update the weights of the block using the inverse hessian

- (updates columns to the right of the current)

Algorithm: Full Algorithm Pseudocode



Algorithm 1 The `SparseGPT` algorithm. We prune the layer matrix $\mathbf{W}$ to $p\%$ unstructured sparsity given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}$, lazy batch-update blocksize B and adaptive mask selection blocksize B_s; each B_s consecutive columns will be $p\%$ sparse.

```
$\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$   // binary pruning mask
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$   // block quantization errors
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$   // Hessian inverse information
for $i = 0, B, 2B, \ldots$ do
  for $j = i, \ldots, i + B - 1$ do
    if $j \bmod B_s = 0$ then
      $\mathbf{M}_{:,j:(j+B_s)} \leftarrow$ mask of $(1-p)\%$ weights $w_c \in \mathbf{W}_{:,j:(j+B_s)}$ with largest $w_c^2 / [\mathbf{H}^{-1}]_{cc}^2$
    end if
    $\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j} \,/\, [\mathbf{H}^{-1}]_{jj}$   // pruning error
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{1} - \mathbf{M}_{:,j}) \cdot \mathbf{E}_{:,j-i}$   // freeze weights
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$   // update
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$   // update
end for
$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}$   // set pruned weights to 0
```

- Updates the remaining weights in all the other blocks, using the accumulated pruning errors
  - (Once again, adjusted by inverse hessian)
- Finally, we multiply the mask by the weights to prune, setting specified weights to 0

Experiments

Pytorch to make Sparse-GPT, Huggingface for models and datasets, and an 80GB NVIDIA A100

Models, Datasets, Evaluation

- OPT family and 176B Bloom
- mainly perplexity, some ZeroShot accuracy
  - Raw-WikiText2
  - Penn Treebank (PTB)
  - Subset of C4 validation data
  - Lambada
  - ARC (Easy and Challenge)
  - PIQA
  - StoryCloze
- relative sparse v.s dense accuracy

Baselines

- Standard magnitude pruning (layer-wise)
- AdaPrune for models up to 1 billion parameters
  - Memory-optimized version
  - Tuned hyper-parameters → 3× speedup with no impact on solution quality

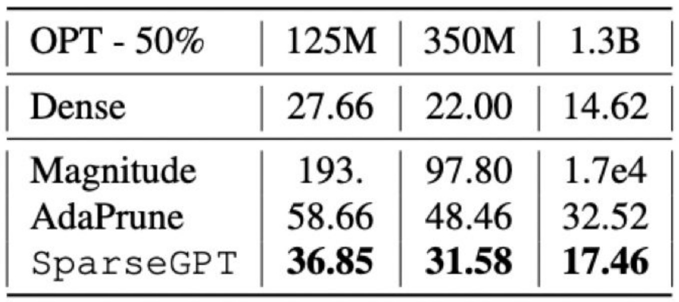
Results: Pruning V.S Model Size

- **Takeaway: “Larger models are easier to sparsify”**

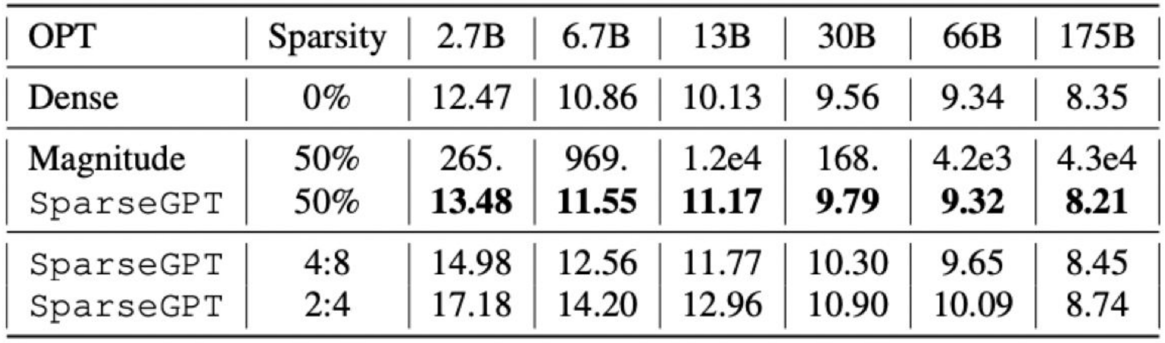
- Magnitude based pruning approaches have sinking accuracies
  - Accurate pruning approaches are a must as size increases

- Sparse-GPT shows best perplexity loss, even at large sizes
  - AdaPrune shows improvement over magnitude pruning, but not comparable with Sparse-GPT

Table 1. OPT perplexity results on raw-WikiText2.



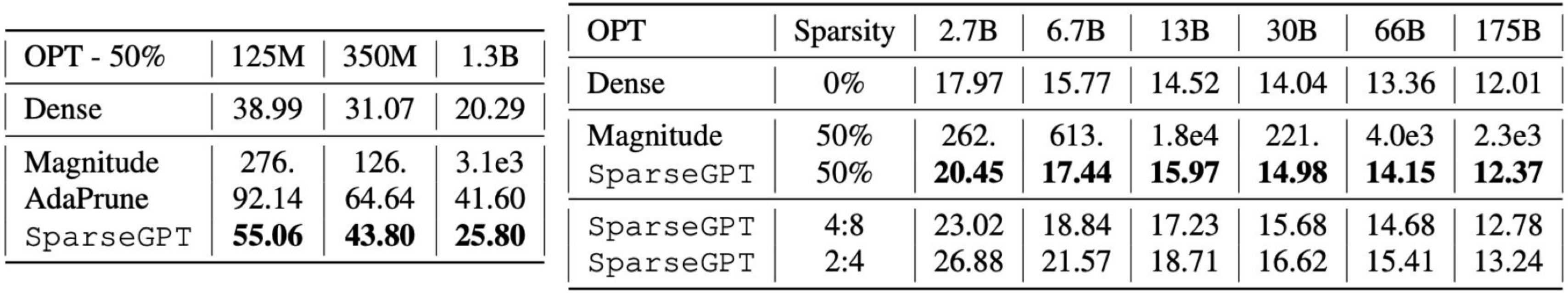
OPT - 50%	125M	350M	1.3B
Dense	27.66	22.00	14.62
Magnitude	193.	97.80	1.7e4
AdaPrune	58.66	48.46	32.52
SparseGPT	**36.85**	**31.58**	**17.46**



OPT	Sparsity	2.7B	6.7B	13B	30B	66B	175B
Dense	0%	12.47	10.86	10.13	9.56	9.34	8.35
Magnitude	50%	265.	969.	1.2e4	168.	4.2e3	4.3e4
SparseGPT	50%	**13.48**	**11.55**	**11.17**	**9.79**	**9.32**	**8.21**
SparseGPT	4:8	14.98	12.56	11.77	10.30	9.65	8.45
SparseGPT	2:4	17.18	14.20	12.96	10.90	10.09	8.74

Results: Pruning V.S Model Size

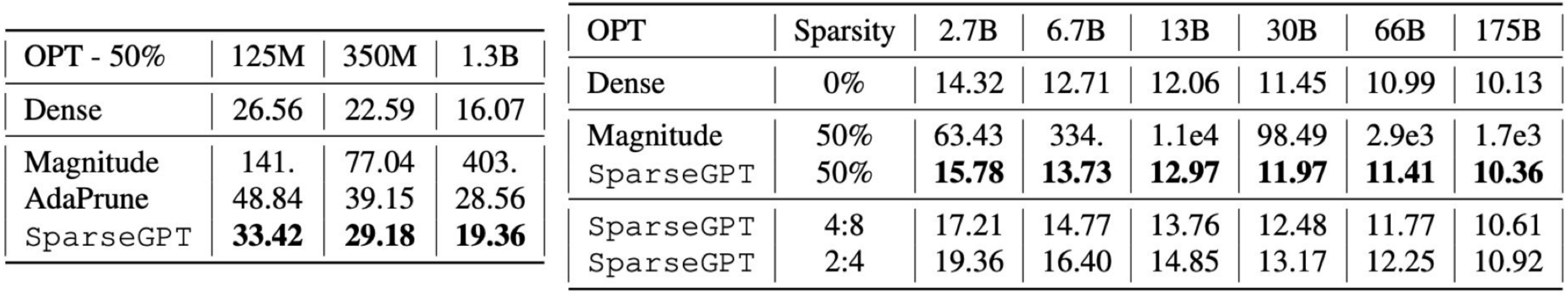
Table 3. OPT perplexity results on PTB.



OPT - 50%	125M	350M	1.3B
Dense	38.99	31.07	20.29
Magnitude	276.	126.	3.1e3
AdaPrune	92.14	64.64	41.60
SparseGPT	**55.06**	**43.80**	**25.80**

OPT	Sparsity	2.7B	6.7B	13B	30B	66B	175B
Dense	0%	17.97	15.77	14.52	14.04	13.36	12.01
Magnitude	50%	262.	613.	1.8e4	221.	4.0e3	2.3e3
SparseGPT	50%	**20.45**	**17.44**	**15.97**	**14.98**	**14.15**	**12.37**
SparseGPT	4:8	23.02	18.84	17.23	15.68	14.68	12.78
SparseGPT	2:4	26.88	21.57	18.71	16.62	15.41	13.24

Table 4. OPT perplexity results on a C4 subset.



OPT - 50%	125M	350M	1.3B
Dense	26.56	22.59	16.07
Magnitude	141.	77.04	403.
AdaPrune	48.84	39.15	28.56
SparseGPT	**33.42**	**29.18**	**19.36**

OPT	Sparsity	2.7B	6.7B	13B	30B	66B	175B
Dense	0%	14.32	12.71	12.06	11.45	10.99	10.13
Magnitude	50%	63.43	334.	1.1e4	98.49	2.9e3	1.7e3
SparseGPT	50%	**15.78**	**13.73**	**12.97**	**11.97**	**11.41**	**10.36**
SparseGPT	4:8	17.21	14.77	13.76	12.48	11.77	10.61
SparseGPT	2:4	19.36	16.40	14.85	13.17	12.25	10.92

Results: Sparsity Scaling for 100B+ parameters

Takeaway: Sparse-GPT can remove 100 billion weights on LLMs w/o much loss

- Magnitude based pruning falls apart once again at high sparsity
- SparseOPTs at higher sparsity perform similarly to a magnitude based pruning approach at lower sparsity
- OPT-175B and BLOOM-176B can achieve up to 60% sparsity w/o much perplexity loss

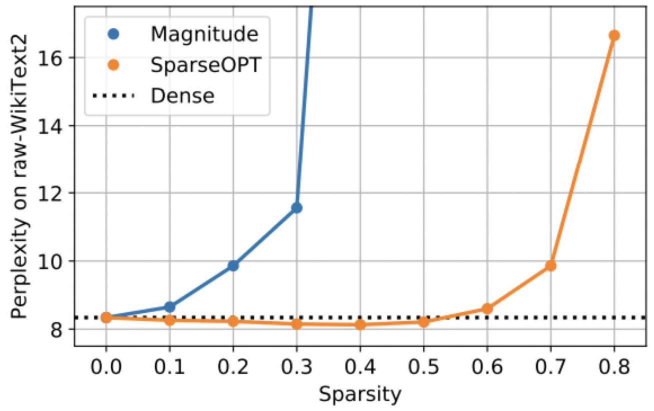


Figure 1. Sparsity-vs-perplexity comparison of `SparseGPT` against magnitude pruning on OPT-175B, when pruning to different *uniform* per-layer sparsities.

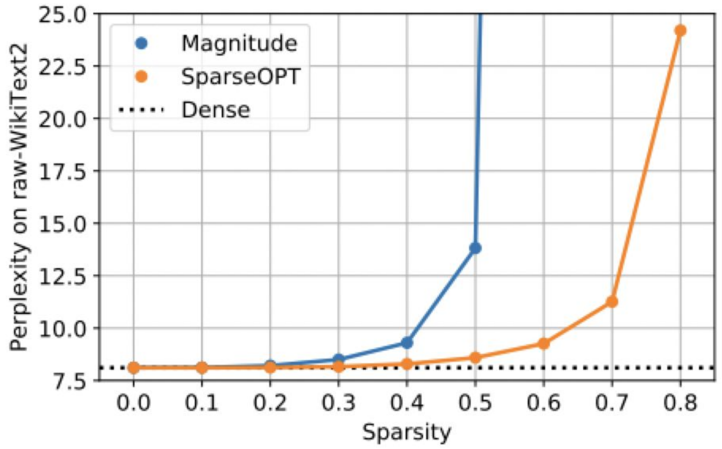


Figure 5. Uniform pruning BLOOM-176B.

Results: Zero Shot Experiments

Takeaway: Sparse-GPT retains accuracy on zero-shot tasks

- Magnitude based pruning sticks to the expected theme
- Sparse-GPT models stay close to original accuracy (around 70%)
- Researchers point out that the “numbers are more noisy” as 2:4 performs better than dense on Lambada
    - However, these unexpected results average out across tasks

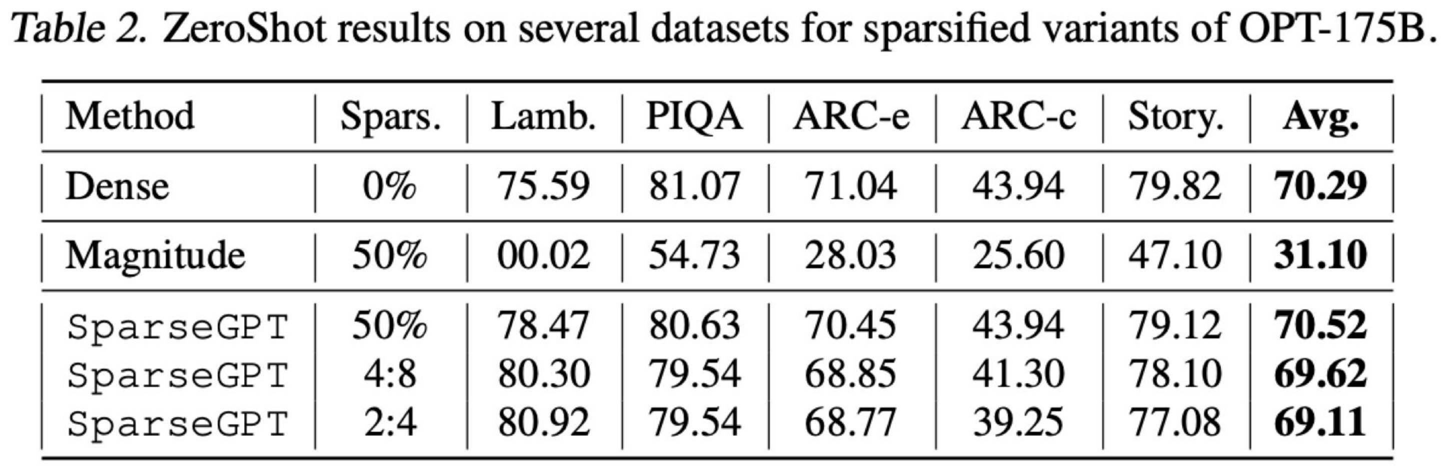


Table 2. ZeroShot results on several datasets for sparsified variants of OPT-175B.

Method	Spars.	Lamb.	PIQA	ARC-e	ARC-c	Story.	Avg.
Dense	0%	75.59	81.07	71.04	43.94	79.82	**70.29**
Magnitude	50%	00.02	54.73	28.03	25.60	47.10	**31.10**
SparseGPT	50%	78.47	80.63	70.45	43.94	79.12	**70.52**
SparseGPT	4:8	80.30	79.54	68.85	41.30	78.10	**69.62**
SparseGPT	2:4	80.92	79.54	68.77	39.25	77.08	**69.11**

Results: Joint Sparsification and Quantization

Takeaway: Sparse-GPT can be combined with Quantization approaches!

- We compare 50% + 4bit models with their memory equivalent, 3bit GPTQ, to see if there's a benefit
- Sparse GPT 50% + 4bit models are more accurate than 3 bit GPTQ models for >= 2.7B params

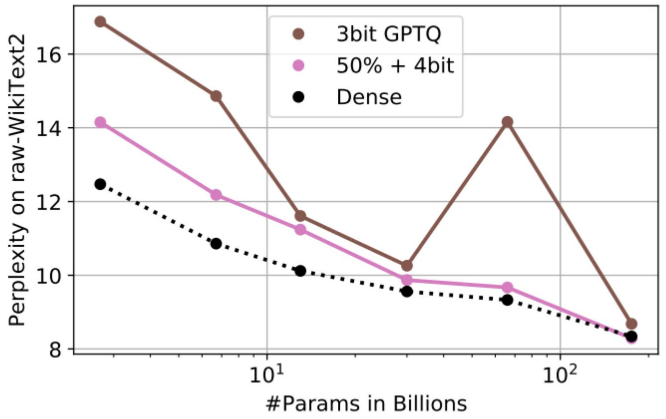


Figure 6. Comparing joint 50% sparsity + 4-bit quantization with size-equivalent 3-bit on the OPT family for $\geq$ 2.7B params.

Results: Sensitivity and Partial N:M sparsity

Takeaway: “Sparse-GPT can generate models (increasing 2:4 sparsity) in 1 pruning pass”

- Later layers in Sparse-GPT are more sensitive than earlier ones
- As it turns out, skipping leaving last third of the model alone returns the best accuracy
- Combining the first n modified layers with the total-n original, Sparse-GPT demonstrates partial N:M sparsity

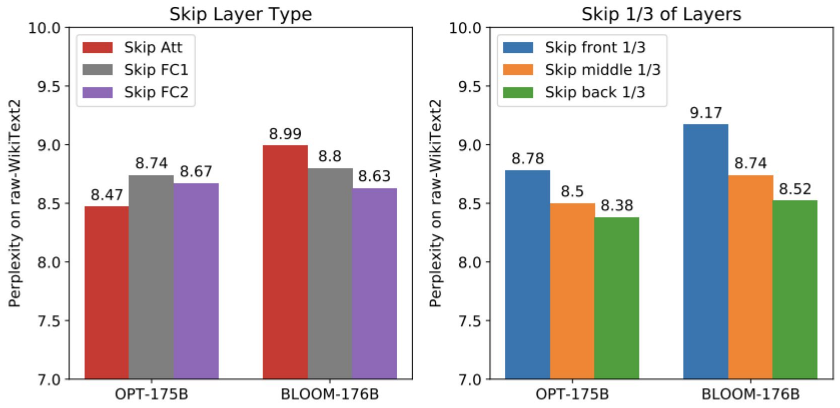


Figure 7. Sensitivity results for partial 2:4 pruning.

Discussion: What can we learn?

Sparse-GPT:

- Novel post training pruning method
- Contribution to NLP: Showing that large (GPT) family models can be compressed in one-shot, no retraining necessary, and with not much loss in accuracy
    - They substantiate these claims by measuring perplexity and zero-shot accuracy
- Further insights
    - Show that LLMs can reach 50-60% sparsity while holding on to accuracy, even dropping 100B+ weights
        - Relative accuracy drop lowers as model size increases….this is big!
    - Larger models are easier to parametrize
    - Accuracy on 0-shot tasks
    - Can be combined with other compression methods

Limitations + Future Work

There are a couple of open questions and motivations that prompt further research

Why are larger models easier to sparsify?

- One obvious speculation is overparameterization
- Nonetheless, the researchers claim that a study on this observation would be a good next direction

Sparse-GPT was mostly evaluated on perplexity, or accuracy on downstream tasks

- How would it fare for open-ended text generation tasks

Sparsity acceleration is a promising and practical approach for future research

- Researchers took preliminary steps to show efficacy of sparse model acceleration, but stress that more work is needed

Questions!