Small Language Models

Sowmya Vajjala
National Research Council, Canada
Applications of Language Models, Spring 2025, IIIT-H

Two Questions

- How do we reduce the size of large language models?
- How do we go about building small language models?

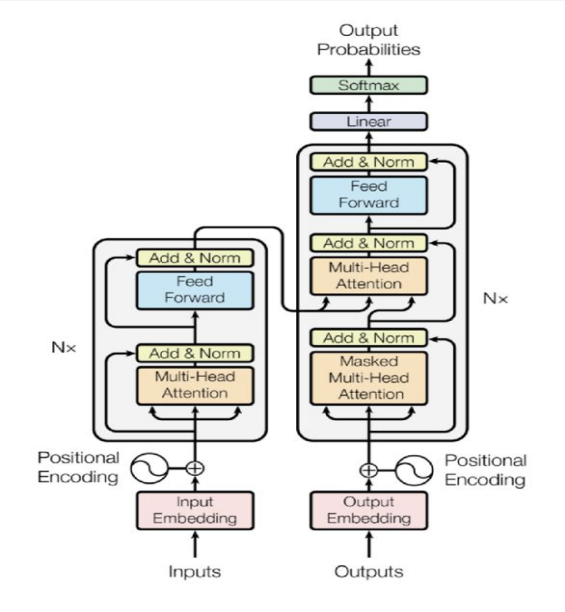
What are small LMs?

- For the rest of this class, I will use “small LM” to mean any language model <10B parameters in size.

- **For the record**: bert-base is ~110M parameters, modernbert-base (2024) is ~150M parameters. GPT4 is expected to have >1.5 Trillion parameters

But, before we get started: What are the “parameters”? How do we calculate them, though?

What is a “model” made up of?




```
>>> from transformers import AutoModel, AutoTokenizer
>>> model = AutoModel.from_pretrained("bert-base-uncased")
>>> model
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
      (0-11): 12 x BertLayer(
        (attention): BertAttention(
          (self): BertSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
          (intermediate_act_fn): GELUActivation()
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
  )
  (pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
  )
)
```

Estimating a model's size

Bert-base-uncased	Key	Shape	Count	
Embedding	embeddings.word_embeddings.weight	[30522, 768]	23,440,896	23,837,184
	embeddings.position_embeddings.weight	[512, 768]	393,216	
	embeddings.token_type_embeddings.weight	[2, 768]	1,536	
	embeddings.LayerNorm.weight	[768]	768	
	embeddings.LayerNorm.bias	[768]	768	

Transformer * 12	encoder.layer.0.attention.self.query.weight	[768, 768]	589,824	7,087,872 * 12 = 85,054,464
	encoder.layer.0.attention.self.query.bias	[768]	768	
	encoder.layer.0.attention.self.key.weight	[768, 768]	589,824	
	encoder.layer.0.attention.self.key.bias	[768]	768	
	encoder.layer.0.attention.self.value.weight	[768, 768]	589,824	
	encoder.layer.0.attention.self.value.bias	[768]	768	
	encoder.layer.0.attention.output.dense.weight	[768, 768]	589,824	
	encoder.layer.0.attention.output.dense.bias	[768]	768	
	encoder.layer.0.attention.output.LayerNorm.weight	[768]	768	
	encoder.layer.0.attention.output.LayerNorm.bias	[768]	768	

Source: https://github.com/google-research/bert/issues/656

Continued ..

	encoder.layer.0.intermediate.dense.weight	[3072, 768]	2,359,296	
	encoder.layer.0.intermediate.dense.bias	[3072]	3072	
	encoder.layer.0.output.dense.weight	[768, 3072]	2,359,296	
	encoder.layer.0.output.dense.bias	[768]	768	
	encoder.layer.0.output.LayerNorm.weight	[768]	768	
	encoder.layer.0.output.LayerNorm.bias	[768]	768	
Pooler	pooler.dense.weight	[768, 768]	589,824	590,592
	pooler.dense.bias	[768]	768	
				109,482,240

Note: You can use model.num_parameters() for any model in huggingface transformers library - these slides are here just to show what is under the hood!

Q1: How do we reduce the size of large language models?

Why reduce the size of LLMs?

LLMs are large, and they are large for a reason. Why make them small?

- Reduce memory footprints of the model
- Have a manageable size for reasonable latency during inference time.
- LLMs are over-parametrized and under-trained. It is better to keep what is needed and discard the rest, to make them more efficient.

Making Larger Models Smaller

Three common approaches:

- **Quantization**: keep the model same, but reduce the precision (number of bits)
- **Pruning**: remove parts of the model that do not play a role in predictive performance
- **Distillation**: train a smaller (student) model to imitate a larger (teacher) model

I will focus on quantization (as that is more common)

What is Quantization?

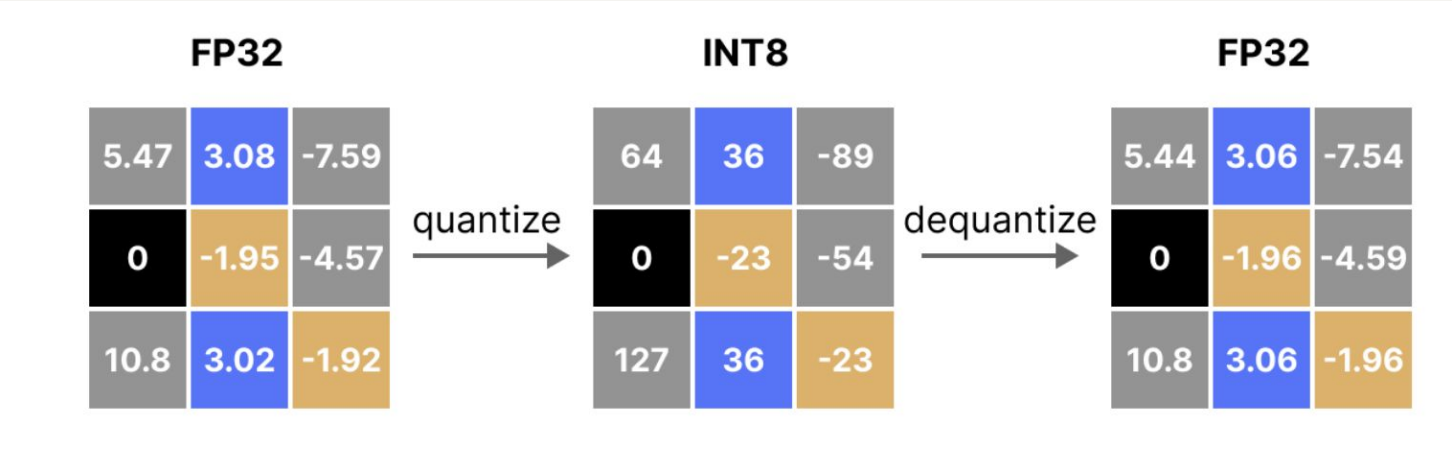
- Consider a 70B LLM. Most models are natively representing data with FP32.
- 70B parameters, each represented with 32 bits (i.e., 4 bytes) results in 280 GB size just to load the model.
- Let us say we reduce the precision of the parameter representation to 8 bits instead, we see a four fold reduction in the memory requirement too by the same amount.
- E.g., a 7B parameter LLM, with 4-bit precision, will need less than 4GB RAM to load. This is what we call Quantization.

(Note: memory requirements for inference are different - we are only talking about loading an LLM here).

A Brief Note on Data Types

- Uint8: unsigned 8-bit integer - we can represent 0 to 255 in binary notation (torch.uint8)
- Signed 8 bit integer - we can represent -128 to + 127 (torch.int8)
- Similarly, in torch, we have 16-bit and 32-bit integer representations as well.
- When it comes to floating point, there are bf16, fp16, **fp32**, fp64 representations in pytorch. (fp32 is more commonly used by LLMs)

Quantization Error

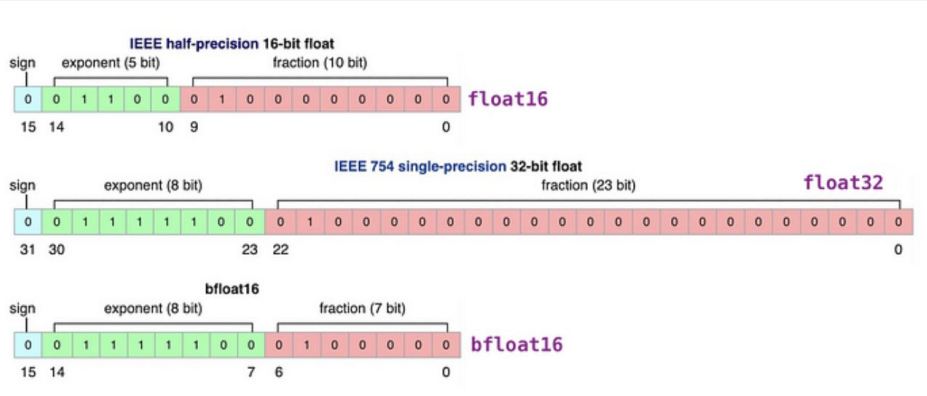


Our Goal: As less quantization error as possible (image: source)

Anatomy of a floating point representation

- Sign
- Exponent (range)
- Fraction (precision)

The different floating point representations in pytorch differ on how they represent the last two.



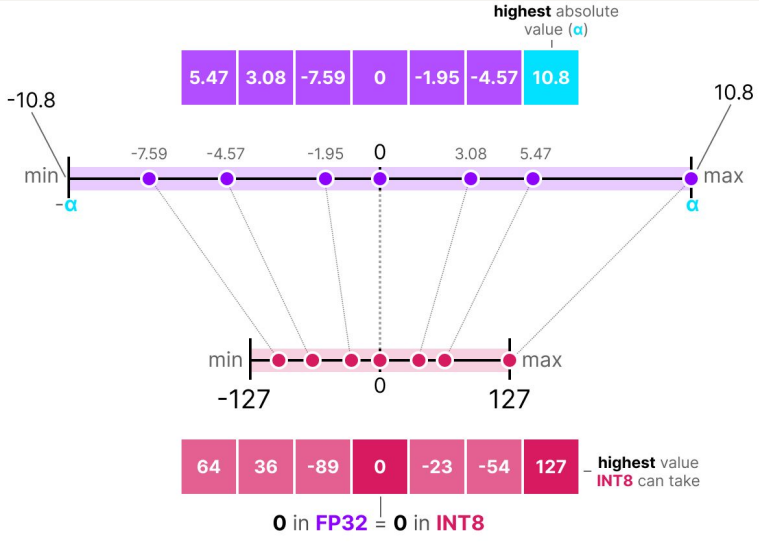
source

Downcasting

- A simple way to quantize a model is downcasting. An fp32 model can become an fp16 one by calling model.to(torch.float16) (Model size reduces to half)
- In downcasting, we convert the model's parameters to a more compact data type (bfloat16). During inference, the model performs its calculations in this data type, and its activations are in this data type.
- Downcasting like this reduces memory requirements, and increases speed, but will degrade the performance with a smaller data type, and won't work if you convert to an integer data type (like the int8 in the example).

Float to Int Conversion - Absmax

- Let us say we are converting from FP32 to Int8. Absmax is all about taking the absolute maximum from the FP32 weights and make it the range to symmetrically map to int8.



Note: Quantization can be asymmetric too.

source

Linear Quantization

- It enables the quantized model to maintain performance much closer to the original model by converting from the compressed data type back to the original FP32 during inference.
- So, when the model makes a prediction, it is performing the matrix multiplications in FP32, and the activations are in FP32. This enables you to quantize the model in data types smaller than bfloat16, such as int8.
- Although this is resource intensive, quantization itself can be called in one line of code today. (e.g., quantize() function in quanto library)

In Practice: An Example

```
model =AutoModelForCausalLM.from_pretrained( "bigscience/bloom-560m")
tokenizer = AutoTokenizer.from_pretrained( "bigscience/bloom-560m")
print("Before quantization")
print(model.transformer.h[ 0].self_attention.query_key_value.weight)
quantize(model, weights="qint8", activations="qint8", device="mps")
freeze(model)
print("After quantization")
print(model.transformer.h[ 0].self_attention.query_key_value.weight)
```

```
Before quantization
Parameter containing:
tensor([[-2.7695e-02,  1.8280e-02, -2.2247e-02,  ..., -4.2610e-03,
         -2.3834e-02, -1.3916e-02],
        [ 3.0079e-03,  2.6703e-02,  2.8641e-02,  ...,  9.5901e-03,
          8.7662e-03, -1.3481e-02],
        [ 1.3985e-02, -1.2695e-02, -3.7479e-03,  ...,  1.8539e-03,
          1.9577e-02, -4.4365e-03],
        ...,
        [ 7.4730e-03, -6.9962e-03, -1.2009e-02,  ..., -5.8174e-04,
          7.0989e-05, -7.3738e-03],
        [ 2.2812e-02, -5.9624e-03,  1.5427e-02,  ...,  2.8706e-03,
         -1.0281e-03,  1.0826e-02],
        [-4.9248e-03, -8.2245e-03, -2.6550e-03,  ..., -8.4991e-03,
         -1.9409e-02, -1.7303e-02]], requires_grad=True)
After quantization
QTensor(tensor([[-46,  30, -37,  ...,  -7, -39, -23],
        [  5,  41,  44,  ...,  15,  13, -21],
        [ 25, -23,  -7,  ...,   3,  35,  -8],
        ...,
        [ 22, -21, -36,  ...,  -2,   0, -22],
        [ 66, -17,  45,  ...,   8,  -3,  31],
        [-15, -26,  -8,  ..., -27, -61, -54]], dtype=torch.int8), scale=tensor([[0.0006],
        [0.0007],
        [0.0006],
        ...,
        [0.0003],
        [0.0003],
        [0.0003]]), public_dtype=torch.float32, requires_grad=True)
```

I used Quanto, but there are other libraries with different quantization approaches such as GPTQ, Bits and Bytes, AWQ etc.

Current Practices

- 8 bit quantized versions of larger LLMs are commonly seen.
- However, nothing prevents us from using 4-bit, 2-bit etc.
- There is even work on 1-bit quantization approaches (BitNet, OneBit, BitNet b1.58 etc)
- However, beyond 8 bits, we see quite a bit of performance degradation, and it is generally not useful in most application scenarios.
- Downsides: Quantization may make models more susceptible to jail breaks (Kumar et.al., 2024), increase calibration errors (Proskurina et.al. 2024), affect multilingual performance and reasoning abilities (Marchisio et.al., 2024) etc.

Embeddings Quantization

- Quantization can happen for embedding models as well, as a post-processing step.
- Why is it needed?:
  - Embedding representations are higher-dimensional, and a lot of real-world applications rely on calculating similarities between embedding vectors on the fly (e.g., search, recommendation systems, outlier detection etc)
  - So, scaling their use to production use cases may lead to expensive, high-latency solutions.

Will quantization help in such cases?

https://huggingface.co/blog/embedding-quantization

Embeddings Quantization - How?

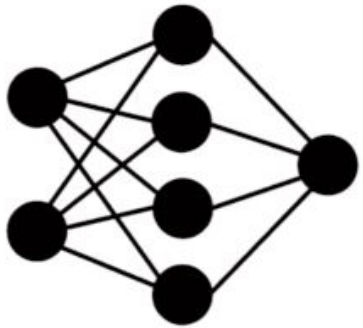
- ➔ Unlike quantization in models where you reduce the precision of weights, quantization for embeddings refers to a post-processing step for the embeddings themselves.
- ➔ Binary quantization in this case refers to the conversion of the float32 values in an embedding to 1-bit values, resulting in a 32x reduction in memory and storage usage.
- ➔ Hamming Distance (number of positions at which the bits of two binary embeddings differ) is used to retrieve these binary embeddings efficiently.
- ➔ How is it used in practice?: Use binarized embeddings for retrieval, and rescore the retrieved list with the regular float32 embedding.

https://huggingface.co/blog/embedding-quantization

Dimensionality Reduction for Embeddings

- Another way of looking at increasing speed without compromising on performance with embeddings is dimensionality reduction.
- However, traditional approaches such as PCA are not known to do well with embeddings.
- Some recent work proposed embedding representations that can be truncated easily, where the vector is somehow ordered i.e., earlier dimensions are more important than later dimensions.
- Matryoshka embedding models are a method to produce embeddings of various sizes where truncation of the embedding vectors won’t result in major changes in performance.

Matroyshka Embedding



1. Compute Matryoshka Embedding

https://huggingface.co/blog/matryoshka

How does it work?

- When we train an embedding model, we follow the typical process where the model produces embeddings for a batch, and an optimizer adjusts weights of the network to minimize the loss.
- For Matryoshka Embedding models, the loss function aims to determine not just the quality of your full-size embeddings, but also the quality of your embeddings at various different dimensionalities.
- In practice, this incentivizes the model to frontload the most important information at the start of an embedding, such that it will be retained if the embedding is truncated.

What is the advantage, though?

```python
from sentence_transformers import SentenceTransformer
from sentence_transformers.util import cos_sim
model = SentenceTransformer("tomaarsen/mpnet-base-nli-matryoshka")
matryoshka_dim = [64,128,256,512,1024]
embeddings = model.encode(["The weather is so nice!", "It's so sunny outside!",
       "He drove to the stadium."])
for dimension in matryoshka_dim:
   print(f"Dimension: {dimension}")
   shrinked_embeddings = embeddings[..., :dimension]    # Shrink the embedding dimensions
   print(embeddings.shape)
   # Similarity of the first sentence to the other two:
   similarities = cos_sim(shrinked_embeddings[0], shrinked_embeddings[1:])
   print(similarities)
```

```
Dimension: 64
(3, 64)
tensor([[0.8910, 0.1337]])
Dimension: 128
(3, 128)
tensor([[0.8765, 0.1413]])
Dimension: 256
(3, 256)
tensor([[0.8522, 0.0917]])
Dimension: 512
(3, 512)
tensor([[0.8408, 0.1078]])
Dimension: 1024
(3, 768)
tensor([[0.8428, 0.0873]])
```

Retrieval can be much faster!

Quantization - Summary

- Quantization reduces the precision, but won't change or remove any parameters.
- 8 bit quantization is a popular approach
- There are existing libraries that support quantization of language models.
- We also quantize embeddings and perform dimensionality reduction over them, which is useful in retrieval, text classification kind of applications.

Pruning

- Idea: Remove irrelevant parameters from the model after training.
- Why?: to reduce computation, memory and hardware requirements.
- In literature, we see methods that apply pruning during pre-training, fine-tuning, or even during inference phases.
- We typically need further fine-tuning to recover the performance loss.

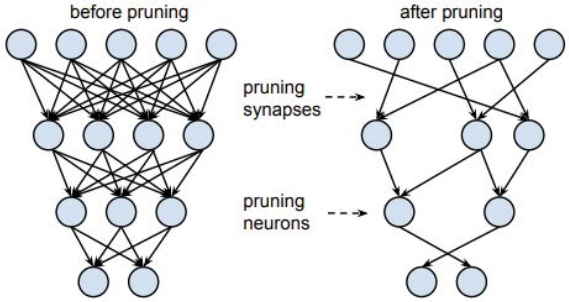


Figure 3: Synapses and neurons before and after pruning.

source

Treviso et.al.(2023)

How is Pruning done?

- Unstructured Pruning: Zero-out all parameters below a threshold.
  - E.g., Magnitude Pruning: weights with the smallest absolute values are set to zero.

- Structured Pruning: Remove specific components (e.g., attention heads, hidden layers etc)

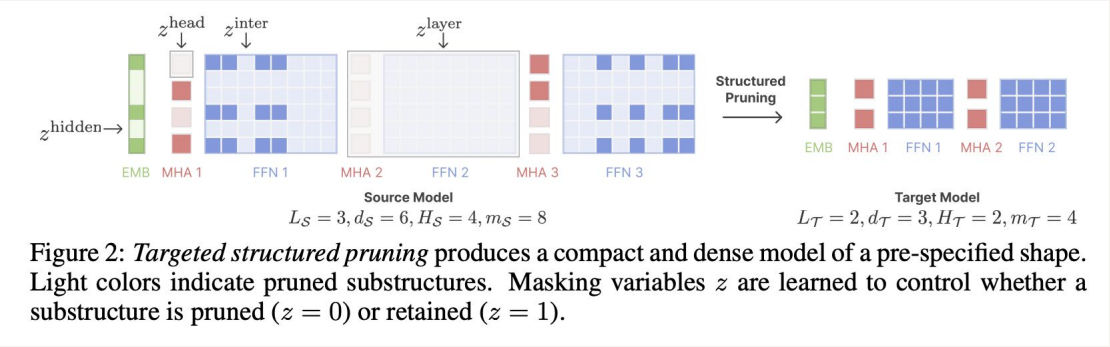


Figure 2: *Targeted structured pruning* produces a compact and dense model of a pre-specified shape. Light colors indicate pruned substructures. Masking variables z are learned to control whether a substructure is pruned ($z = 0$) or retained ($z = 1$).

Source: ShearedLLama paper, ICLR 2024

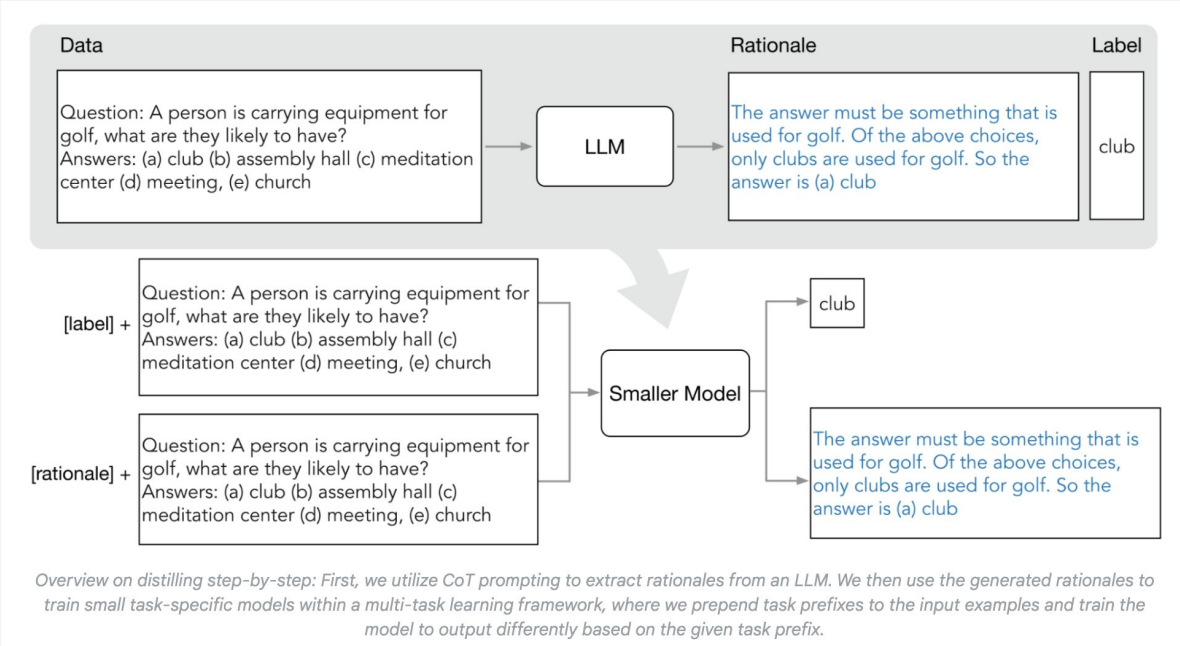
- Unstructured pruning preserves performance, whereas structured pruning improves speed.

Distillation

- Idea: Train one (student) model to replicate the behavior of another (teacher) model
- What happens?: often leads to the student outperforming a similarly sized model trained without this supervision.
- Different ways:
  - Distilling larger (generic) models to build task-specific models
  - Distilling pre-trained models to make them better with several downstream tasks.
- Challenges:
  - Added cost of learning hyper-parameters for the student model
  - Ensuring no loss in performance or efficiency.

Treviso et.al.(2023)

How to Distil - Distilling Step by Step



Overview on distilling step-by-step: First, we utilize CoT prompting to extract rationales from an LLM. We then use the generated rationales to train small task-specific models within a multi-task learning framework, where we prepend task prefixes to the input examples and train the model to output differently based on the given task prefix.

Source: Google Blog

Summary

- Quantization: no parameters are changed, up to k bits of precision
- Pruning: some parameters are set to zero, the rest are unchanged
- Distillation: Almost all parameters are changed!

Q2: How do we go about building small language models?

Why Small Models?

- If we can build a large model and reduce its size, why build other small models?

- Smaller models can potentially be useful in a range of scenarios:
  - Local deployment and more focused use cases.
  - Edge deployment
  - Other resource constrained scenarios
  - Assisted generation/speculative decoding
  - Faster inference

Speculative Decoding

- LLMs are faster at confirming that they would generate a certain sequence than they are at generating that sequence themselves
- Small models that are trained using the same tokenizer and in a similar fashion to a larger model can be used to quickly generate candidate sequences aligned with the large model,
- And large model can validate and accept as its own generated text.
- In practice: Gemma 2 2B can be used for assisted generation with the pre-existing Gemma 2 27B model.

What are some small LMs?

- Phi-series from Microsoft
- OpenELM series from Apple
- SmolLM series from Huggingface
- MobileLLM from Meta
- Ministral models from Mistral AI

Etc

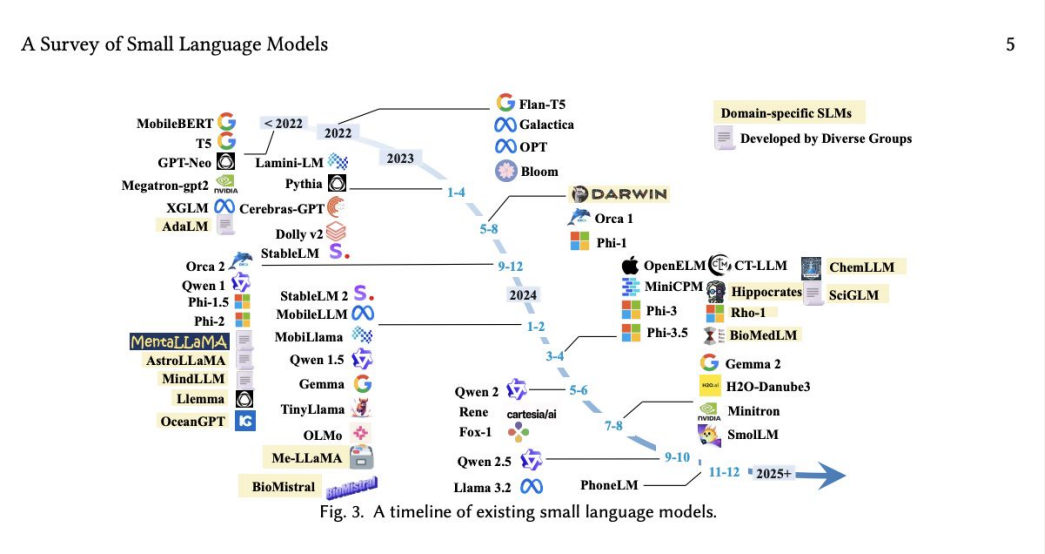


Fig. 3. A timeline of existing small language models.

[Wang et.al., 2024]

How does one build a small LM?

- Train on less amount of data?
- Train for a smaller amount of time?
- Train with more data for less amount of time?
- Train with less data longer?
- Train with larger batch sizes?

What are some common features?

- More hidden layers
- Longer training times
- Large amounts of training data
- Use of synthetic data (entirely synthetic data based small LMs also exist!)

(Note: The training part is still resource intensive in most cases)

Hardware Requirements for small LMs

A. Pre-training hyper-parameters

The pre-training hyper-parameters for different OpenELM configurations are given in Tab. 9.

	270M	450M	1.1B	3B
Dimension d_{model}	1280	1536	2048	3072
Num. of layers N	16	20	28	36
Head dimension d_h	64	64	64	128
$\alpha_{min}, \alpha_{max}$ (Eq. (1))	0.5, 1.0			
β_{min}, β_{max} (Eq. (1))	0.5, 4.0			
Normalization layer	RMSNorm			
Positional embeddings	RoPE			
Attention variant	Grouped query attention			
Activation	SwiGLU			
Context length	2048			
Batch size (tokens)	approx. 4M			
Weight tying [36]	yes			
Warm-up iterations	5,000			
Training steps	350,000			
Warm-up init. LR	0.000001			
Max. LR	0.0053	0.0039	0024	0.0012
Min. LR	10% of the max. LR			
Loss function	Cross-entropy			
Optimizer	AdamW (β_1=0.9, β_2=0.95, $\epsilon = 1.e-8$)			
Weight decay	0.1			
Activation checkpointing	✗	✓	✓	✓
FSDP	✗	✗	✗	✓
GPUs	128	128	128	128
GPU Type	A100	H100	A100	H100
GPU Memory	80 GB	80 GB	80 GB	80 GB
Training time (in days)	3	3	11	13

Table 9. Pre-training details for different variants of OpenELM.

openELM

Training cost

It takes the following number of days to train MobileLLM on 1T tokens using 32 NVIDIA A100 80G GPUs.

125M	350M	600M	1B	1.5B
~3 days	~6 days	~8 days	~12 days	~18 days

mobileLLM

Throughout the section, we work with several architectures of models whose size ranges between roughly 1M and 35M parameters, and whose number of layers range between 1 and 8 layers. All of the models can be trained on a single V100 GPU within at most 30 hours.

TinyStories

Model Size	GPU Count	Total GPU hours required
70 M	32	510
160 M	32	1,030
410 M	32	2,540
1.0 B	64	4,830
1.4 B	64	7,120
2.8 B	64	14,240
6.9 B	128	33,500
12 B	256	72,300
Total		136,070

Table 5. Model sizes in the *Pythia* suite, number of GPUs used during training, and the total number of GPU hours, calculated via (iteration time (s) × number of iterations × number of GPUs ÷ 3600 s/hour). All GPUs are A100s with 40GB of memory.

Pythia suite

Summary

- Small LMs have clear advantages for many application scenarios.
- There are many options out there in terms of pre-trained and instruction-tuned models.
- Even quantized versions of larger models *can* be fine-tuned for more task-specific use cases
- Hardware requirements are still challenging, but it is possible to train tiny models with a single GPU.
- Food for thought: Brave people can explore training their own small LM like TinyStories for an Indian language and see whether that sort of approach really works :D

We started with two questions

- How do we reduce the size of large language models?
- How do we go about building small language models?

I hope you found some answers!

Additional Readings

- What are Small Language Models? - IBM article
- A visual guide to quantization by Marten Grootendorst
- Binary and Scalar Embedding Quantization for Significantly Faster & Cheaper Retrieval
- **Distiller** is an open-source Python package for neural network compression research

- How to Prune and Distill Llama-3.1 8B to an NVIDIA Llama-3.1-Minitron 4B Model

- Beyond Answers: Transferring Reasoning Capabilities to Smaller LLMs Using Multi-Teacher Knowledge Distillation