Efficient LLMs

Advances in Large Language Models

ELL8299 · AIL861, Semester 1, 2025-26

Yatin Nandwani
Research Scientist, IBM Research

Training Vs Inference in LLMs

$P_\theta(\text{class} \mid X)$ $\quad P_\theta(y \mid X) \quad \forall y \in \mathcal{Y}$

$M_\theta(X)$ $\quad P_\theta(X = [x_1 \; \ldots \; x_T])$ $\quad 50,000 = |V|$

$50,k$ $\quad V^T$

V

$P_\theta(X)$ ↑ if X is a valid sentence

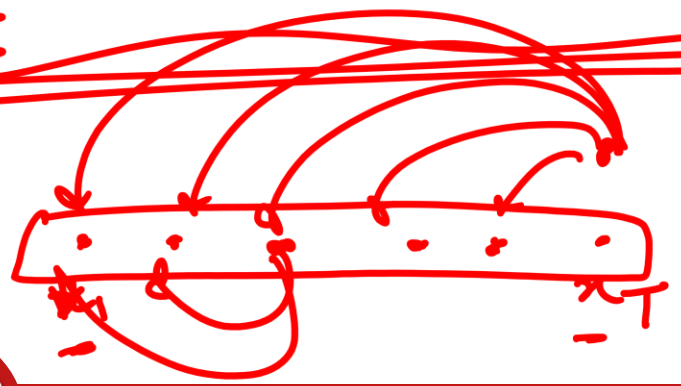
$P_\theta(X) = P_\theta(x_1)\, P_\theta(x_2 \mid x_1)\, P_\theta(x_3 \mid x_1, x_2) \cdots P_\theta(x_t \mid x_1 \cdots x_{t-1})$

$P_\theta(X)$

$P_\theta(X)$

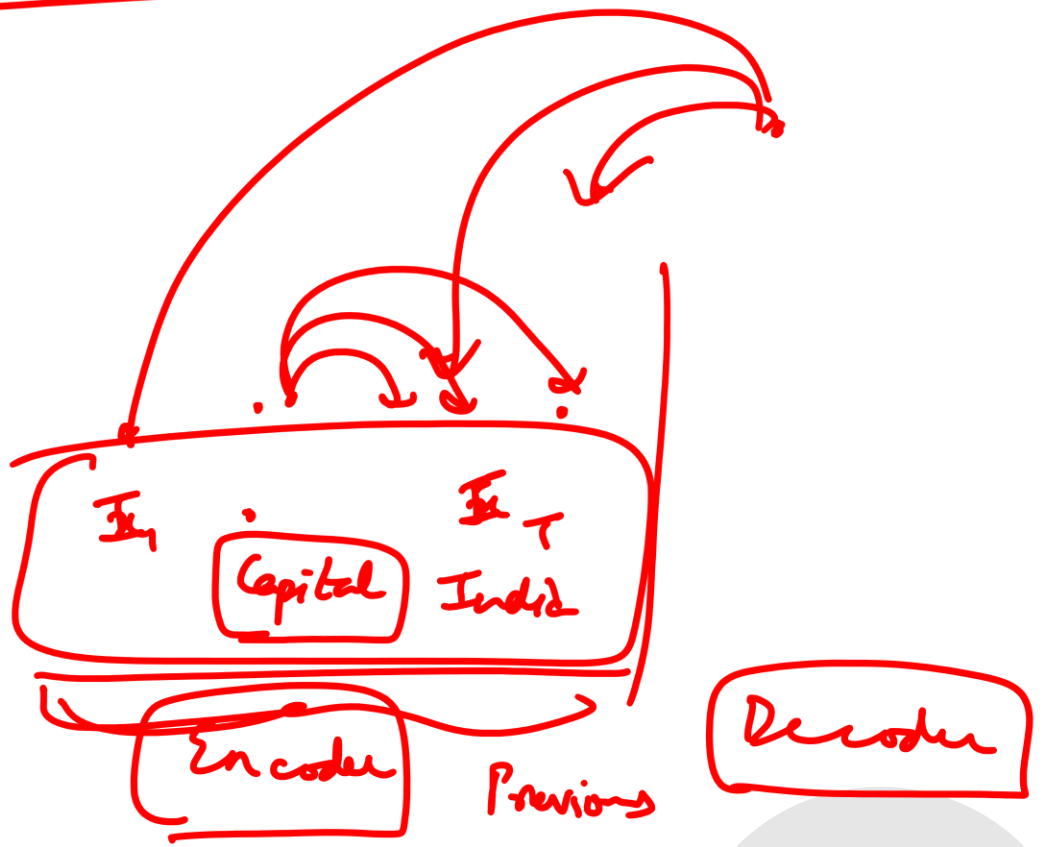
X

LO



$\log P_\theta(x) = \log P_\theta(x_1 | [BOS]) + \log P_\theta(x_2 | [BOS], x_1) + \log(P_\theta(x_T | \cdots$

$P_\theta(x_T) \quad P_\theta(x_{T+1} | x_{T+\bullet})$



Forward Pass

```python
class ToyLM(nn.Module):
    def __init__(self, vocab_size, hidden_dim, num_layers):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, hidden_dim)
        self.layers = nn.ModuleList([
            TransformerLayer(hidden_dim)
            for _ in range(num_layers)
        ])
        self.lm_head = nn.Linear(hidden_dim, vocab_size)

    def forward(self, input_ids):
        x = self.embed(input_ids)  # input embeddings
        for layer in self.layers:  # stack of transformer layers
            x = layer(x)
        logits = self.lm_head(x)   # output LM head
        return logits
```

Training Loop

```python
import torch
model = ToyLM(vocab_size, hidden_dim, num_layers)
optimizer = torch.optim.AdamW(model.parameters())

for input_ids, labels in dataloader:
    logits = model(input_ids)
```

Training Loop

```python
import torch
model = ToyLM(vocab_size, hidden_dim, num_layers)
optimizer = torch.optim.AdamW(model.parameters())

for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
```

Input :- The cat sat on the mat

↓

Label : Cat

Training Loop

```
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference

```python
import torch

def generate(model, input_string):
    input_ids = tokenizer.encode(input_string)
    #input_id_tokens: ["<s>","The","cat","sat"]
```

Training Loop

```python
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference

```python
import torch

def generate(model, input_string):
    input_ids = tokenizer.encode(input_string)
    #input_id_tokens: ["<s>","The","cat","sat"]
    for _ in range(max_len):
        logits = model(input_ids)
```

Training Loop

```python
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference

```python
import torch

def generate(model, input_string):
    input_ids = tokenizer.encode(input_string)
    #input_id_tokens: ["<s>","The","cat","sat"]

        logits = model(input_ids)
        next_token = torch.argmax(logits[:, -1, :],
                                  dim=-1
        )
```

Training Loop

```python
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference

```python
import torch

def generate(model, input_string):
    input_ids = tokenizer.encode(input_string)
    #input_id_tokens: ["<s>","The","cat","sat"]

        logits = model(input_ids)
        next_token = torch.argmax(logits[:, -1, :],
                                  dim=-1
        )
        if next_token[0] == "</s>":
            break
```

Training Loop

```python
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference

```python
import torch

def generate(model, input_string):
    input_ids = tokenizer.encode(input_string)
    #input_id_tokens: ["<s>","The","cat","sat"]

        logits = model(input_ids)
        next_token = torch.argmax(logits[:, -1, :],
                                  dim=-1
        )
        if next_token[0] == "</s>":
            break
        input_ids = torch.cat([
            input_ids,
            next_token.unsqueeze(-1)], dim=-1
        )
```

Training Loop

```python
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference

```python
import torch

def generate(model, input_string):
    input_ids = tokenizer.encode(input_string)
    #input_id_tokens: ["<s>","The","cat","sat"]
    for _ in range(max_len):
        logits = model(input_ids)
        next_token = torch.argmax(logits[:, -1, :],
                                  dim=-1
        )
        if next_token[0] == "</s>":
            break
        input_ids = torch.cat([
            input_ids,
            next_token.unsqueeze(-1)], dim=-1
        )
```

Training Loop

```python
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference

```python
import torch

def generate(model, input_string):
    input_ids = tokenizer.encode(input_string)
    #input_id_tokens: ["<s>","The","cat","sat"]
    for _ in range(max_len):
        logits = model(input_ids)
        next_token = torch.argmax(logits[:, -1, :],
                                  dim=-1
        )
        if next_token[0] == "</s>":
            break
        input_ids = torch.cat([
            input_ids,
            next_token.unsqueeze(-1)], dim=-1
        )

input_sting = "The cat sat"
model = ToyLM(vocab_size, hidden_dim, num_layers)
output_ids  = generate(input_string, model)
print(tokenizer.decode(input_ids[0]))
```

Training Loop

```python
for input_ids, labels in dataloader:
    logits = model(input_ids)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Forward Pass

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference w/ caching

Forward Pass w/ caching

Forward Pass w/o caching

```
13      def forward(self, input_ids):
14          x = self.embed(input_ids)
15          for layer in self.layers:
16              x = layer(x)
17          logits = self.lm_head(x)
18          return logits
```

Inference w/ caching

Forward Pass w/ caching

```python
def forward(self, input_ids, cache=None):
    x = self.embed(input_ids)

    new_cache = []
    for i, layer in enumerate(self.layers):
        # Each layer consumes (x, past_kv)
        # and returns (x, new_kv)
        past_kv = cache[i] if cache else None
        x, kv = layer(x, past_kv)
        new_cache.append(kv)

    logits = self.lm_head(x)
    return logits, new_cache
```

Forward Pass w/o caching

```python
    def forward(self, input_ids):
        x = self.embed(input_ids)
        for layer in self.layers:
            x = layer(x)
        logits = self.lm_head(x)
        return logits
```

Inference w/ caching

Forward Pass w/ caching

```
24  def forward(self, input_ids, cache=None):
25      x = self.embed(input_ids)
26
27      new_cache = []
28      for i, layer in enumerate(self.layers):
29          # Each layer consumes (x, past_kv)
30          # and returns (x, new_kv)
31          past_kv = cache[i] if cache else None
32          x, kv = layer(x, past_kv)
33          new_cache.append(kv)
34
35      logits = self.lm_head(x)
36      return logits, new_cache
```

Forward Pass w/o caching

```
13      def forward(self, input_ids):
14          x = self.embed(input_ids)
15          for layer in self.layers:
16              x = layer(x)
17          logits = self.lm_head(x)
18          return logits
```

Inference w/ caching

```
38 def generate(model, input_string, max_len, eos_token_id):
39     # 1. Prefill: run full prompt once
40     input_ids = tokenizer.encode(input_string).unsqueeze(0)
41     logits, cache = model(input_ids, cache=None)
42
```

Forward Pass w/ caching

```
24 def forward(self, input_ids, cache=None):
25     x = self.embed(input_ids)
26
27     new_cache = []
28     for i, layer in enumerate(self.layers):
29         # Each layer consumes (x, past_kv)
30         # and returns (x, new_kv)
31         past_kv = cache[i] if cache else None
32         x, kv = layer(x, past_kv)
33         new_cache.append(kv)
34
35     logits = self.lm_head(x)
36     return logits, new_cache
```

Forward Pass w/o caching

```
13     def forward(self, input_ids):
14         x = self.embed(input_ids)
15         for layer in self.layers:
16             x = layer(x)
17         logits = self.lm_head(x)
18         return logits
```

Inference w/ caching

```python
def generate(model, input_string, max_len, eos_token_id):
    # 1. Prefill: run full prompt once
    input_ids = tokenizer.encode(input_string).unsqueeze(0)
    logits, cache = model(input_ids, cache=None)

    # 2. Iterative decoding
    for _ in range(max_len):
        next_token = argmax(logits[:, -1, :])
        input_ids = cat([input_ids,
                         next_token.unsqueeze(-1)], dim=-1)

        # --- Early stop if EOS is generated ---
        if next_token.item() == eos_token_id:
            break

        # feed only the last token with cache
        logits, cache = model(input_ids[:, -1:], cache)

```

Forward Pass w/ caching

```python
def forward(self, input_ids, cache=None):
    x = self.embed(input_ids)

    new_cache = []
    for i, layer in enumerate(self.layers):
        # Each layer consumes (x, past_kv)
        # and returns (x, new_kv)
        past_kv = cache[i] if cache else None
        x, kv = layer(x, past_kv)
        new_cache.append(kv)

    logits = self.lm_head(x)
    return logits, new_cache
```

Forward Pass w/o caching

```python
def forward(self, input_ids):
    x = self.embed(input_ids)
    for layer in self.layers:
        x = layer(x)
    logits = self.lm_head(x)
    return logits
```

Blockwise Self-Attention Computation

The	key	difference	between	LLM	training	and	inference
K1	K2	K3	K4	K5	K6	K7	K8
							Q8

V1	The
V2	key
V3	difference
V4	between
V5	LLM
V6	training
V7	and
V8	inference

The	key	difference	between	LLM	training	and	inference
K1	K2	K3	K4	K5	K6	K7	K8
							Q8

$$\frac{\exp(q_8 * k_1) * \mathrm{v}_1 + \cdots + \exp(q_8 * k_4) * \mathrm{v}_4 + \exp(q_8 * k_5) * \mathrm{v}_5 + \cdots + \exp(q_8 * k_8) * \mathrm{v}_8}{\exp(q_8 * k_1) + \cdots + \exp(q_8 * k_4) + \exp(q_8 * k_5) + \cdots + \exp(q_8 * k_8)}$$

V1	The
V2	key
V3	difference
V4	between
V5	LLM
V6	training
V7	and
V8	inference

The	key	difference	between	LLM	training	and	inference
K1	K2	K3	K4	K5	K6	K7	K8
							Q8

$$\frac{\sum_{i=1..8} \exp(q_8 * k_i) * v_i}{\sum_{i=1..8} \exp(q_8 * k_i)}$$

V1	The
V2	key
V3	difference
V4	between
V5	LLM
V6	training
V7	and
V8	inference

The	key	difference	between	LLM	training	and	inference
K1	K2	K3	K4	K5	K6	K7	K8

$$\frac{\sum_{i=1..8} \exp(q_8 * k_i) * v_i}{\sum_{i=1..8} \exp(q_8 * k_i)}$$

inf.
Q8

The	key	diff.	b/w
K1	K2	K3	K4
s1	s2	s3	s4
a1	a2	a3	a4

V1
V2
V3
V4

inf.
Q8

LLM	tr.	and	inf.
K5	K6	K7	K8
s5	s6	s7	s8
a5	a6	a7	a8

V5
V6
V7
V8

V1	The
V2	key
V3	difference
V4	between
V5	LLM
V6	training
V7	and
V8	inference

$$l^{(1)} = \sum_{i=1..4} \exp(S_i)$$

$$O^{(1)} = \frac{\mathbf{A}^{(1)}\mathbf{V}^{(1)}}{l^{(1)}}$$

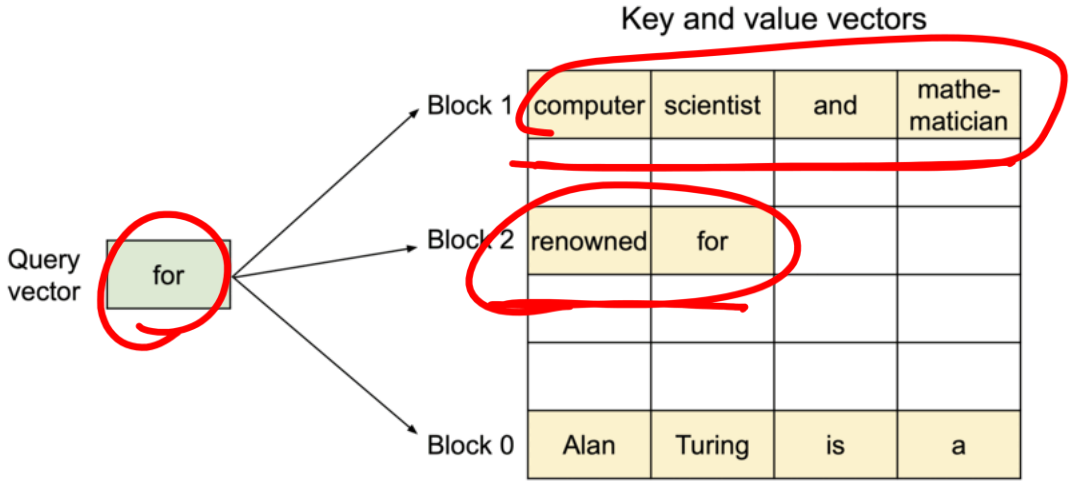
$$l^{(2)} = \sum_{i=5..8} \exp(S_i)$$

$$O^{(2)} = \frac{\mathbf{A}^{(2)}\mathbf{V}^{(2)}}{l^{(2)}}$$

$$O = \frac{l^{(1)}O^{(1)} + l^{(2)}O^{(2)}}{l^{(1)} + l^{(2)}}$$

Paged Attention

- Tensor operations require contiguous memory
- How to compute attention *softmax* across fragmented memory?
- Paged Attention!



$$softmax([A_1, A_2]) = [\, \alpha\ softmax(A_1), \beta\ softmax(A_2)\,]$$

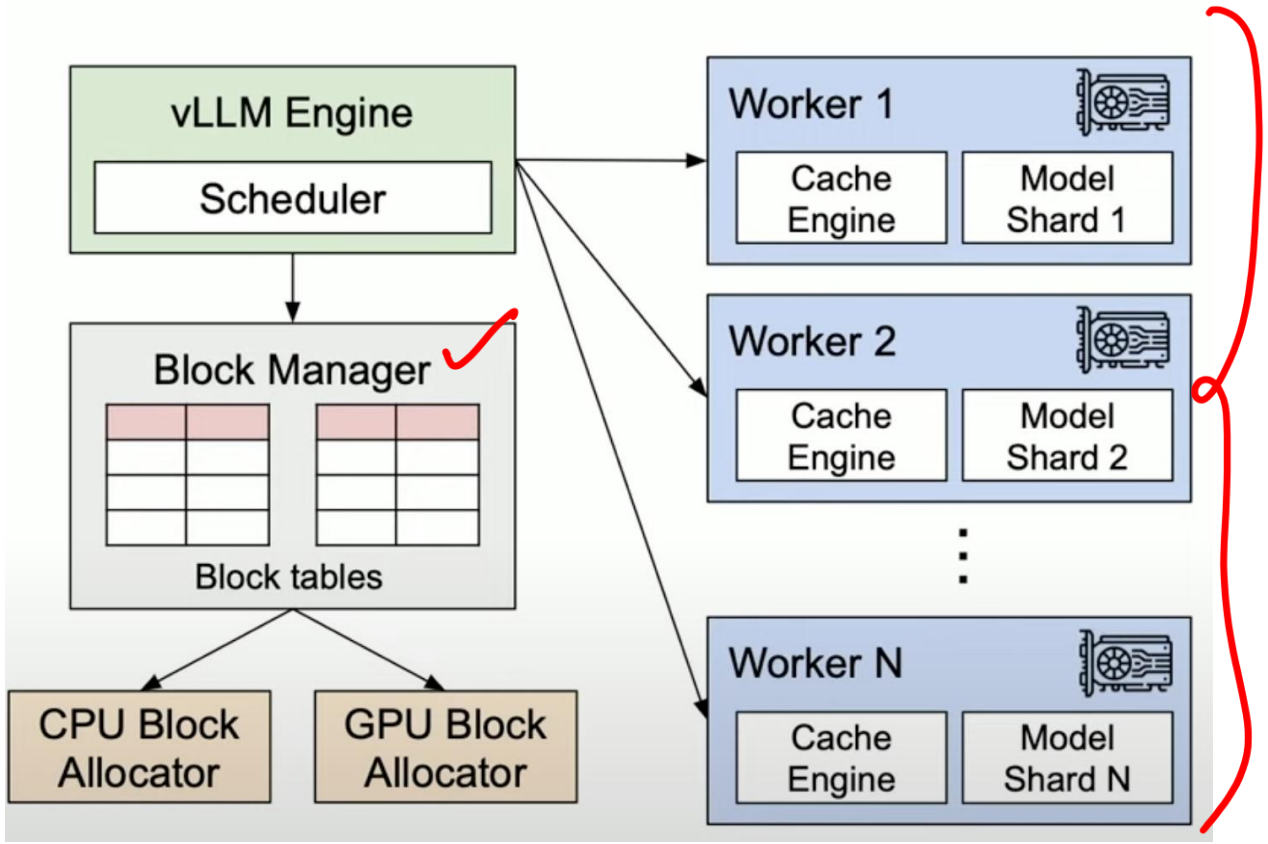
$$softmax([A_1, A_2]) \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \alpha\ softmax(A_1) * V_1 + \beta\ softmax(A_2) * V_2$$

Homework!

1. Understand the idea of **Safe-Softmax**
2. Read about the concept of **Tiling** used in Matrix Multiplication and Flash Attention

Reference: https://courses.cs.washington.edu/courses/cse599m/23sp/notes/flashattn.pdf

System Architecture and Implementation



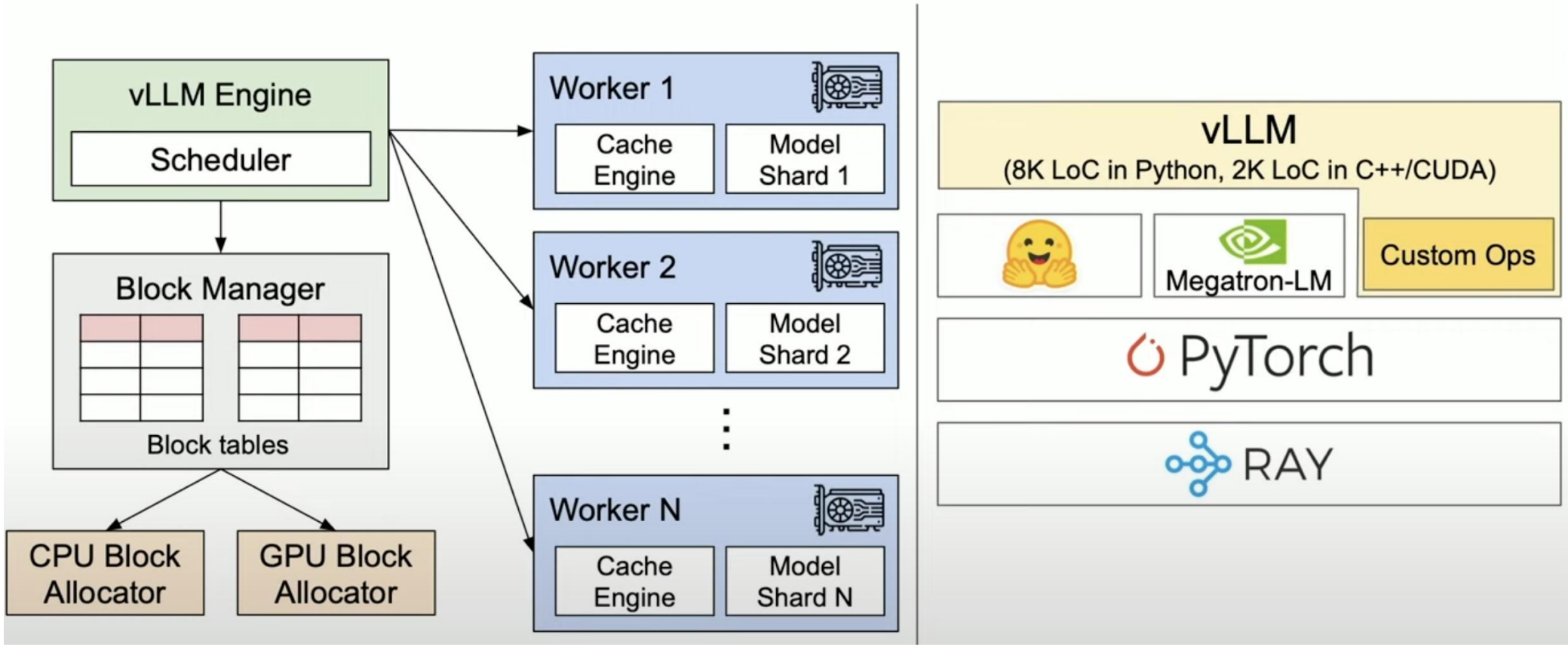
End to end llm serving engine

3 components –

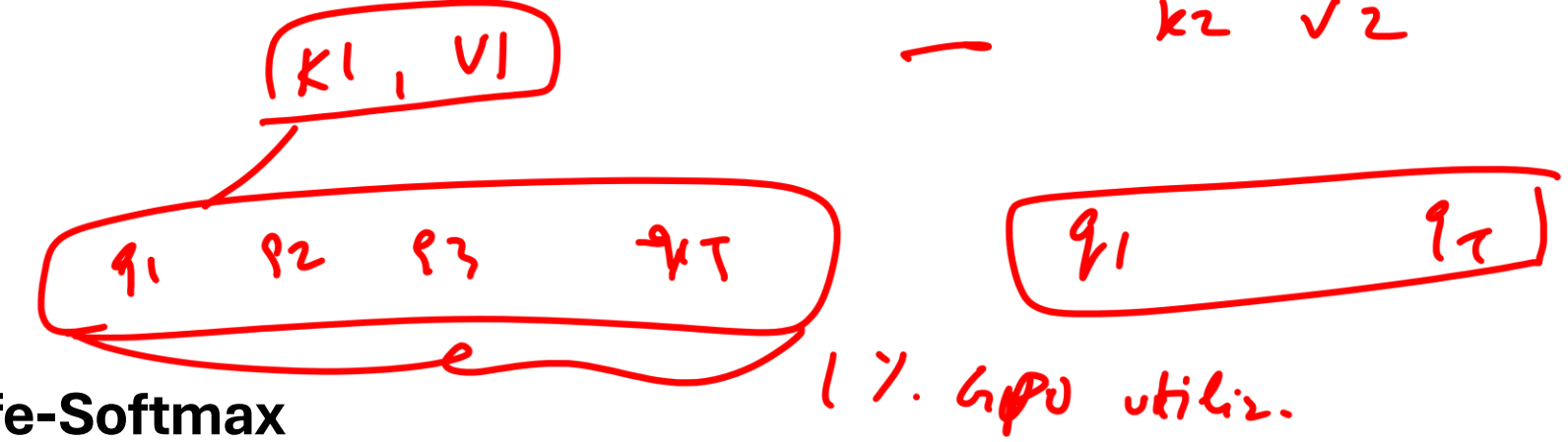
- A frontend
- A distributed model executor (TP on single node, PP across nodes)
- A scheduler

Centralized engine to manage block table

- At each iteration, it sends GPU memory requests to the GPUs;
- Cache engine in the GPU allocates the physical memory blocks


vLLM Engine
Scheduler
Block Manager
Block tables
CPU Block Allocator
GPU Block Allocator
Worker 1
Cache Engine
Model Shard 1
Worker 2
Cache Engine
Model Shard 2
Worker N
Cache Engine
Model Shard N
vLLM
(8K LoC in Python, 2K LoC in C++/CUDA)
Megatron-LM
Custom Ops
PyTorch
RAY

Homework!

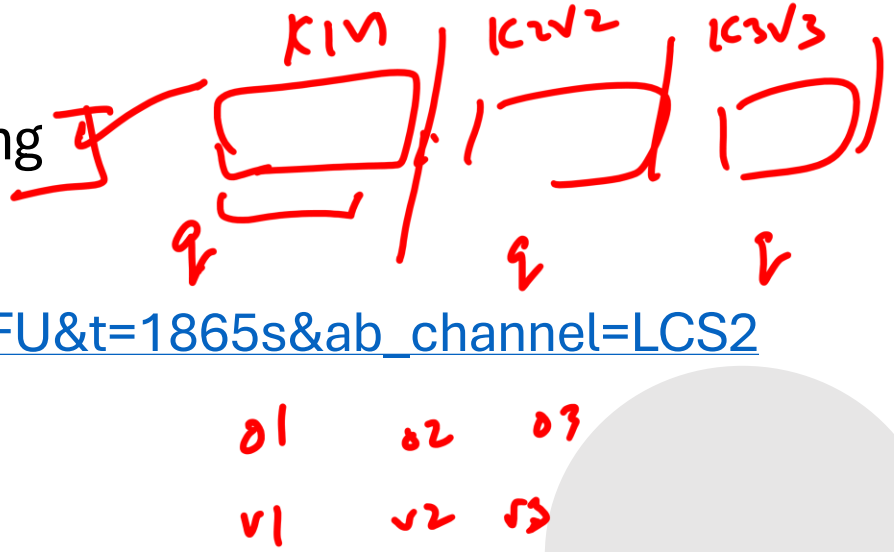


1. Understand the idea of **Safe-Softmax**
2. Read about the concept of **Tiling** used in Matrix Multiplication and Flash Attention

Reference: https://courses.cs.washington.edu/courses/cse599m/23sp/notes/flashattn.pdf

3. **Flash Decoding** – variant of flash attention for decoding
4. **Speculative decoding** – guess and verify.

Reference: https://www.youtube.com/watch?v=JVUWCv_vMFU&t=1865s&ab_channel=LCS2



So Far...

Efficient Training

1. Parallelism for Scale -
   - *Distribute model & data to fit massive training*
   - *DP, ZeRO-1/2/3, TP (w/ SP), CP, PP*
2. Efficient Implementations
   - *Flash Attention*
   - *Liger Kernels*

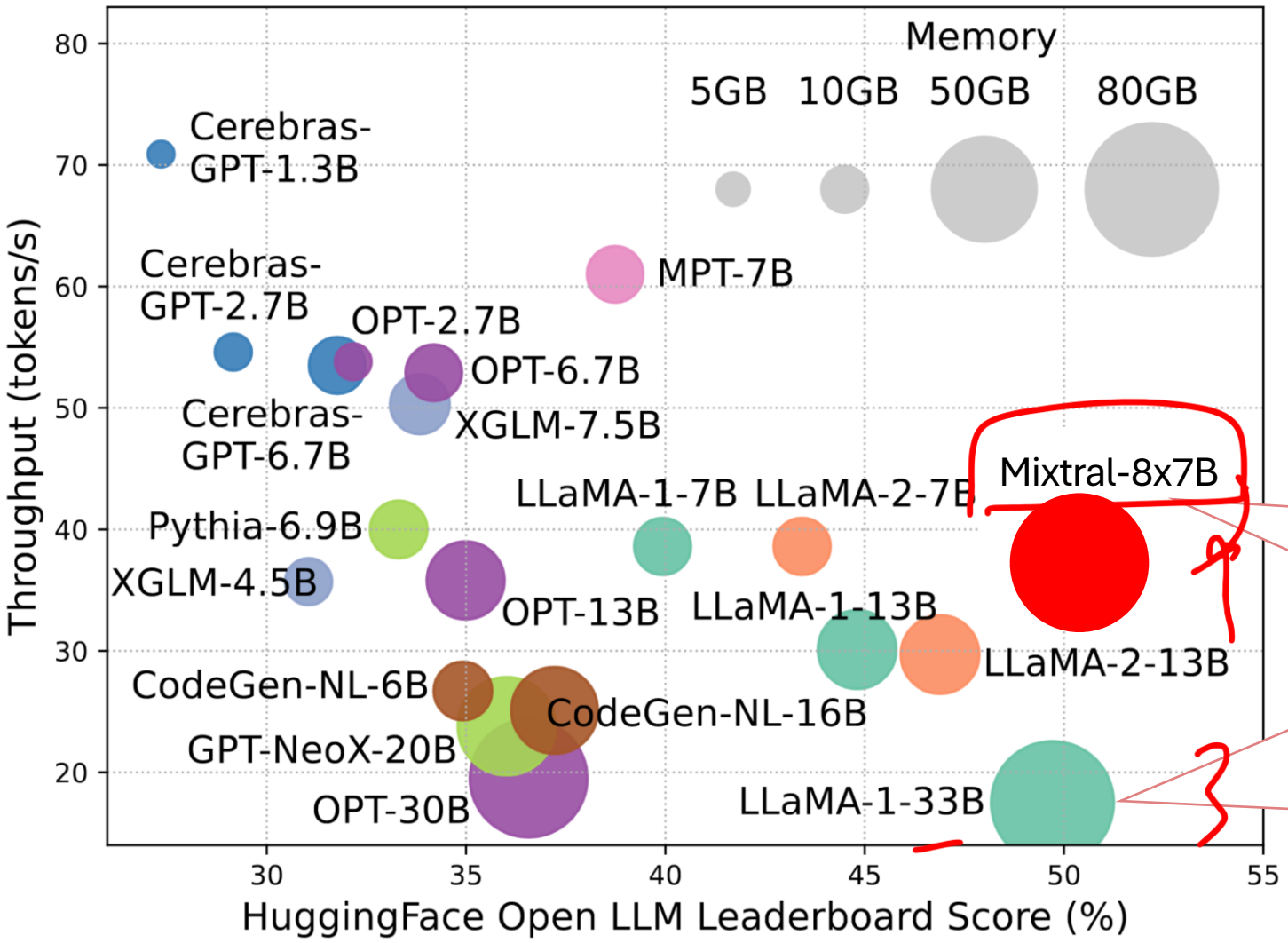
Efficient Inference

1. KV Caching
2. vLLMs and paged attention
3. Flash decoding
4. Speculative decoding

Homework!

Inference Throughput vs Performance

Inference Throughput vs Performance



- Similar performance, different throughput! How?
- Efficient design –
  - MoE

So Far...

Efficient Training

1. Parallelism for Scale -
   - *Distribute model & data to fit massive training*
   - *DP, ZeRO-1/2/3, TP (w/ SP), CP, PP*

2. Efficient Implementations
   - *Flash Attention*
   - *Liger Kernels*

Efficient Inference

1. KV Caching
2. vLLMs and paged attention
3. Flash decoding
4. Speculative decoding

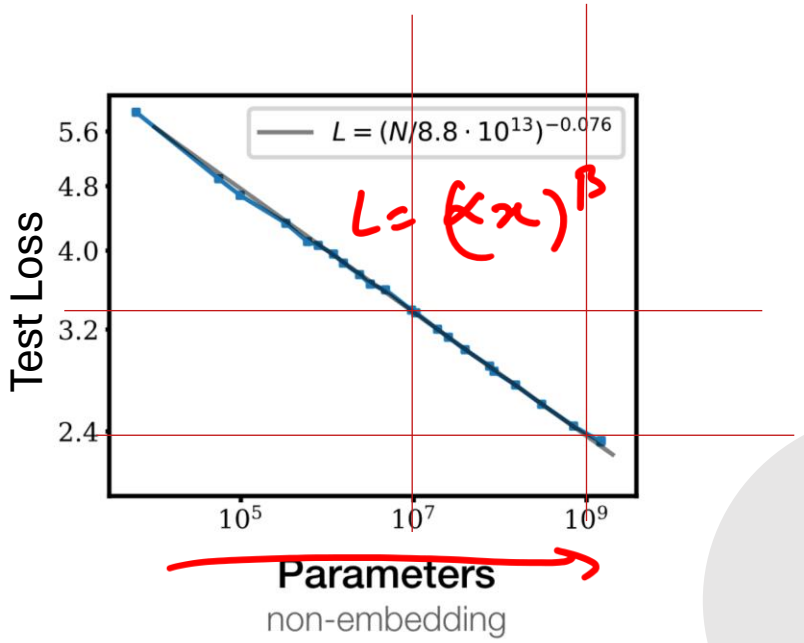
Homework!

Efficient Design

1. MoE architecture

Neural Scaling Laws Kaplan et al 2020

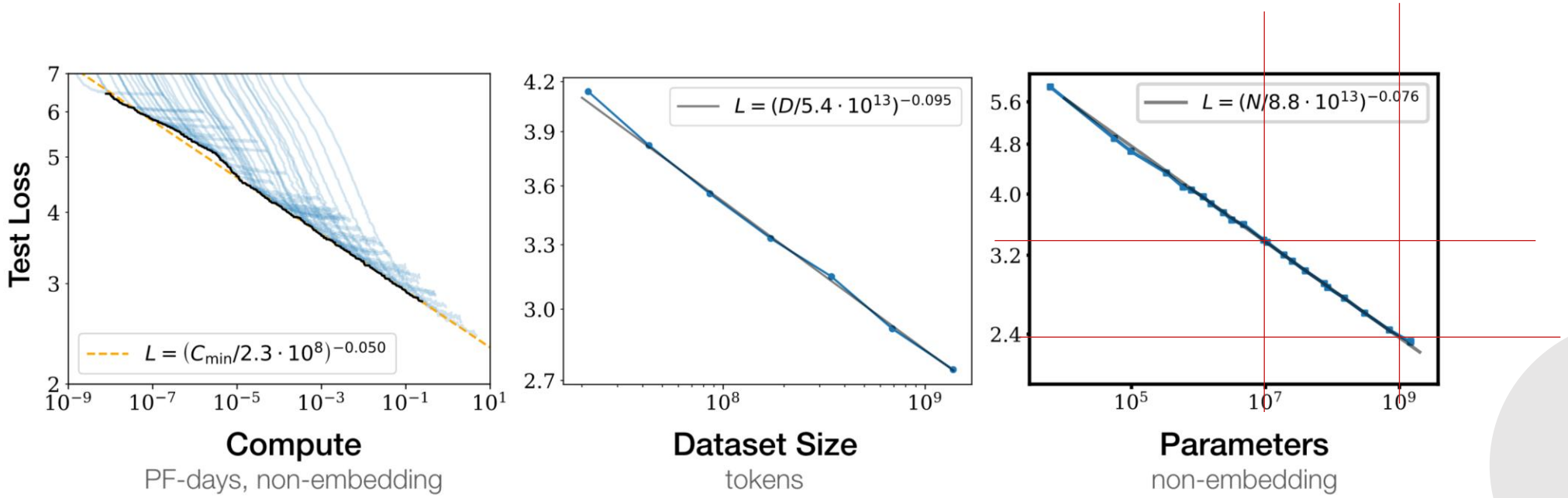
- Performance improve smoothly as we increase the **model size**
- Larger models → smoother loss curves, better generalization, new emergent abilities



Source: Scaling Laws for Neural Language Models, Kaplan et al . 2020, Open AI

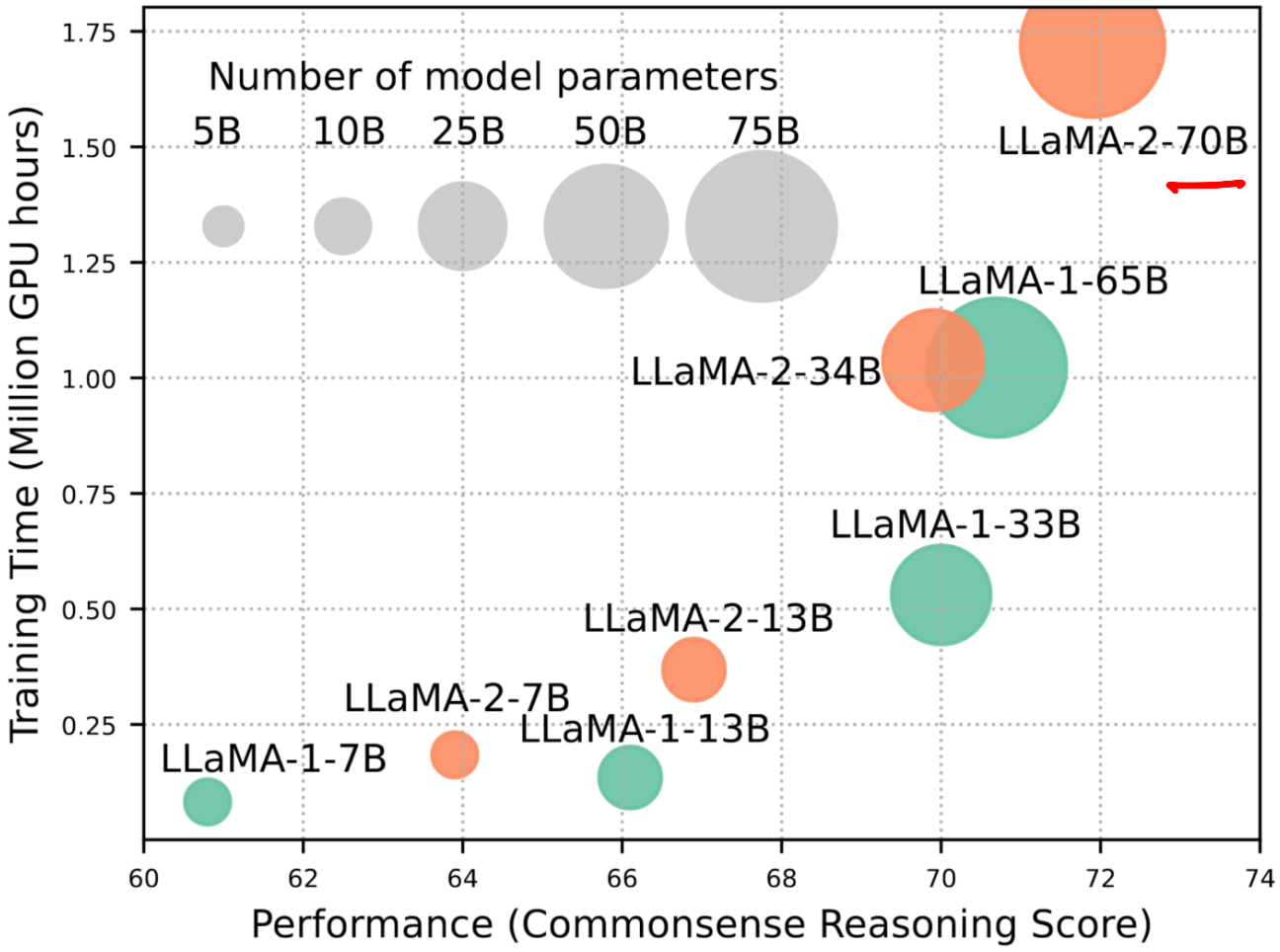
Neural Scaling Laws Kaplan et al 2020

- Performance improve smoothly as we increase the **compute, dataset size**, or the model size



Source: Scaling Laws for Neural Language Models, Kaplan et al . 2020, Open AI

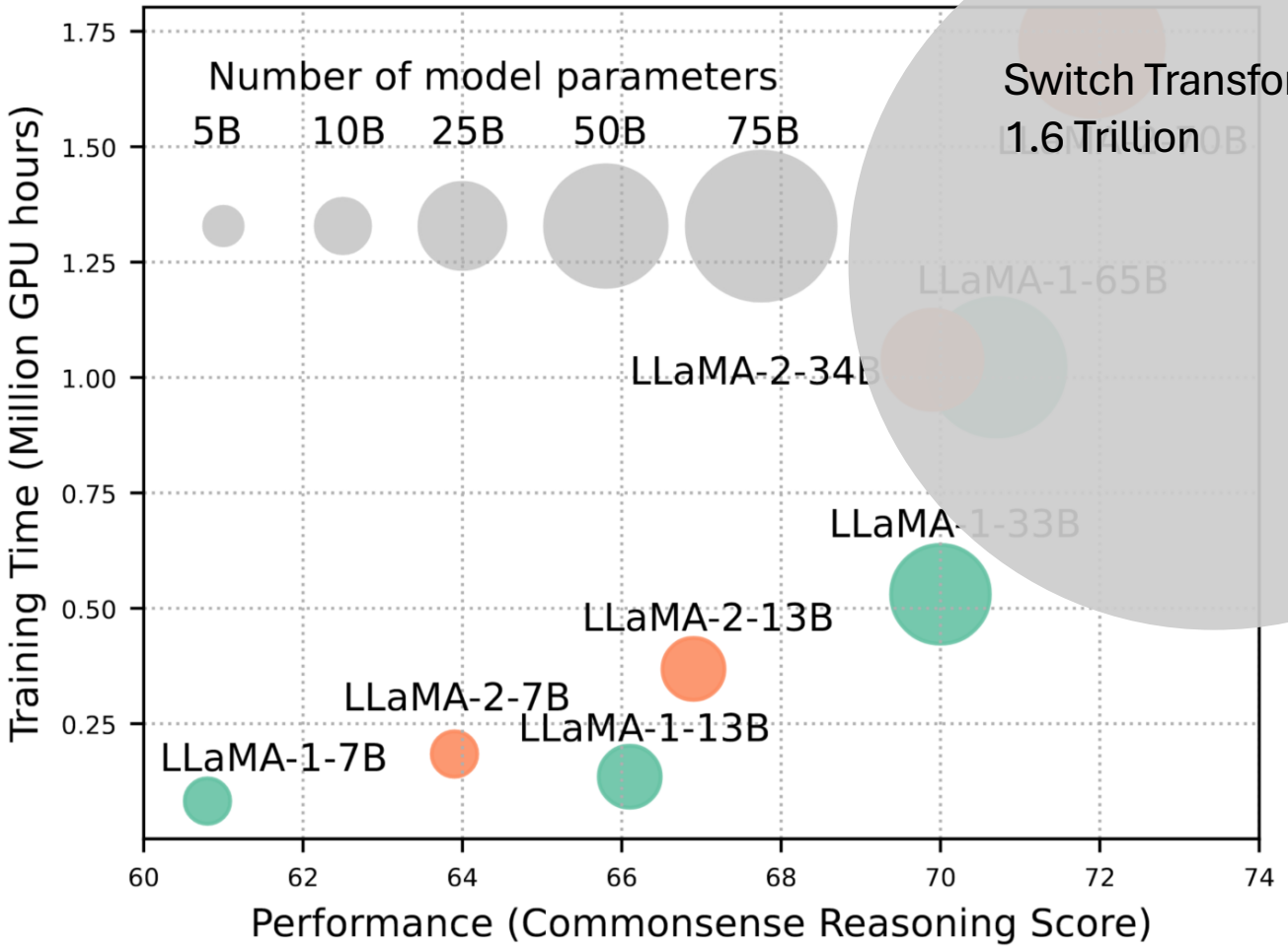
Training Resources vs Performance



- Based on Nvidia A100 80GB GPU

https://huggingface.co/spaces/optimum/llm-perf-leaderboard

Training Resources vs Performance



- Based on Nvidia A100 80GB GPU

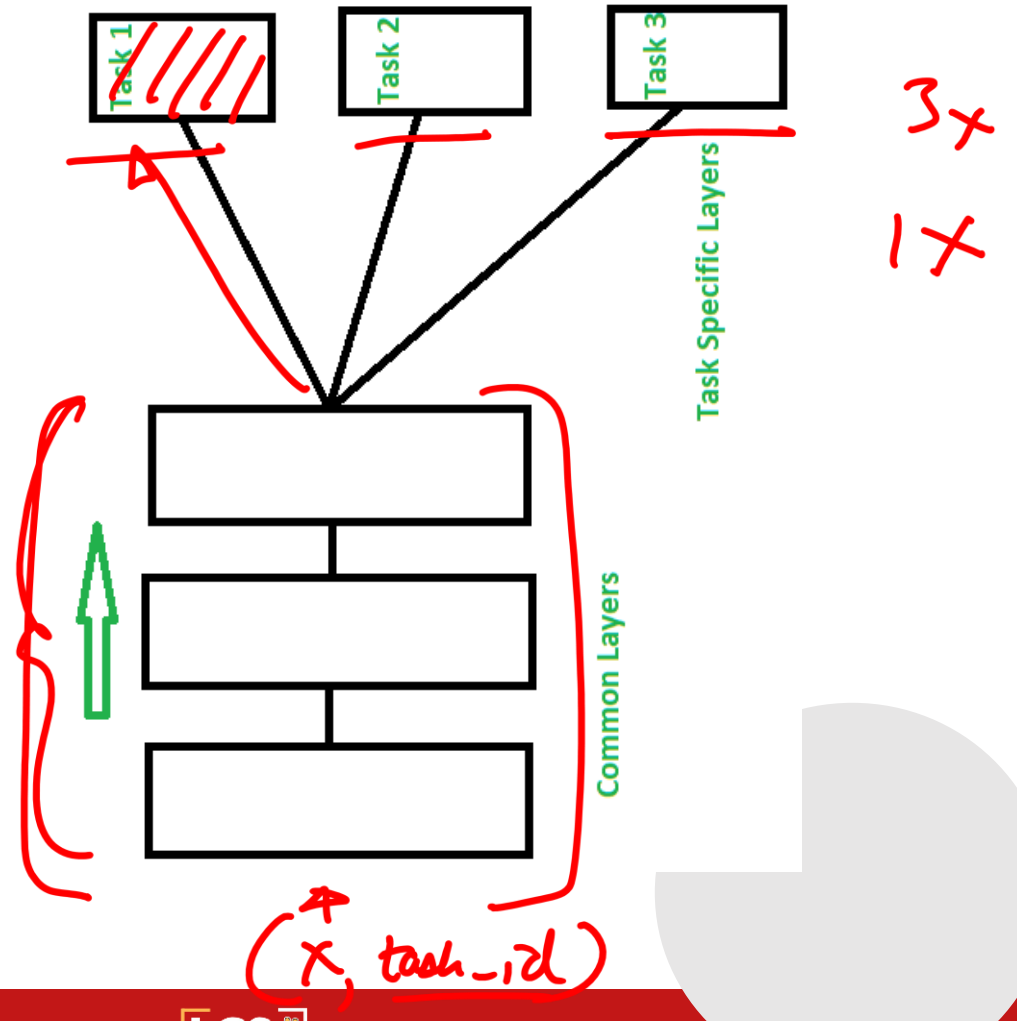
https://huggingface.co/spaces/optimum/llm-perf-leaderboard

Any efficient way to increase model size?

- **LLMs as generalists** → they implicitly do *multi-task learning* (MTL)
  - summarization, QA, coding, reasoning, etc.
- How does a general MTL model look like?

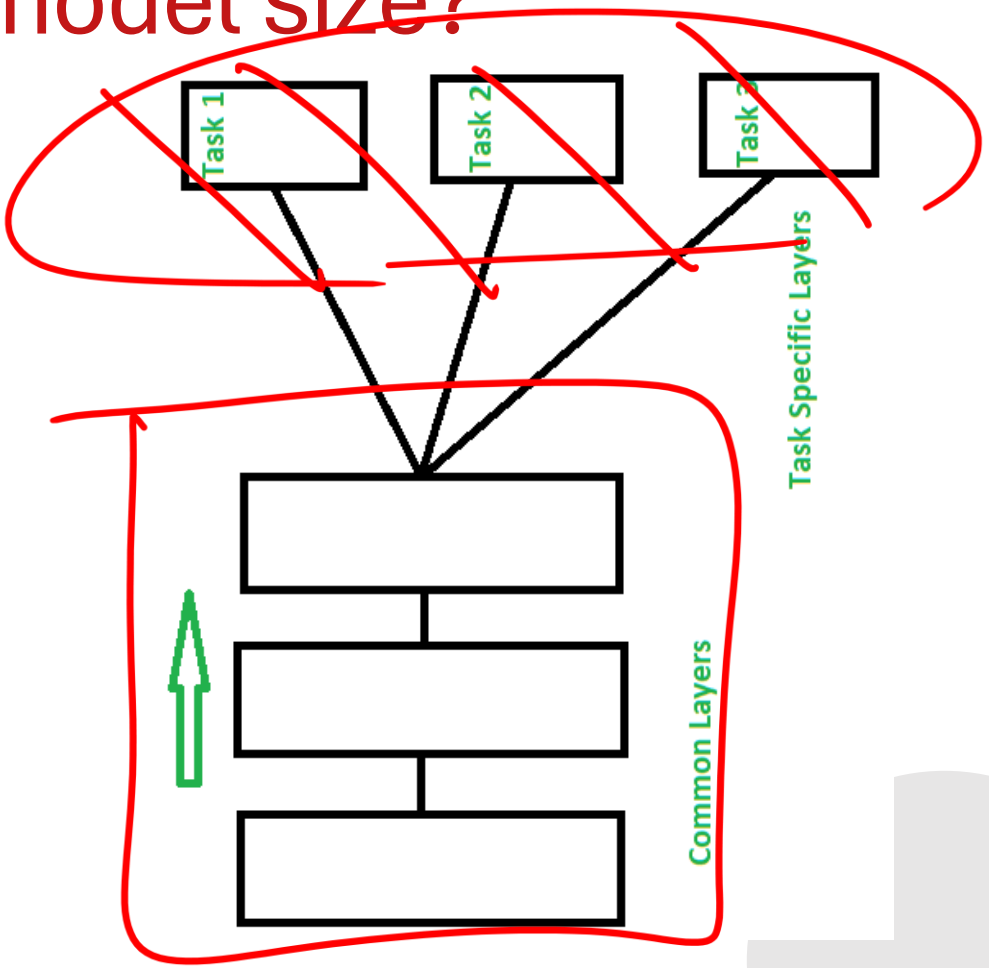
Any efficient way to increase model size?

- **LLMs as generalists** → they implicitly do *multi-task learning* (MTL)
  - summarization, QA, coding, reasoning, etc.
- How does a general MTL model look like?
  - Shared parameters (always active)
  - Task specific parameters (active based on input/desired output)



Any efficient way to increase model size?

- **LLMs as generalists** → they implicitly do *multi-task learning* (MTL)
  - summarization, QA, coding, reasoning, etc.
- How does a general MTL model look like?
  - Shared parameters (always active)
  - Task specific parameters (active based on task)
- In LLMs → all parameters active for all tasks
- Can we create a model where only few parameters are active for a given input?



Mixture of Experts

Adaptive Mixtures of Local Experts

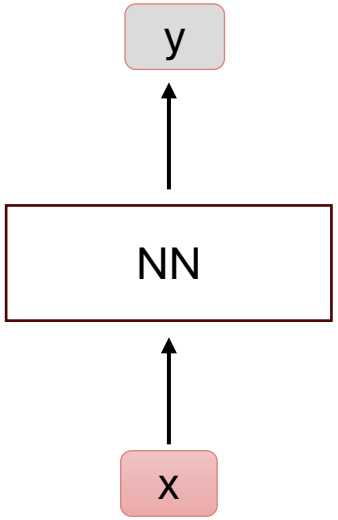
Robert A. Jacobs
Michael I. Jordan
Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, Cambridge, MA 02139 USA

Steven J. Nowlan
Geoffrey E. Hinton
Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A4

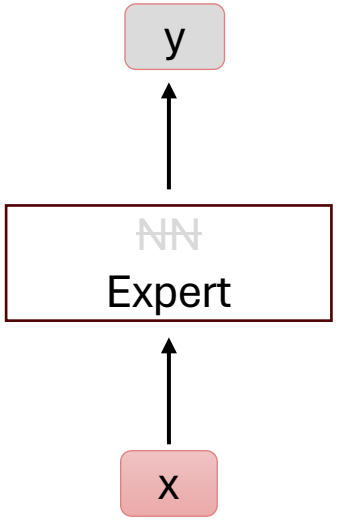
Published in 1991!

We present a new supervised learning procedure for systems composed of many separate networks, each of which learns to handle a subset of the complete set of training cases. The new procedure can be viewed either as a modular version of a multilayer supervised network, or as an associative version of competitive learning. It therefore provides a new link between these two apparently different approaches. We demonstrate that the learning procedure divides up a vowel discrimination task into appropriate subtasks, each of which can be solved by a very simple expert network.

Mixture of Experts

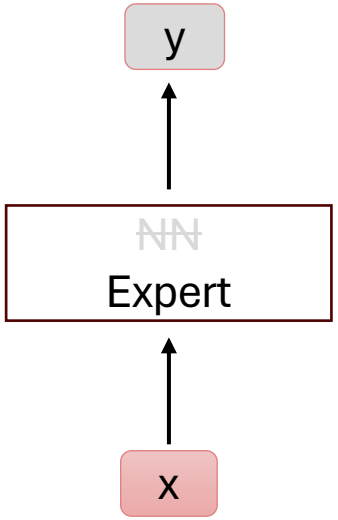


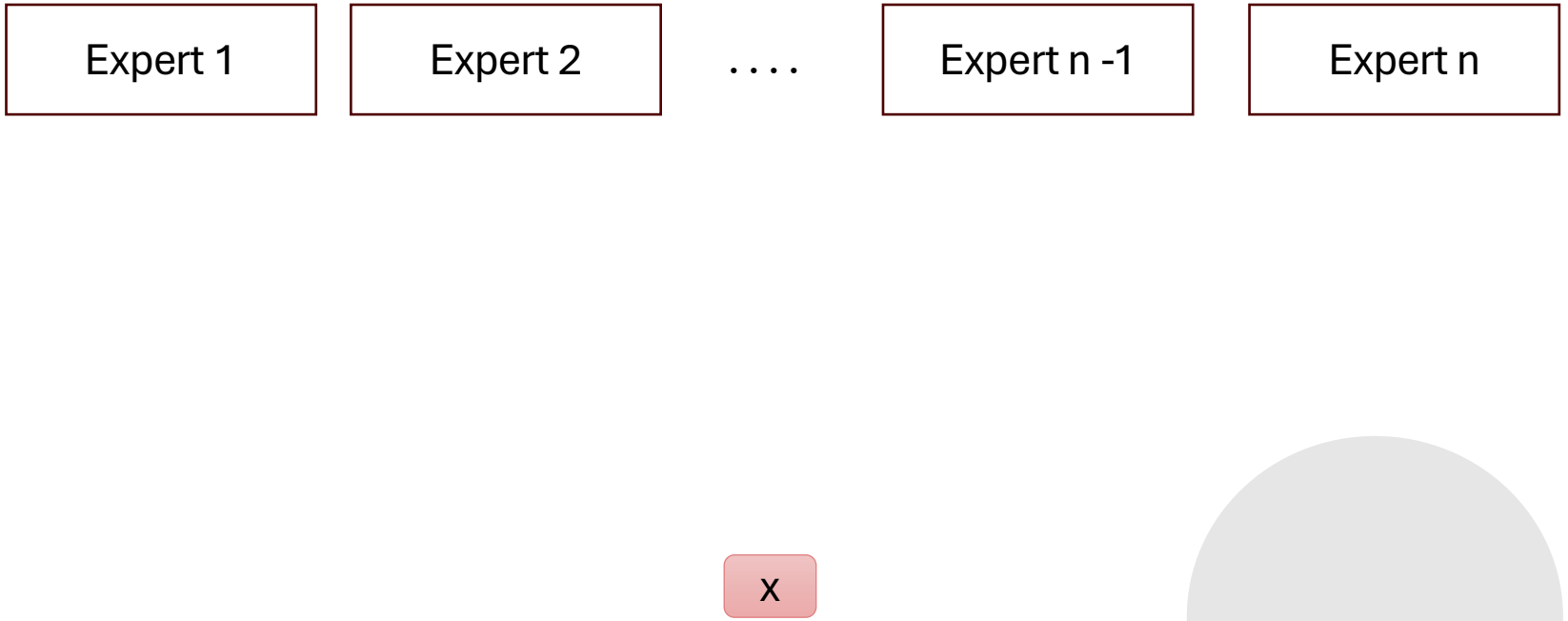
Mixture of Experts



How to decide which expert to use?

Mixture of Experts

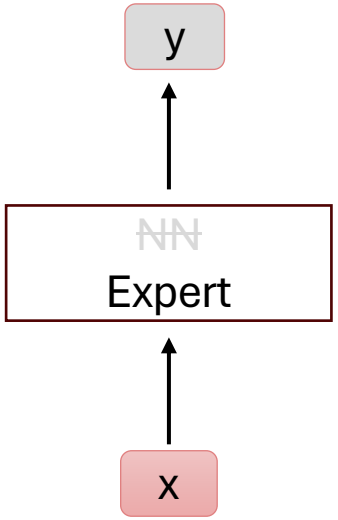


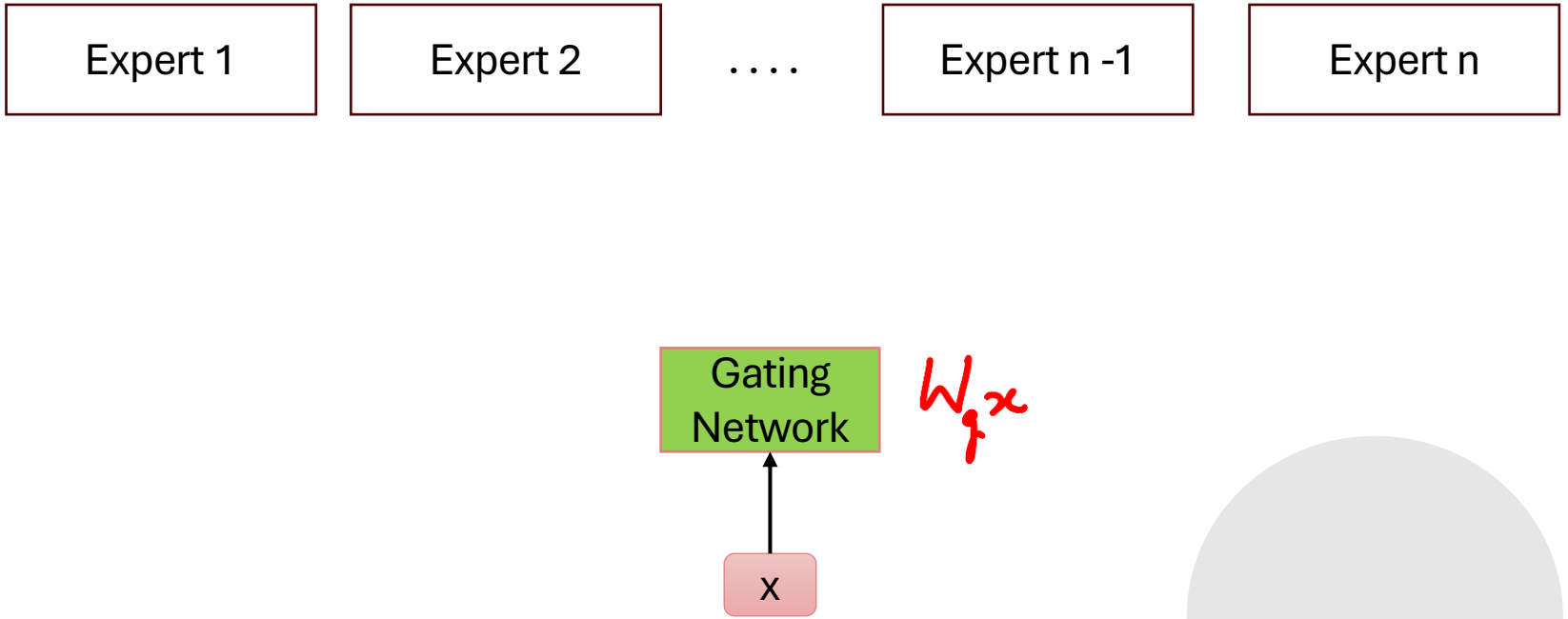


How to decide which expert to use?

Let's have a "Gating Network" that decides (*probabilistically*)

Mixture of Experts

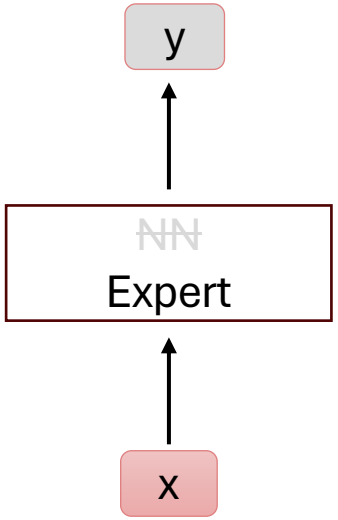


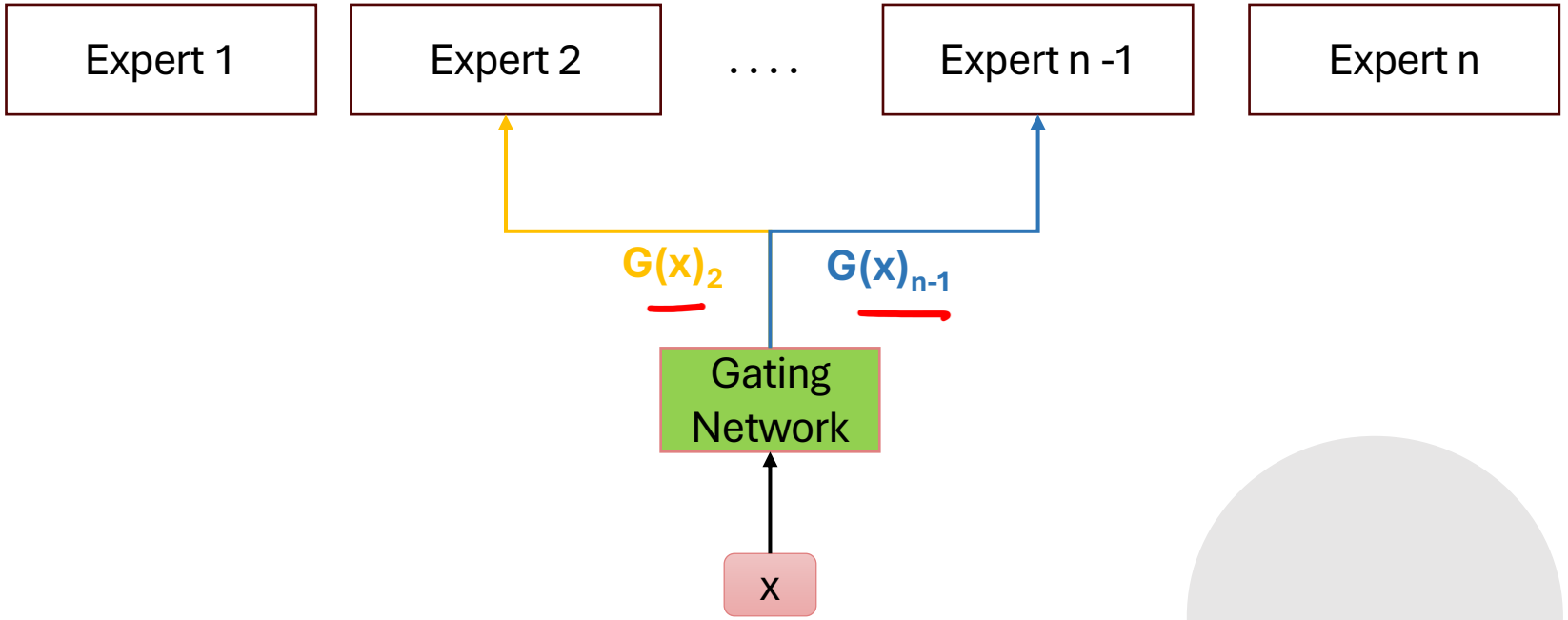


How to decide which expert to use?

Let's have a "Gating Network" that decides (*probabilistically*)

Mixture of Experts

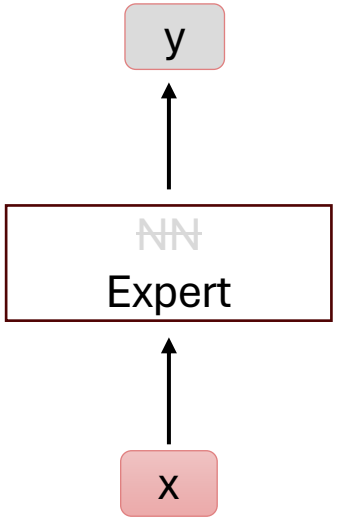


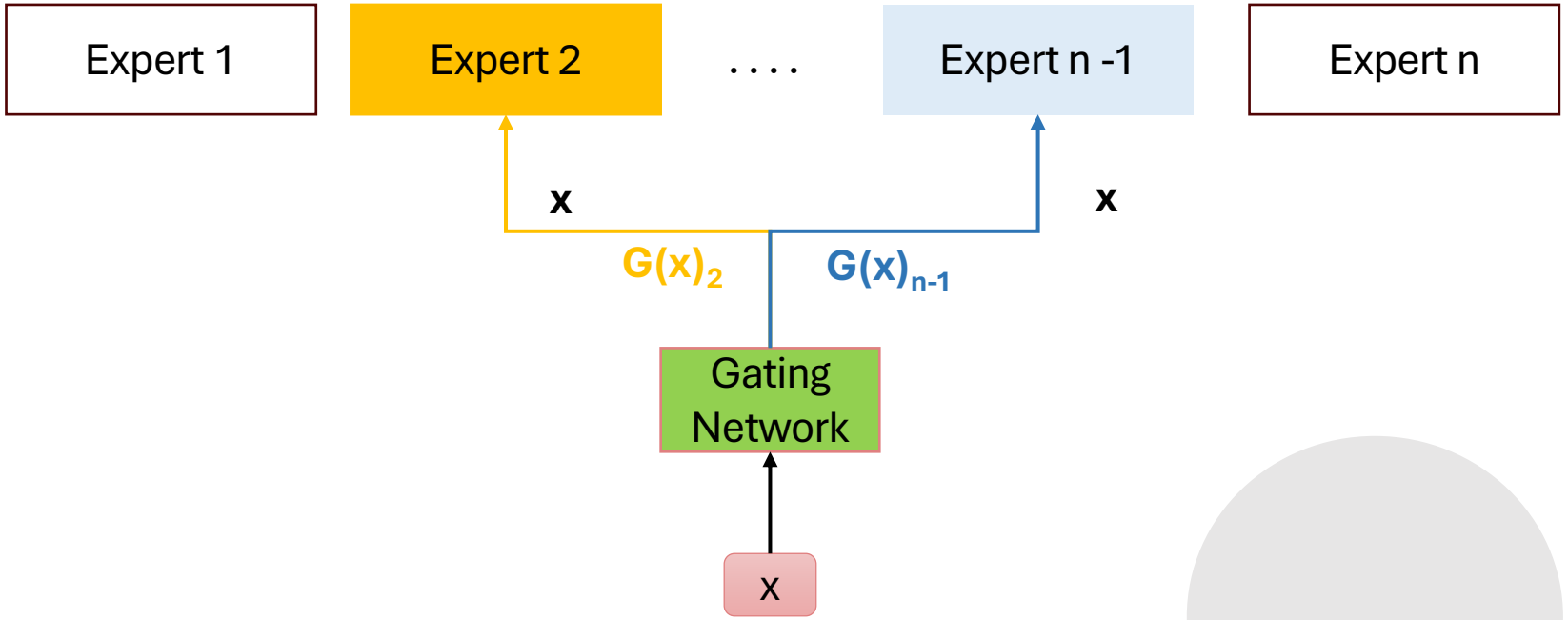


How to decide which expert to use?

Let's have a "Gating Network" that decides (*probabilistically*)

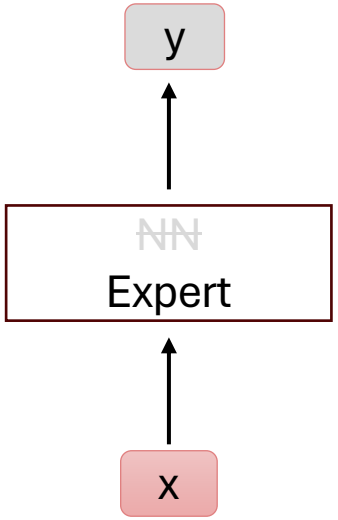
Mixture of Experts

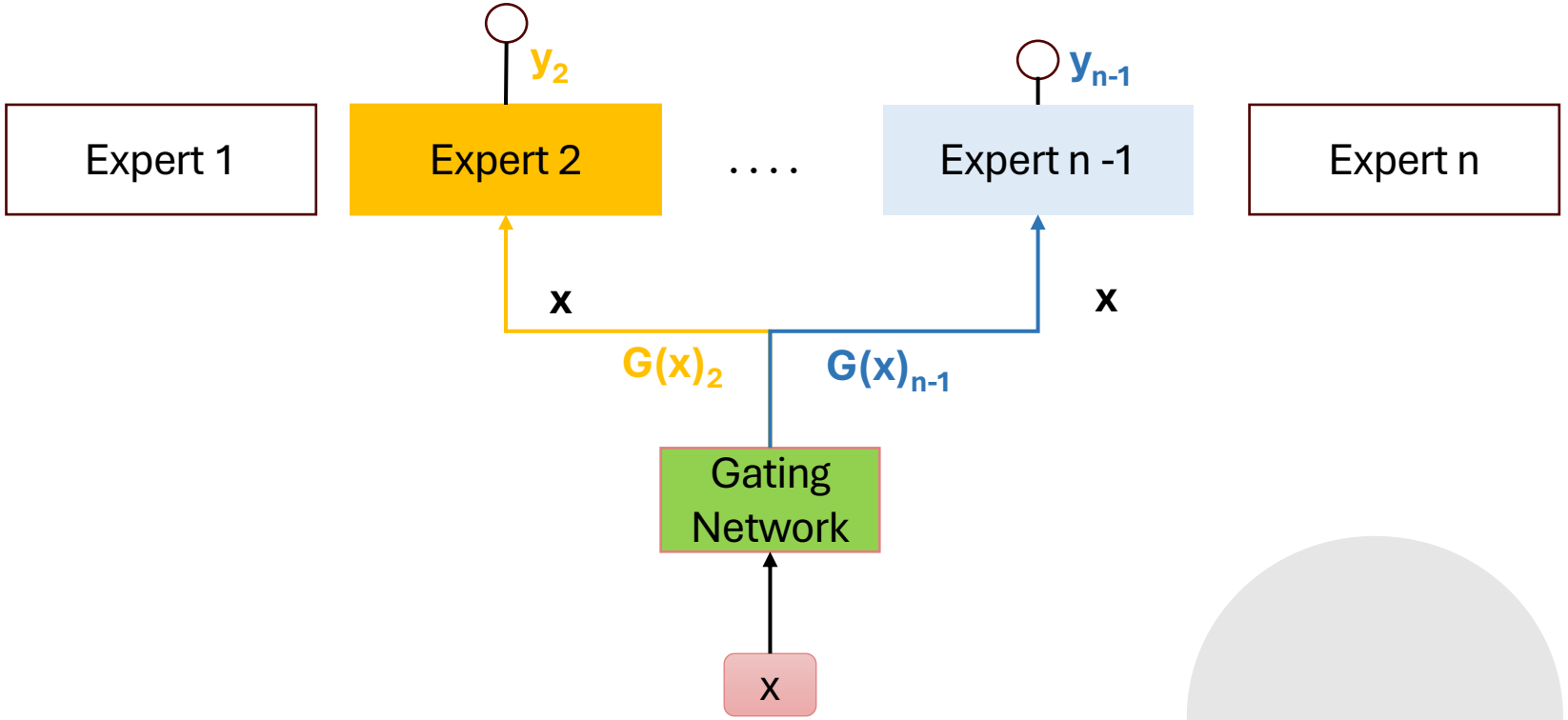




How to decide which expert to use?

Mixture of Experts

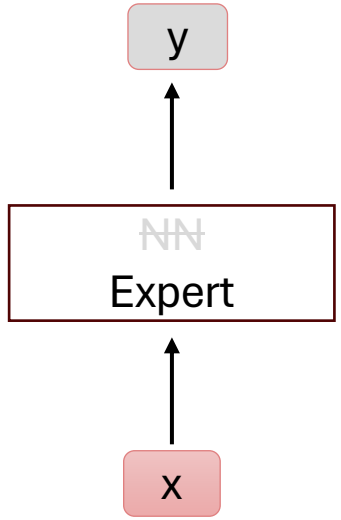


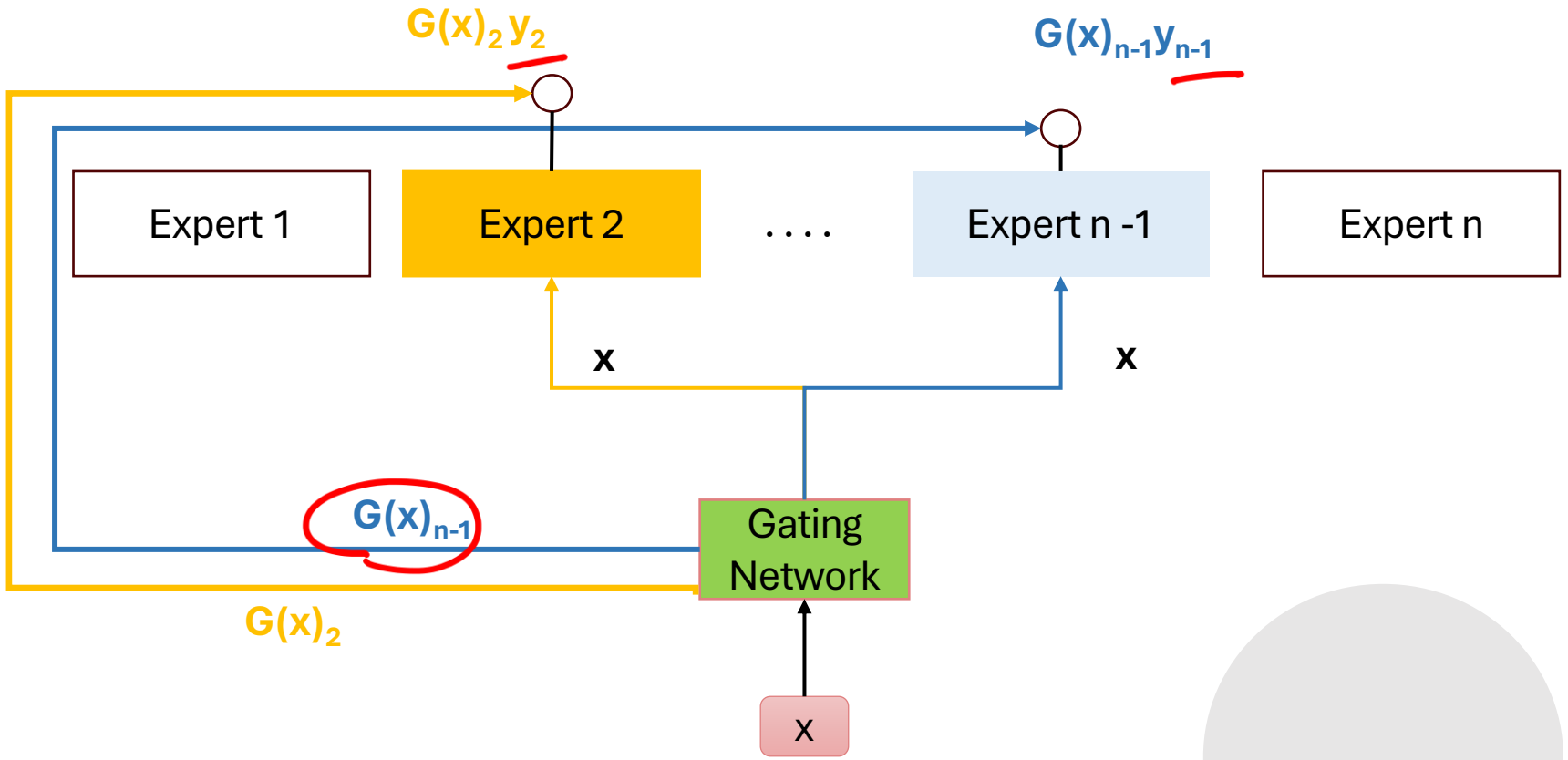


How to decide which expert to use?

Let's have a "Gating Network" that decides (*probabilistically*)

Mixture of Experts

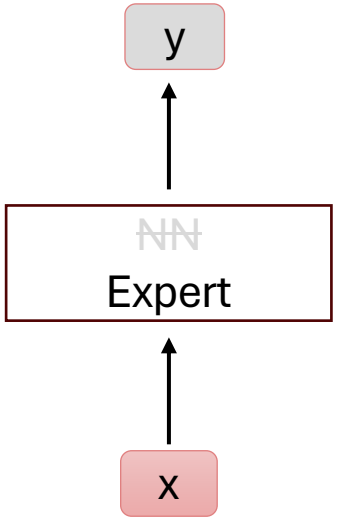


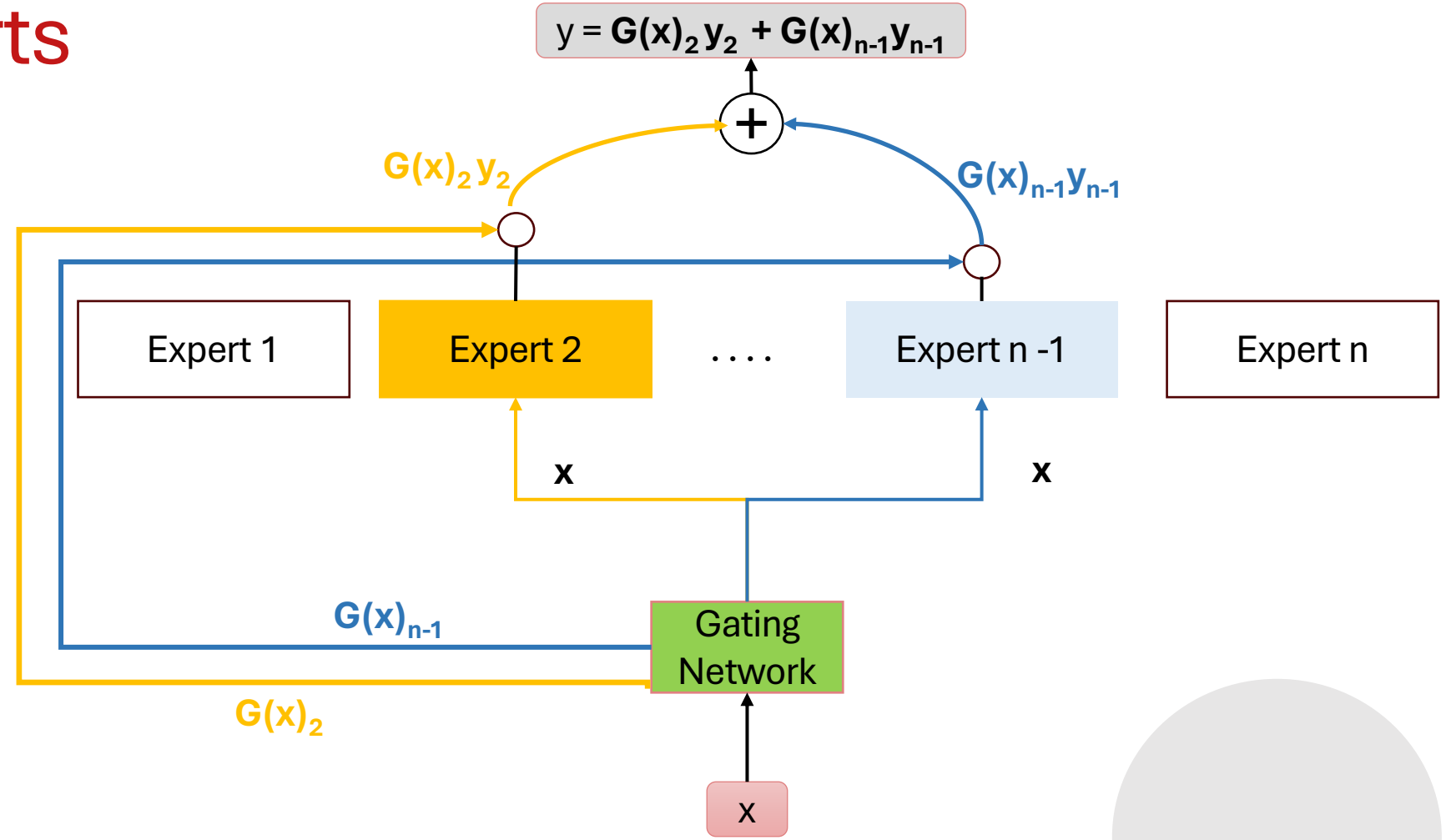


Mixture of Experts

How to decide which expert to use?

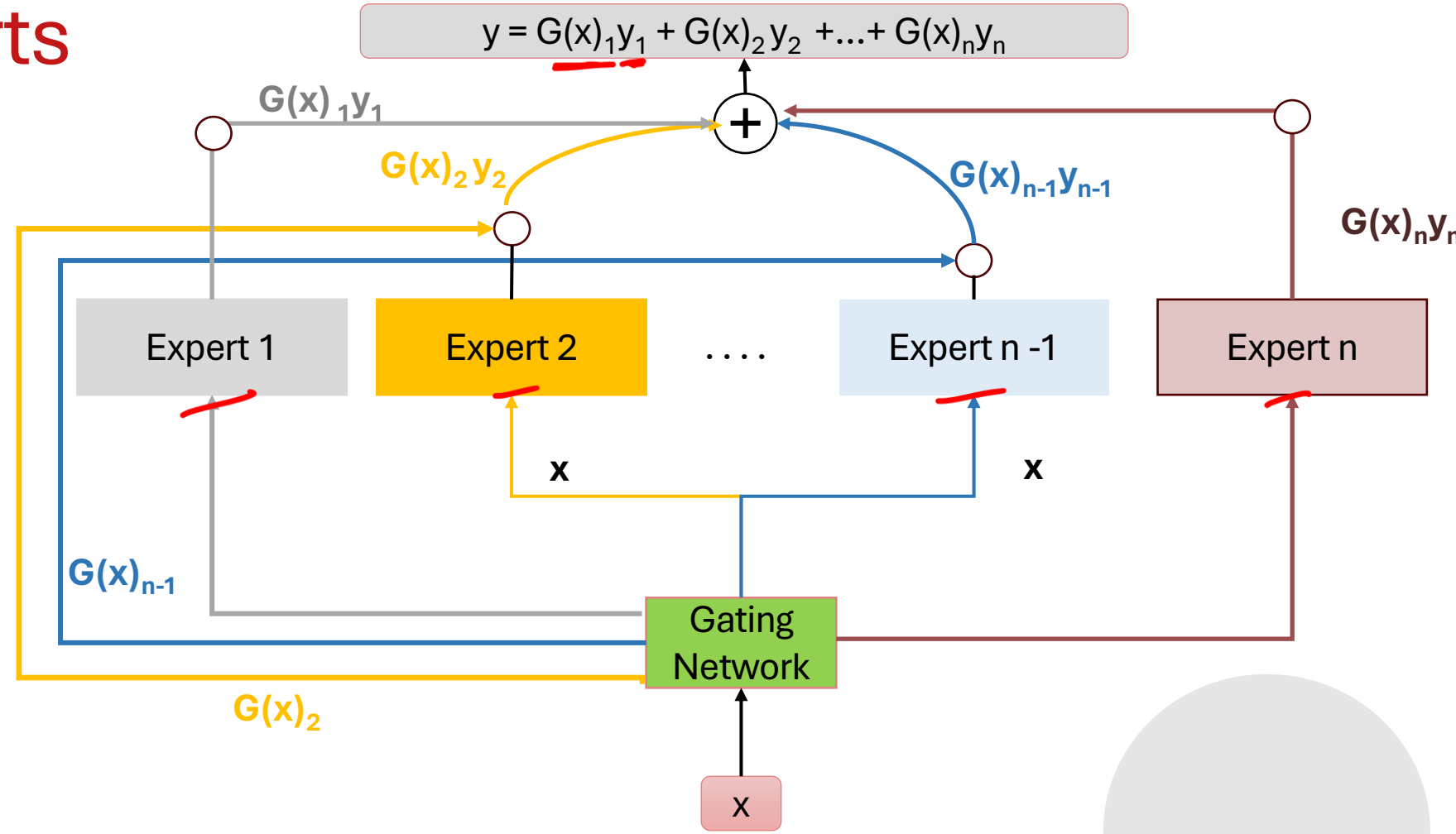
Let's have a "Gating Network" that decides (*probabilistically*)





Mixture of Experts

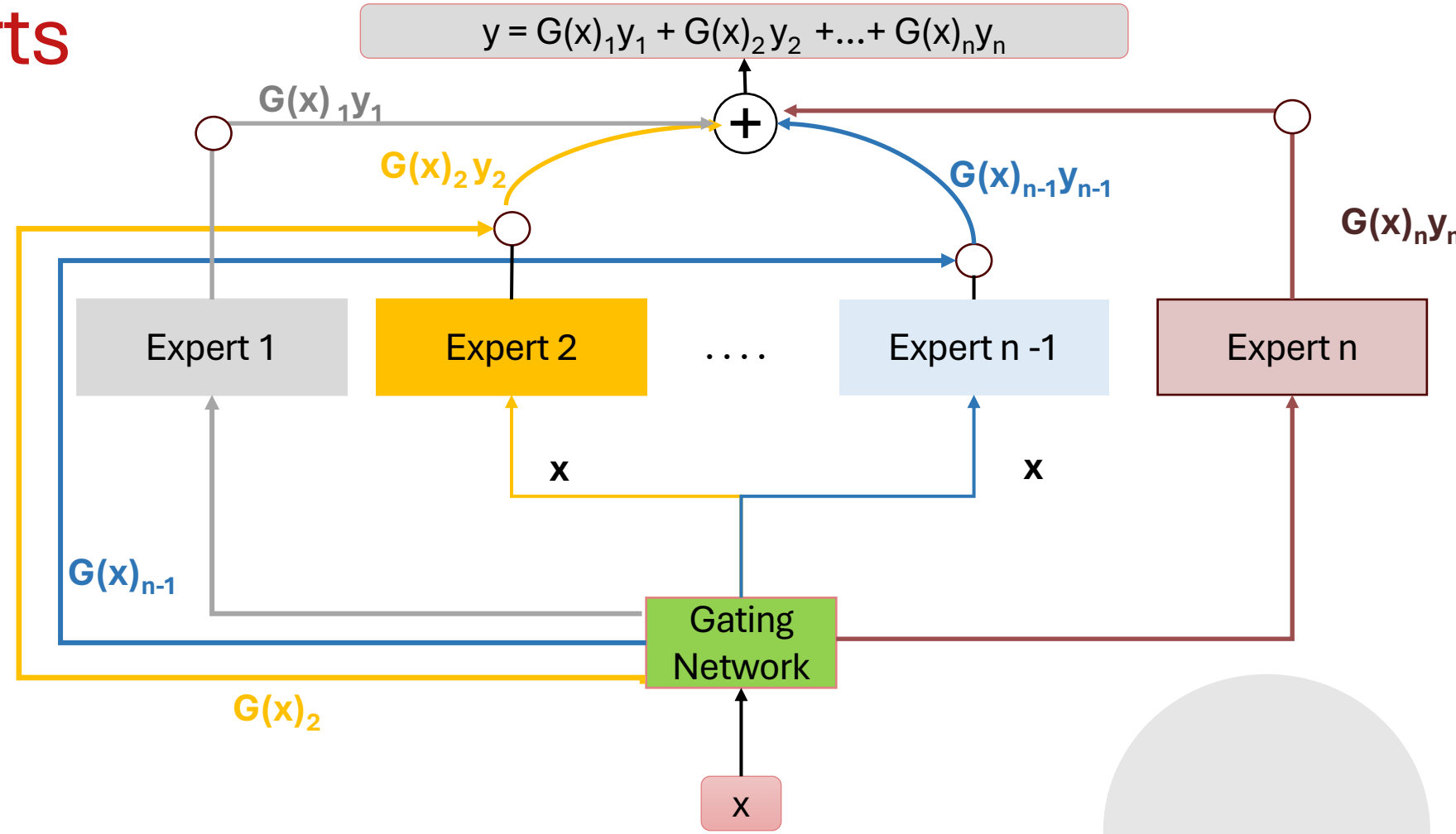
$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$



Mixture of Experts

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

$$y = \sum_{i=1}^{n} G(x)_i E_i(x)$$

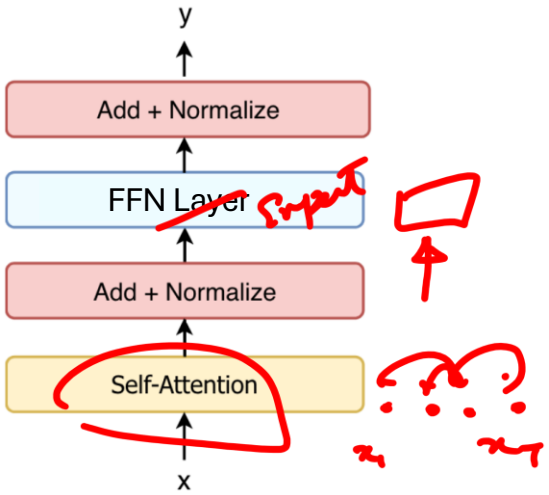


Mixture of Experts - Chronology

- Mixture of Experts Model [Jacobs et al., 1991; Jordan and Jacobs, 1994; Jordan et al., 1997; Tresp, 2001; Collobert et al., 2002;]
- MoE layer in Deep Learning [Eigen et al., 2013; Shazeer et al., 2017; Dauphin et al., 2017; Vaswani et al., 2017]
- MoE layer in Transformer based LLMs [Fedus et al., 2021; Du, Nan, et al., 2021]
- Mixtral-8x7B [Jiang et al. 2024] - Apache 2.0 license; surpassed GPT-3.5 Turbo, Claude-2.1, Gemini Pro, and Llama 2 70B - chat model on human benchmarks
- GPT-4, Deepseek-V3/R1, MiniMax-01

Content credits: https://www.youtube.com/watch?v=TwHPxUAuqy4

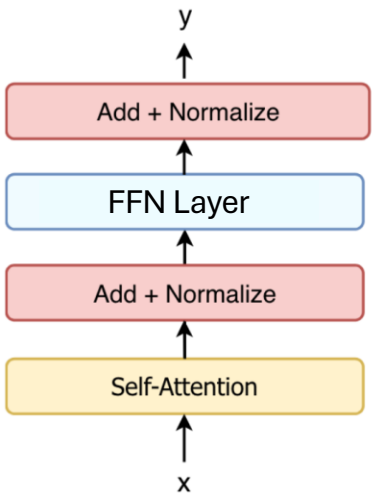
How/where to use Mixture of Experts in Transformers?



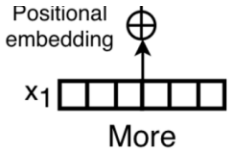
Dense Transformer Block

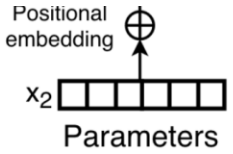
Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

Mixture of Experts as a Layer



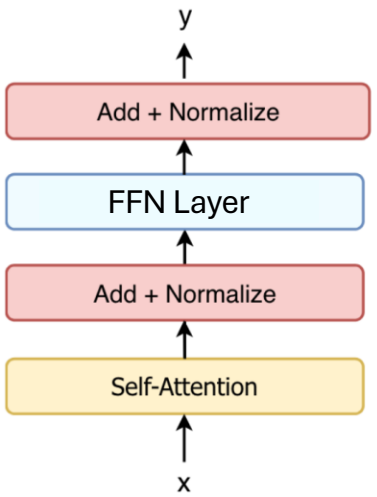
Dense Transformer Block



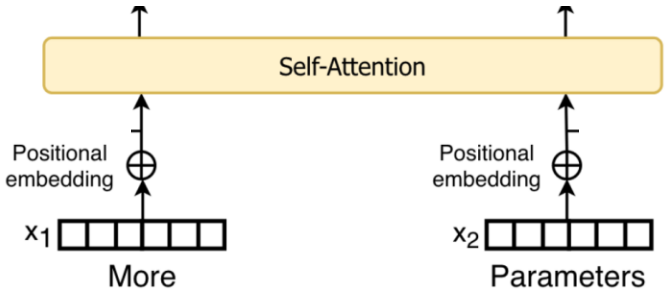


Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

Mixture of Experts as a Layer

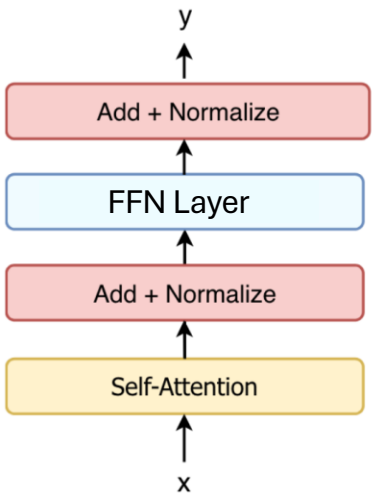


Dense Transformer Block

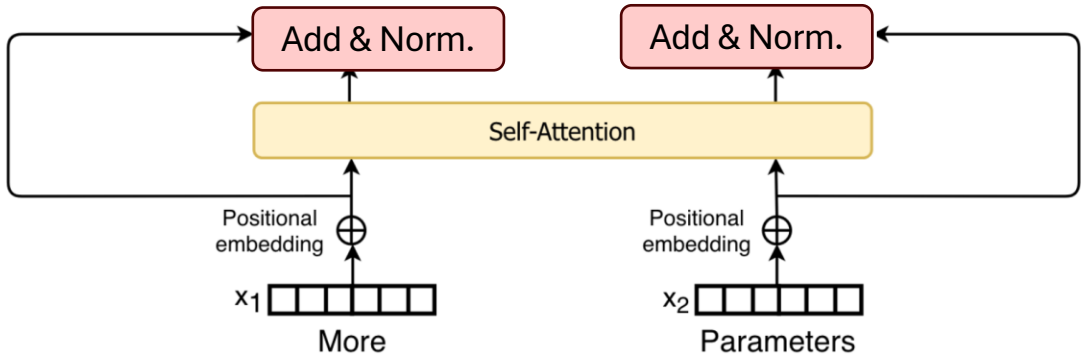


Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

Mixture of Experts as a Layer

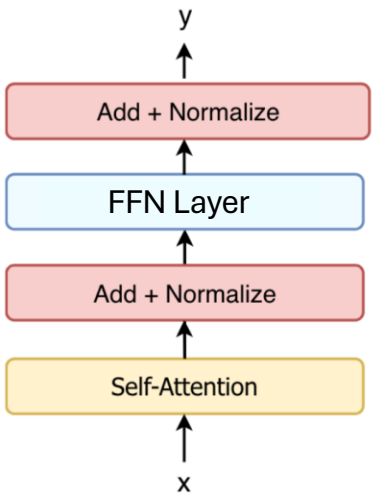


Dense Transformer Block

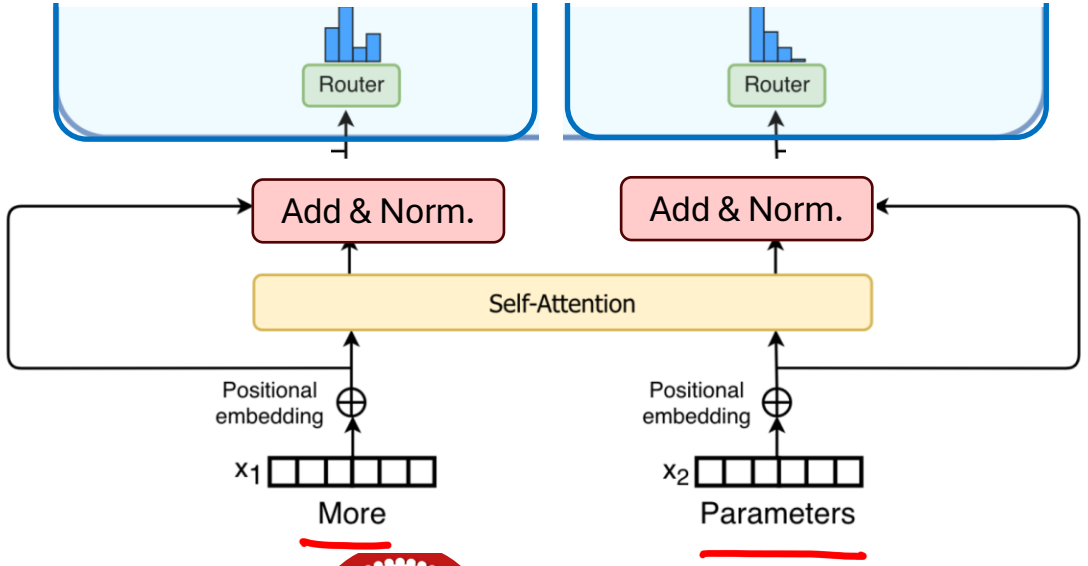


Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

Mixture of Experts as a Layer

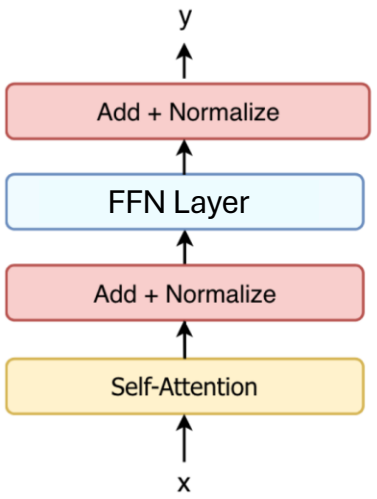


Dense Transformer Block

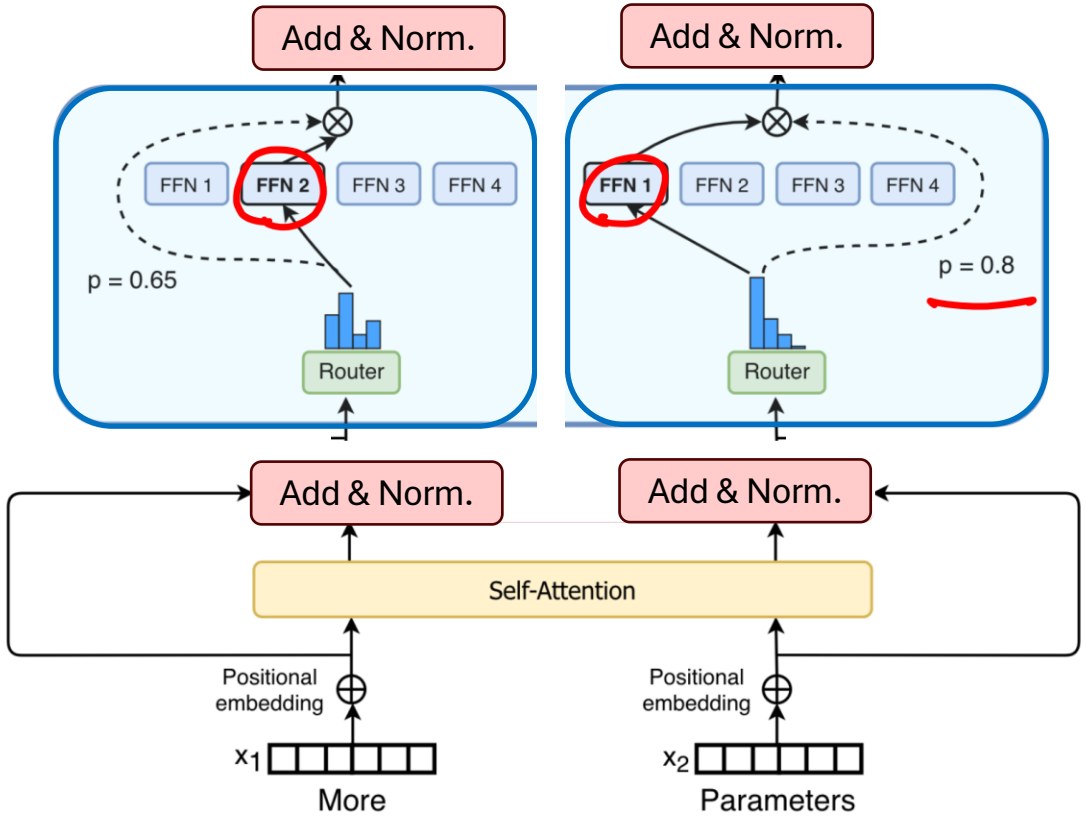


Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

Mixture of Experts as a Layer

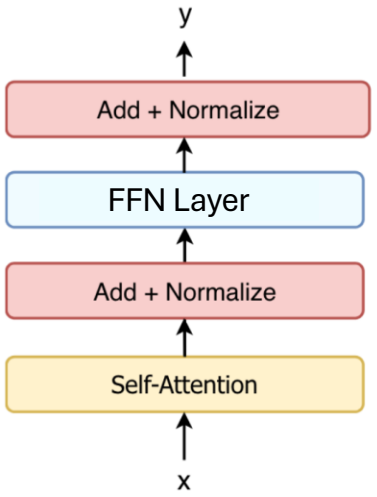


Dense Transformer Block

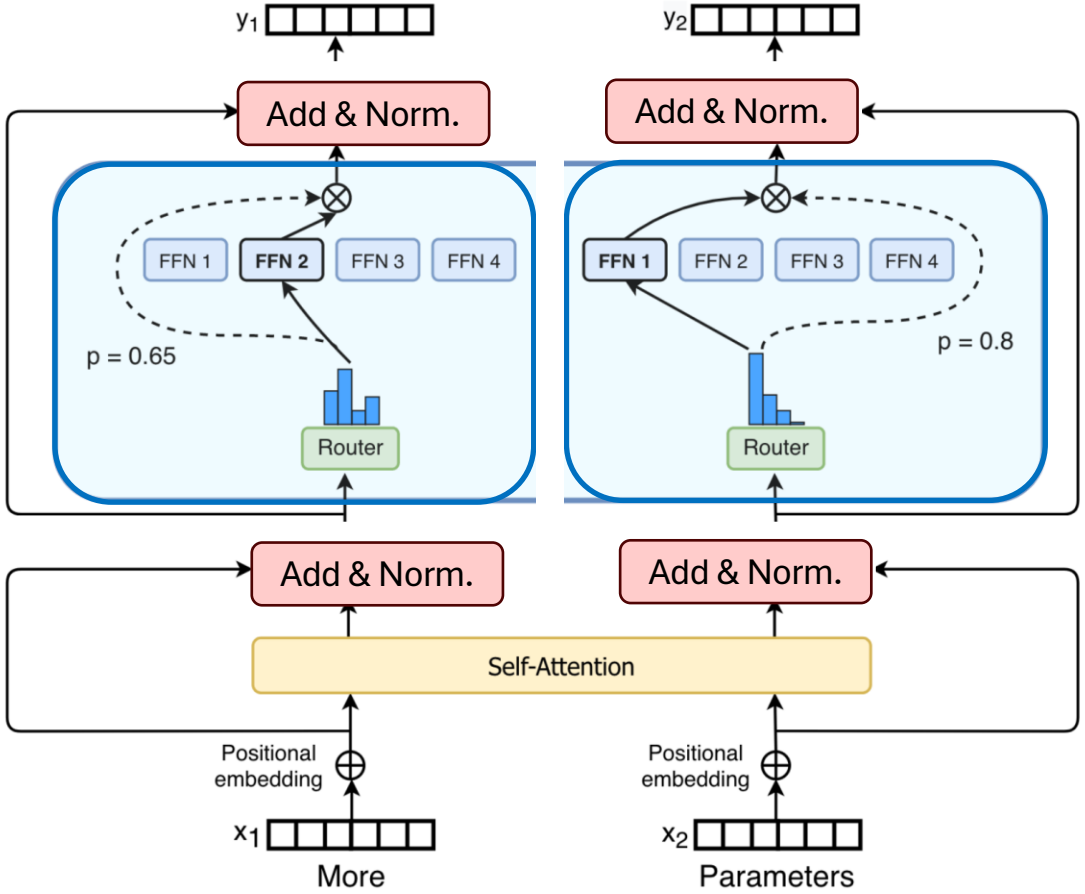


Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

Mixture of Experts as a Layer



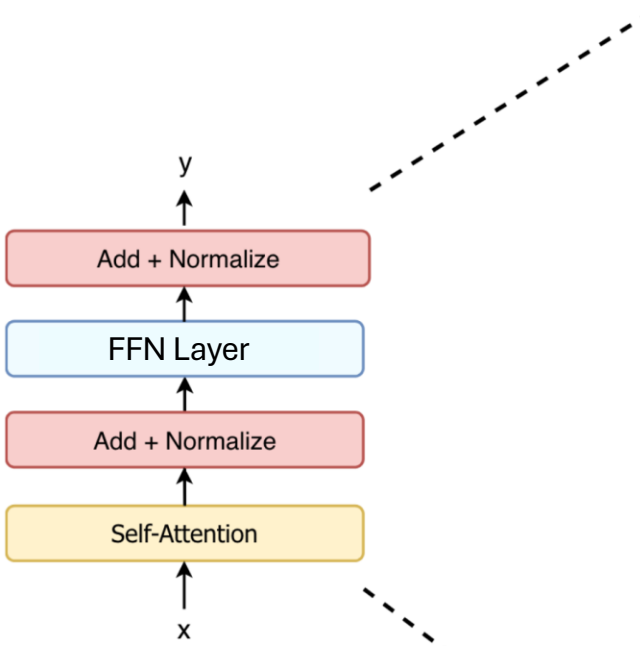
Dense Transformer Block



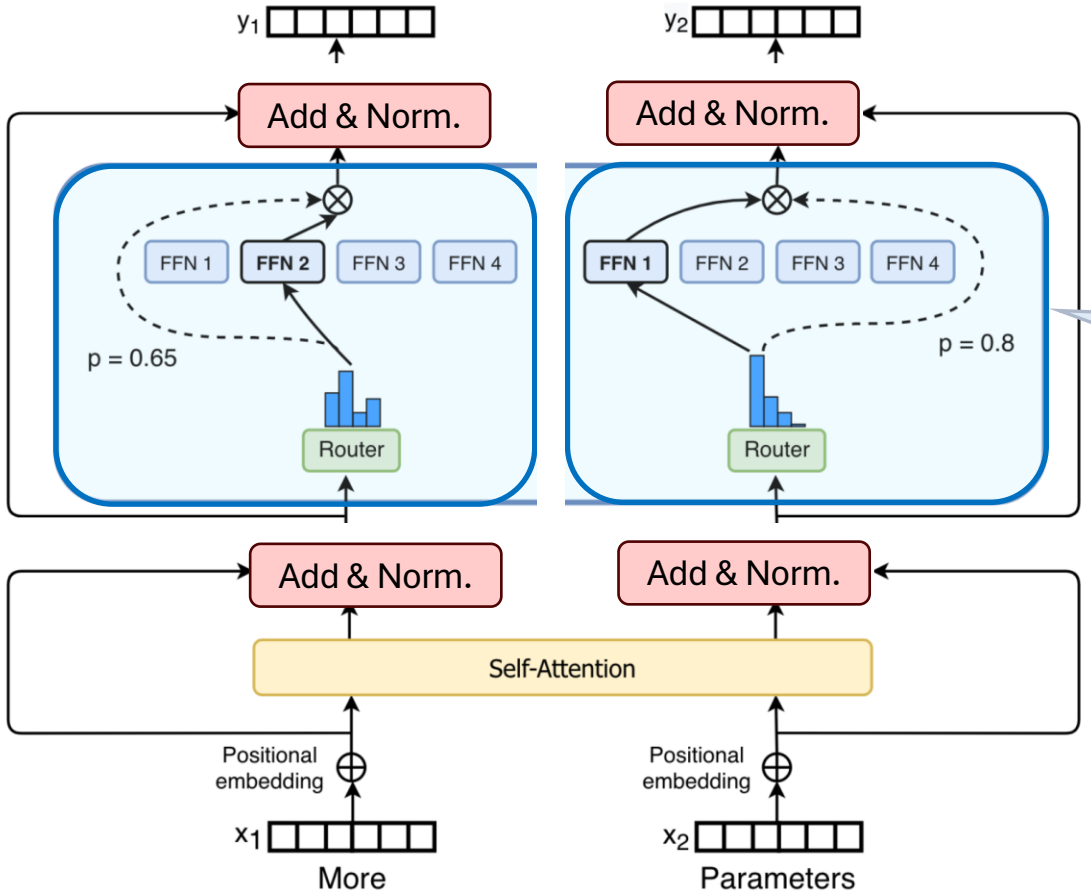
$4h^2 + 4h^2$
$+4h \quad +h$
$8h^2 + 5h \quad 10+8h^2$
$3h^2 + 1h^2$
$4h^2$

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

Mixture of Experts as a Layer



Dense Transformer Block



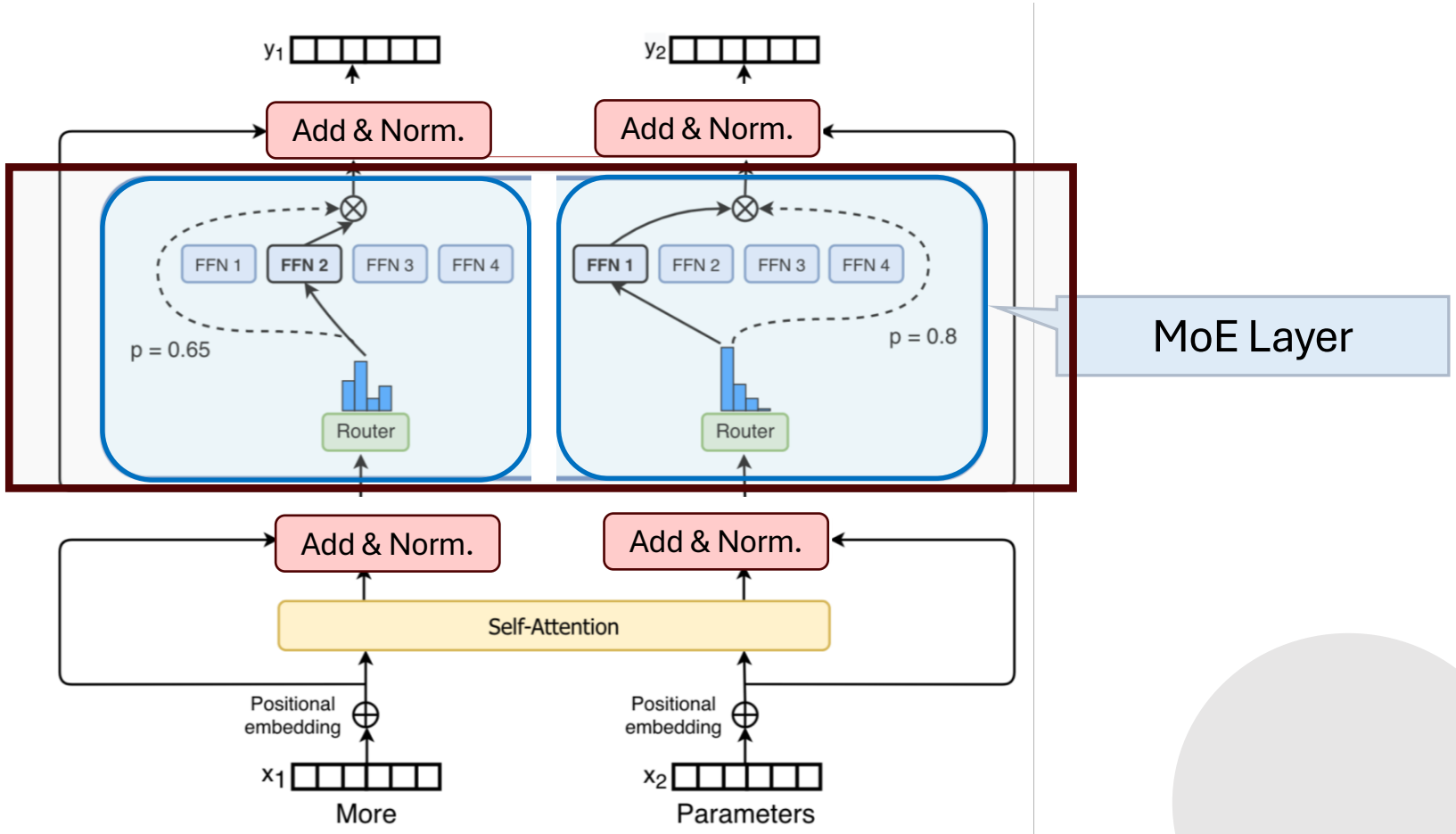
MoE Layer

Sparsely Activated MoE Transformer Block

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

Sparse MoE Layer – Greedy expert selection

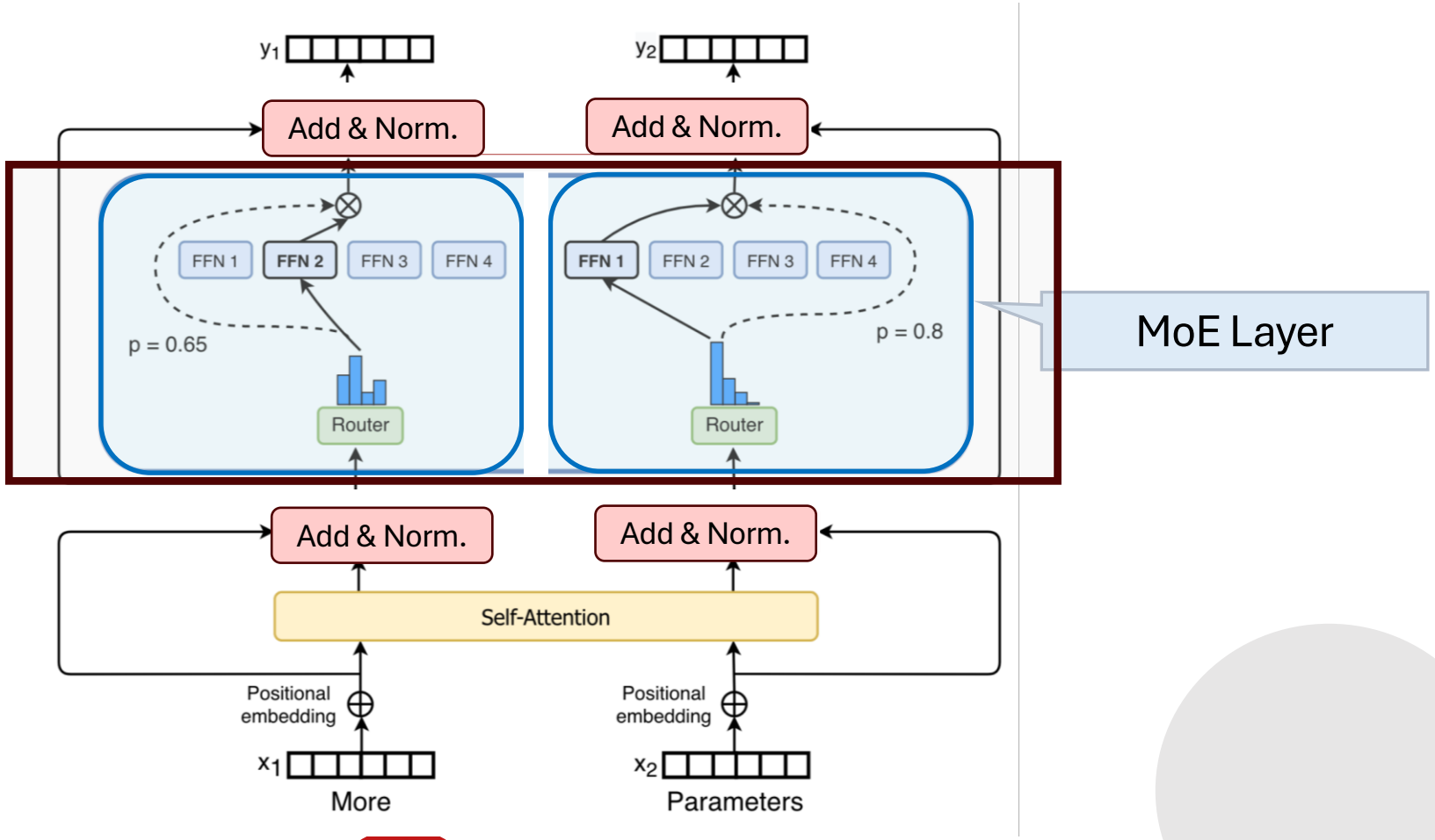
$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$



Sparse MoE Layer – Greedy expert selection

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

$$i* = argmax\ G_\sigma(x)$$

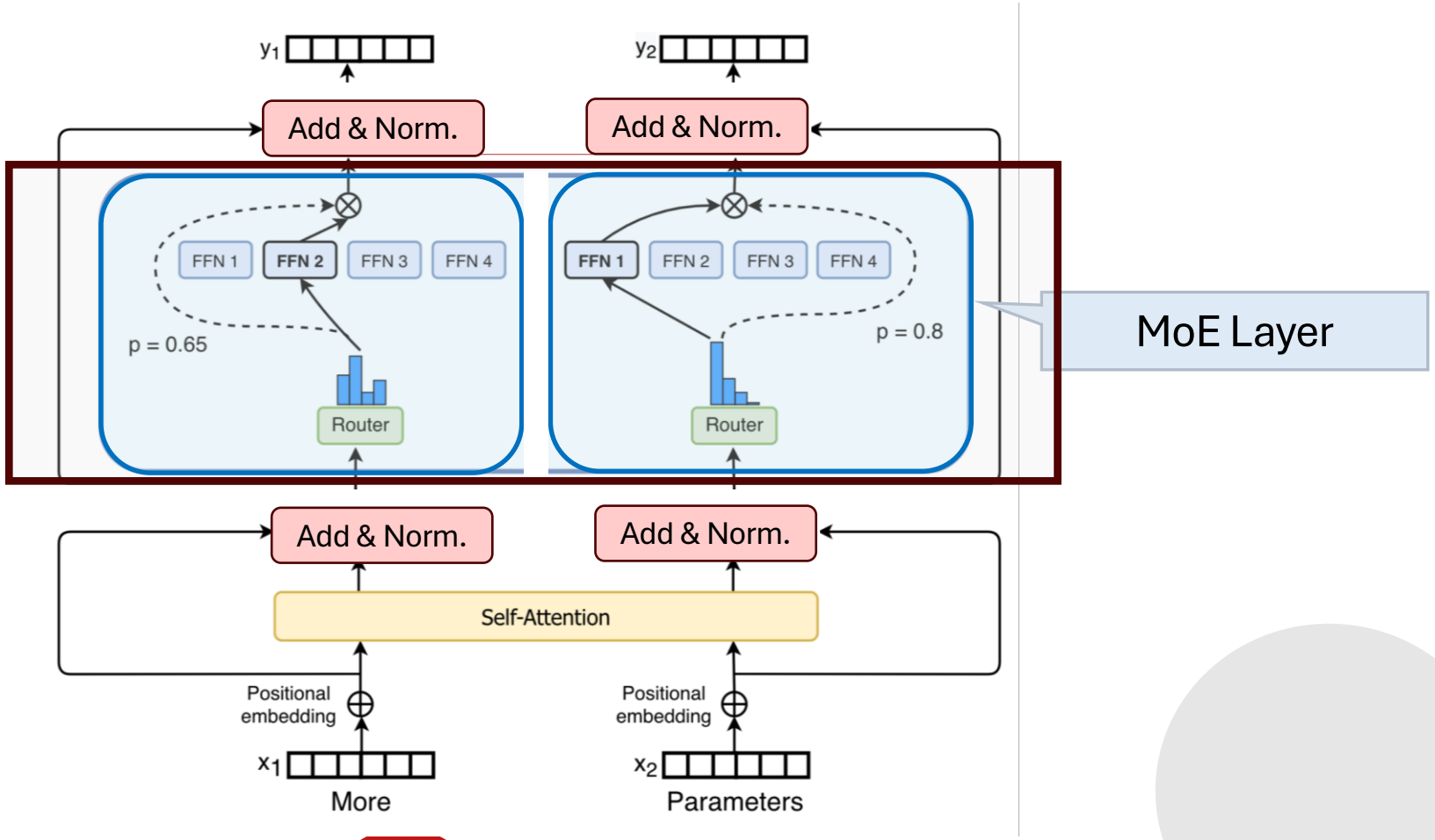


Sparse MoE Layer – Greedy expert selection

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

$$i* = argmax\ G_\sigma(x)$$

$$y = G(x)_{i*} E_{i*}(x)$$



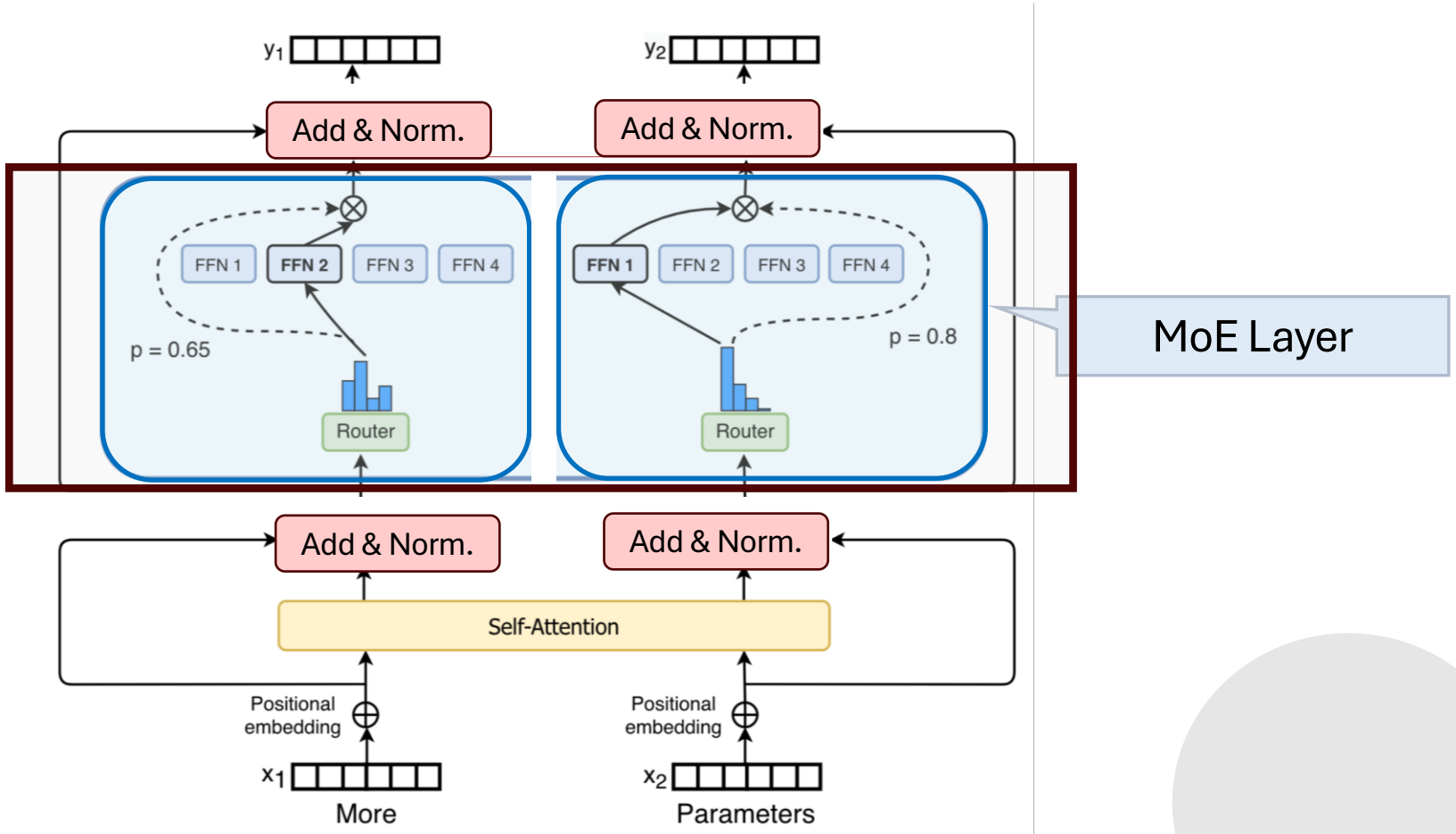
Sparse MoE Layer – Greedy expert selection

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

$$i* = argmax\ G_\sigma(x)$$

$$y = G(x)_{i*} E_{i*}(x)$$

$$\frac{\nabla L}{\nabla E_{i*}(x)} = G(x)_{i*} \frac{\nabla L}{\nabla y}$$



Sparse MoE Layer – Greedy expert selection

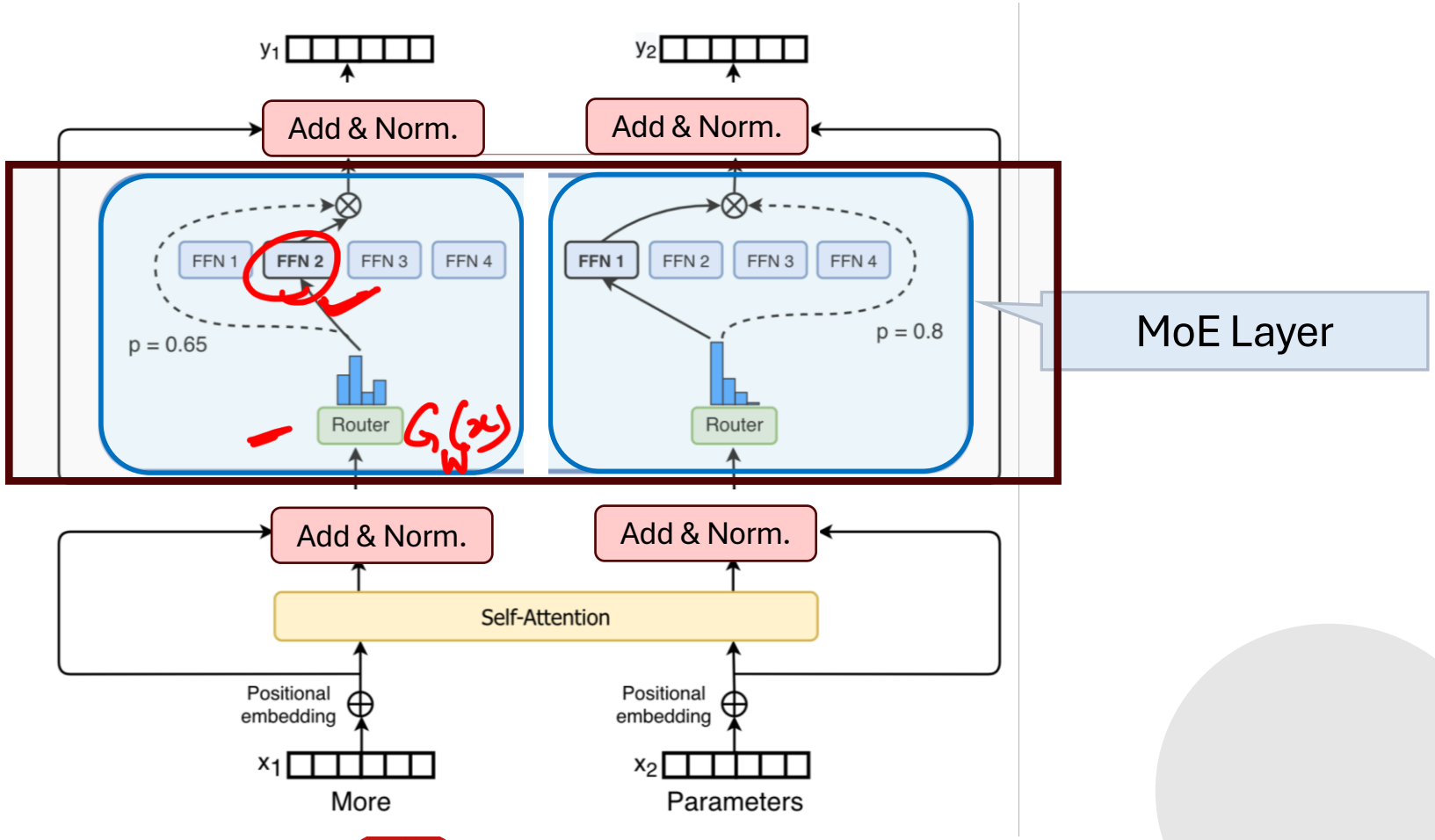
$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

$$i* = argmax\ G_\sigma(x)$$

$$y = G(x)_{i*} E_{i*}(x)$$

$$\frac{\nabla L}{\nabla E_{i*}(x)} = G(x)_{i*} \frac{\nabla L}{\nabla y}$$

$$\frac{\nabla L}{\nabla G(x)_{i*}} = (E_{i*}(x))^T \frac{\nabla L}{\nabla y}$$



Sparse MoE Layer – Greedy expert selection

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

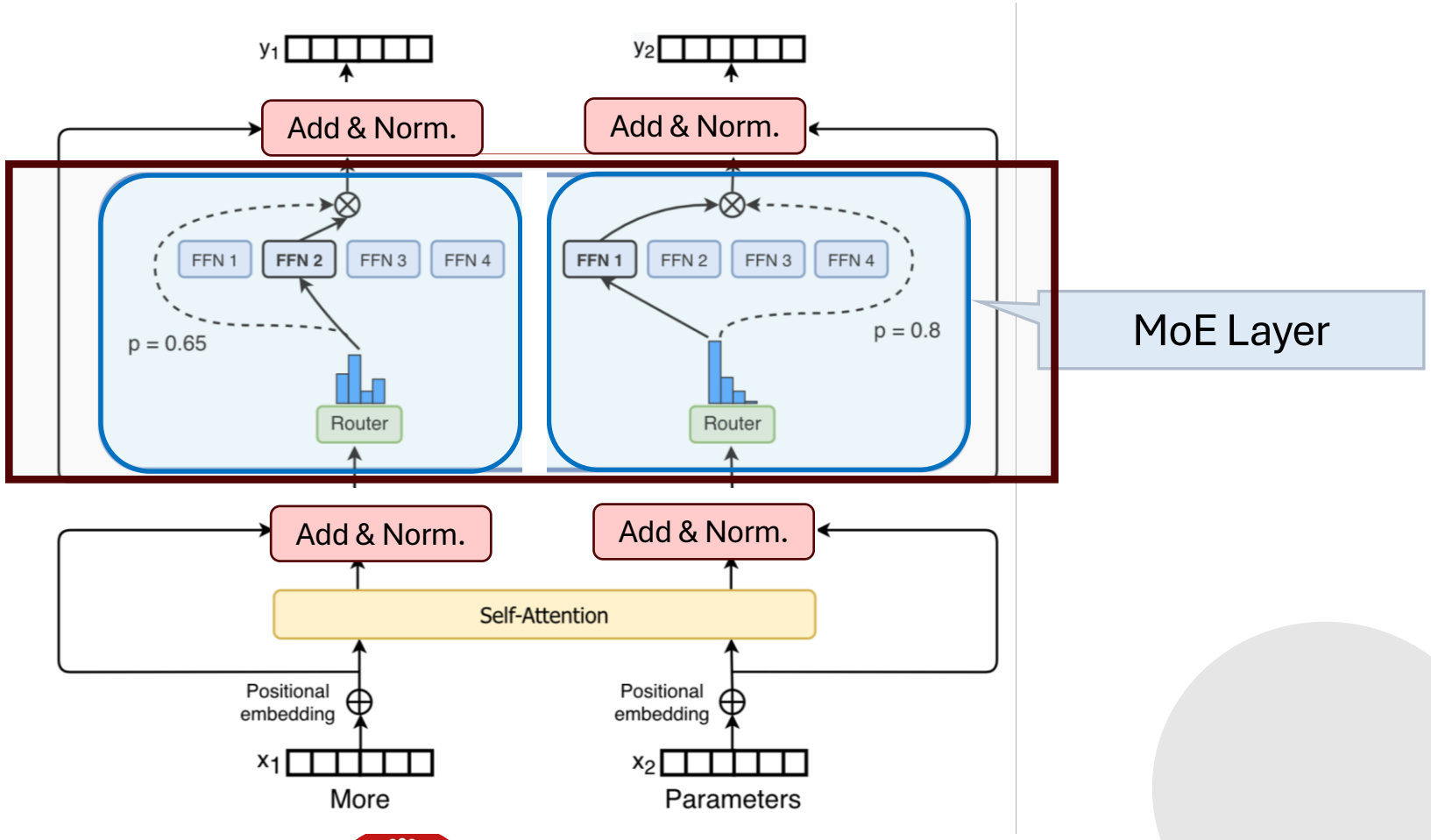
$$i* = argmax\ G_\sigma(x)$$

$$y = G(x)_{i*} E_{i*}(x)$$

$$\frac{\nabla L}{\nabla E_{i*}(x)} = G(x)_{i*} \frac{\nabla L}{\nabla y}$$

$$\frac{\nabla L}{\nabla G(x)_{i*}} = (E_{i*}(x))^T \frac{\nabla L}{\nabla y}$$

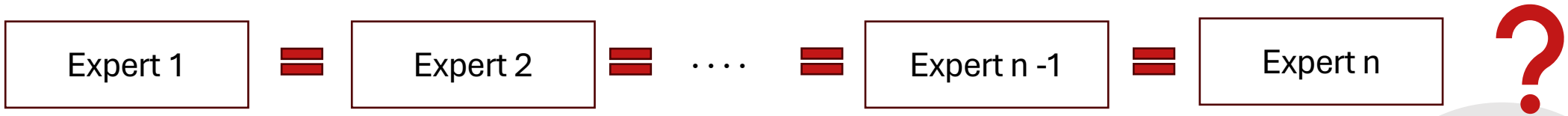
$$\frac{\nabla L}{\nabla E_i(x)} = \mathbf{0}\ ;\ \frac{\nabla L}{\nabla G(x)_i} = 0\ for\ i \neq i*$$



Why do experts not collapse in practice?

- All experts have:
  - Same architecture
  - Same initialization scheme
  - Trained with same optimizer

- So why don't they collapse? I.e.



Why do experts not collapse in practice?

Towards Understanding the Mixture-of-Experts Layer in Deep Learning

Zixiang Chen
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
chenzx19@cs.ucla.edu

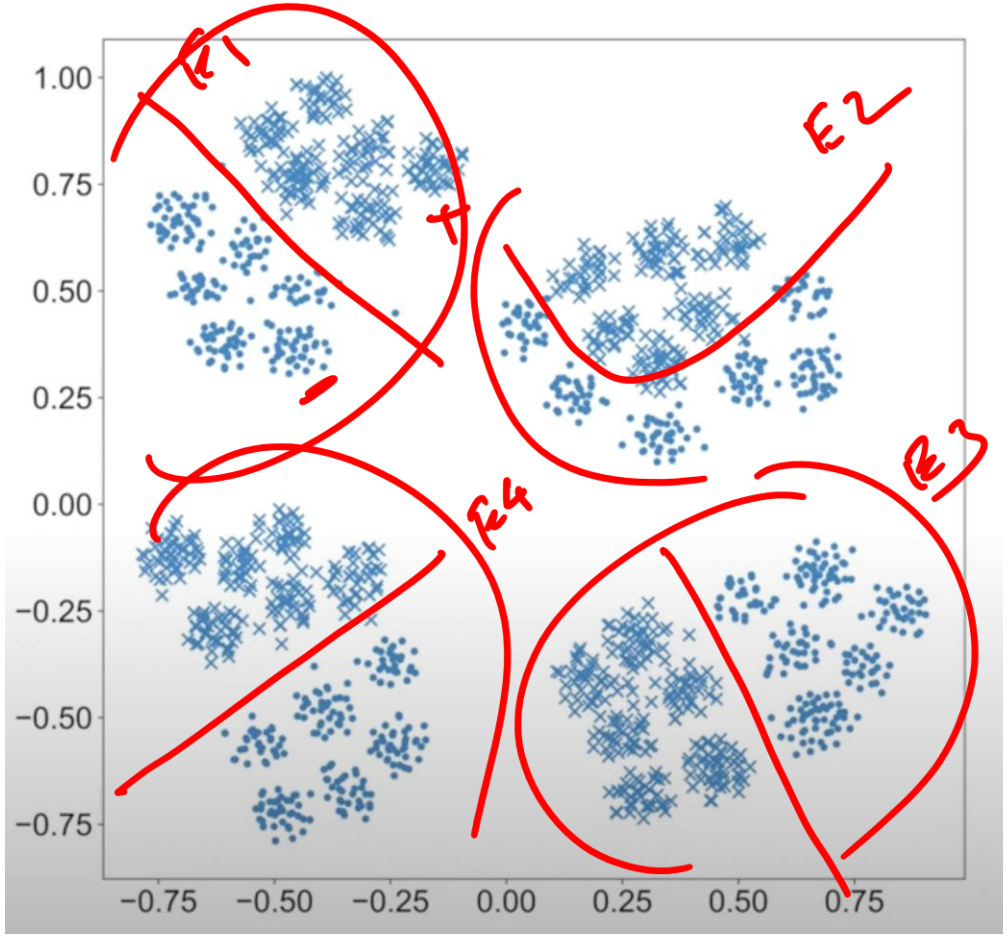
Yihe Deng
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
yihedeng@cs.ucla.edu

Yue Wu
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
ywu@cs.ucla.edu

Quanquan Gu
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
qgu@cs.ucla.edu

Yuanzhi Li
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
yuanzhil@andrew.cmu.edu

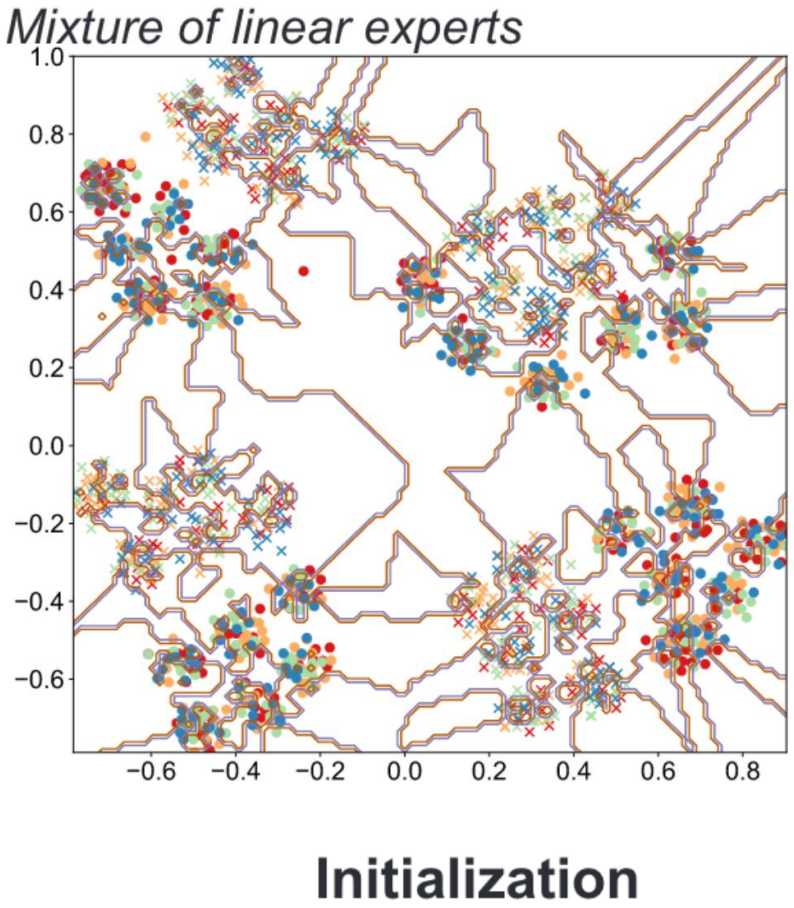
Why do experts not collapse in practice?



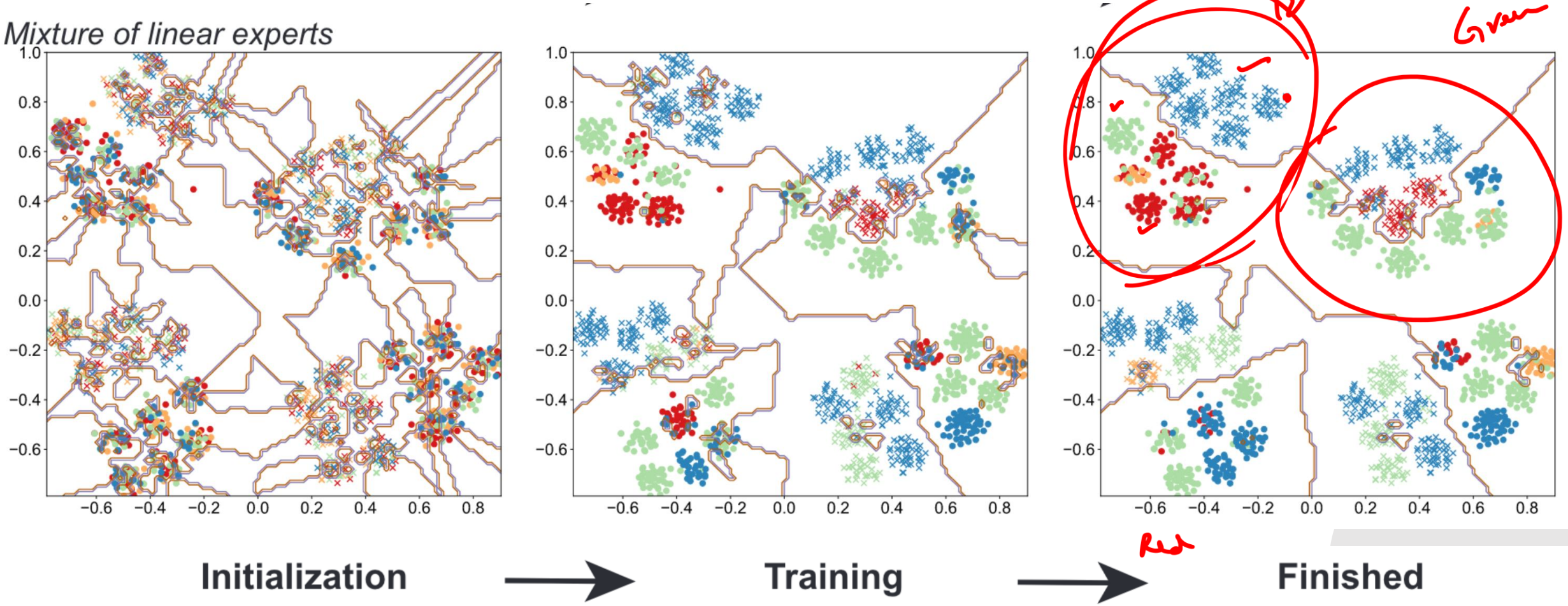
- ✓ Synthetically generated 50 dim. data
- ✓ Visualization in 2-D space
- ✓ 4 clusters
- ✓ Each cluster is linearly separable
- ✓ Ideally, a mixture of 4 linear experts sufficient for classification

$E \rightarrow W_{ei} x$

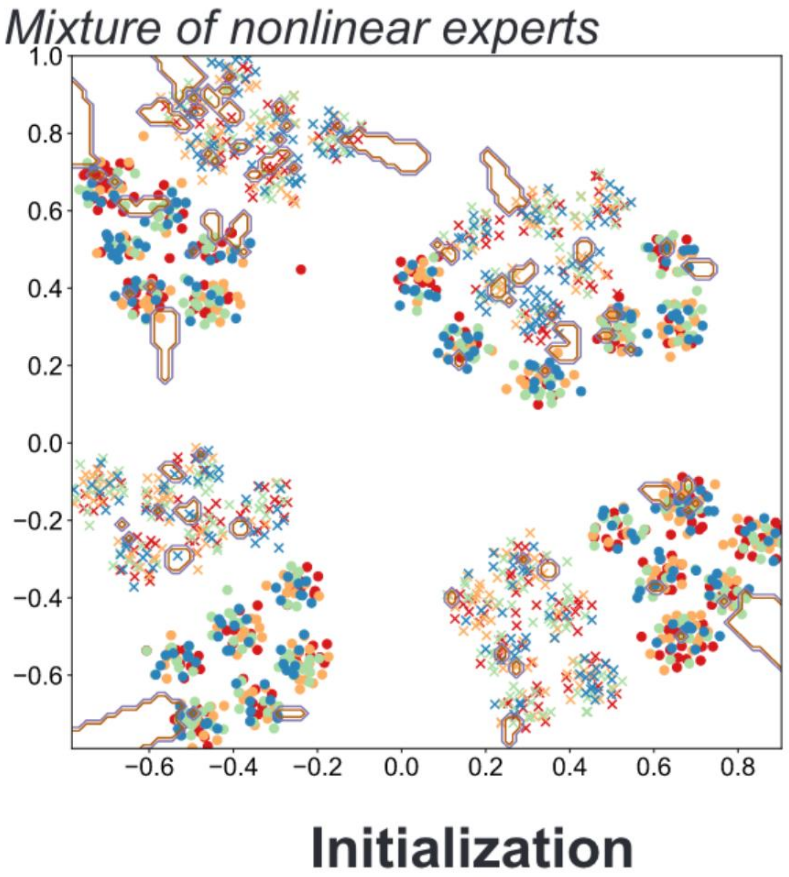
Why do experts not collapse in practice?



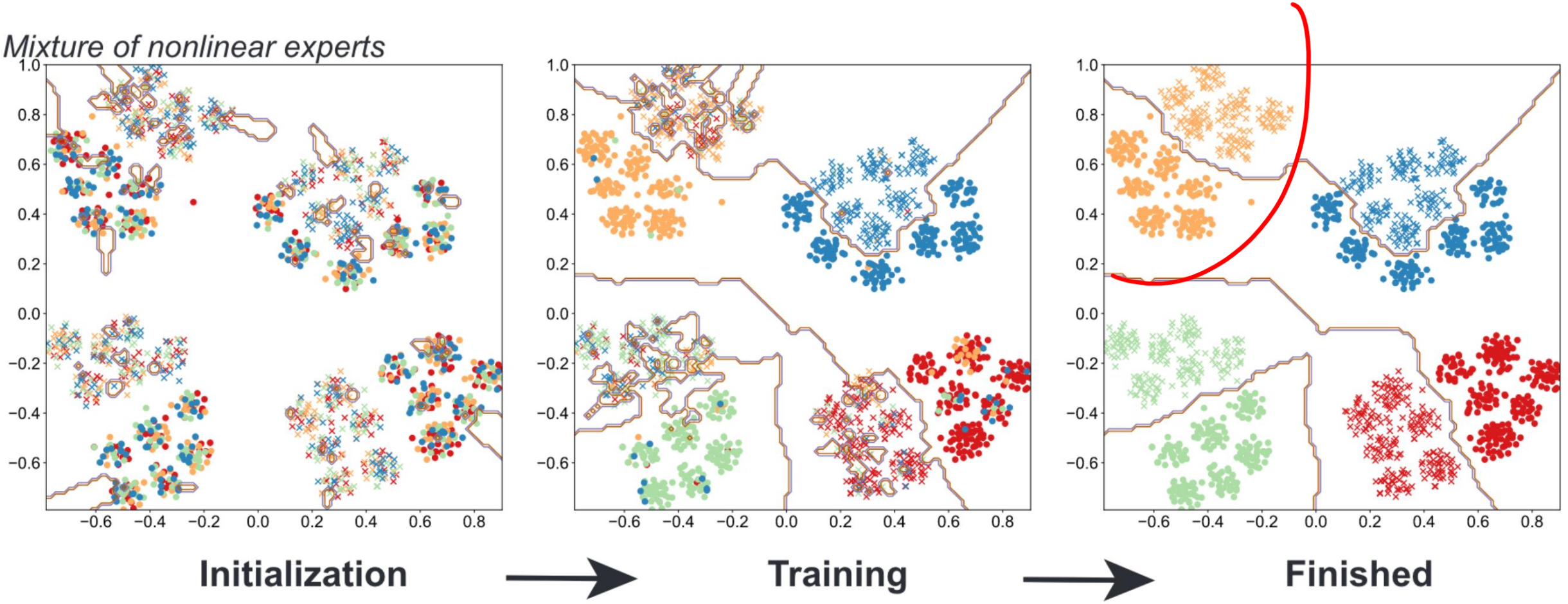
Why do experts not collapse in practice?



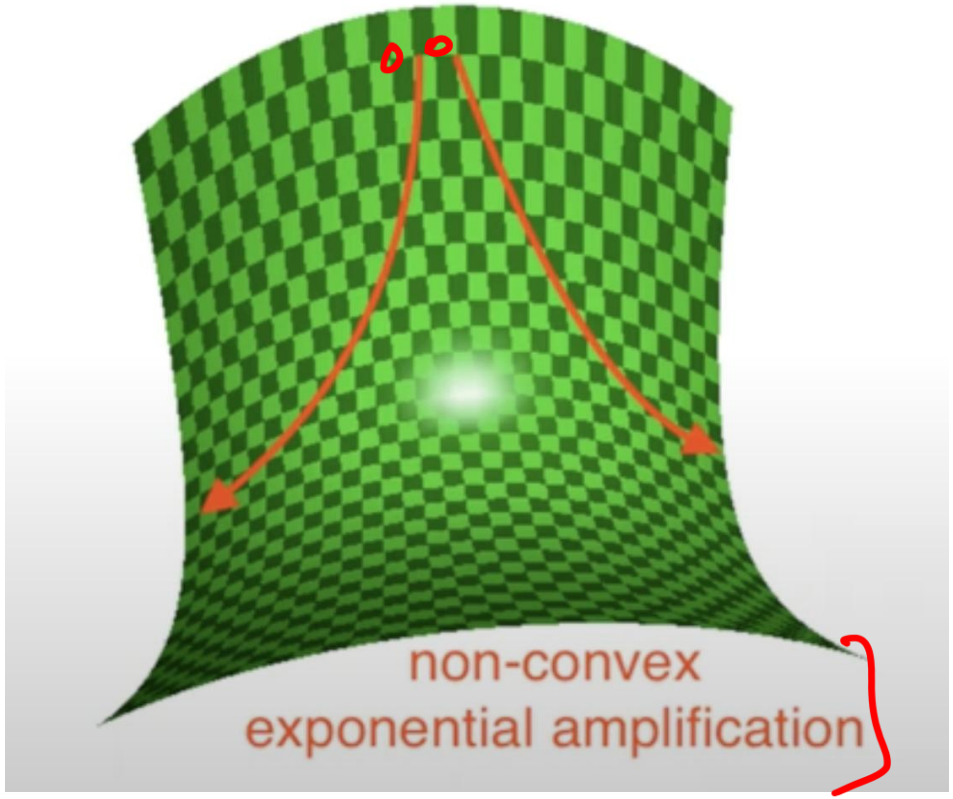
Why do experts not collapse in practice?

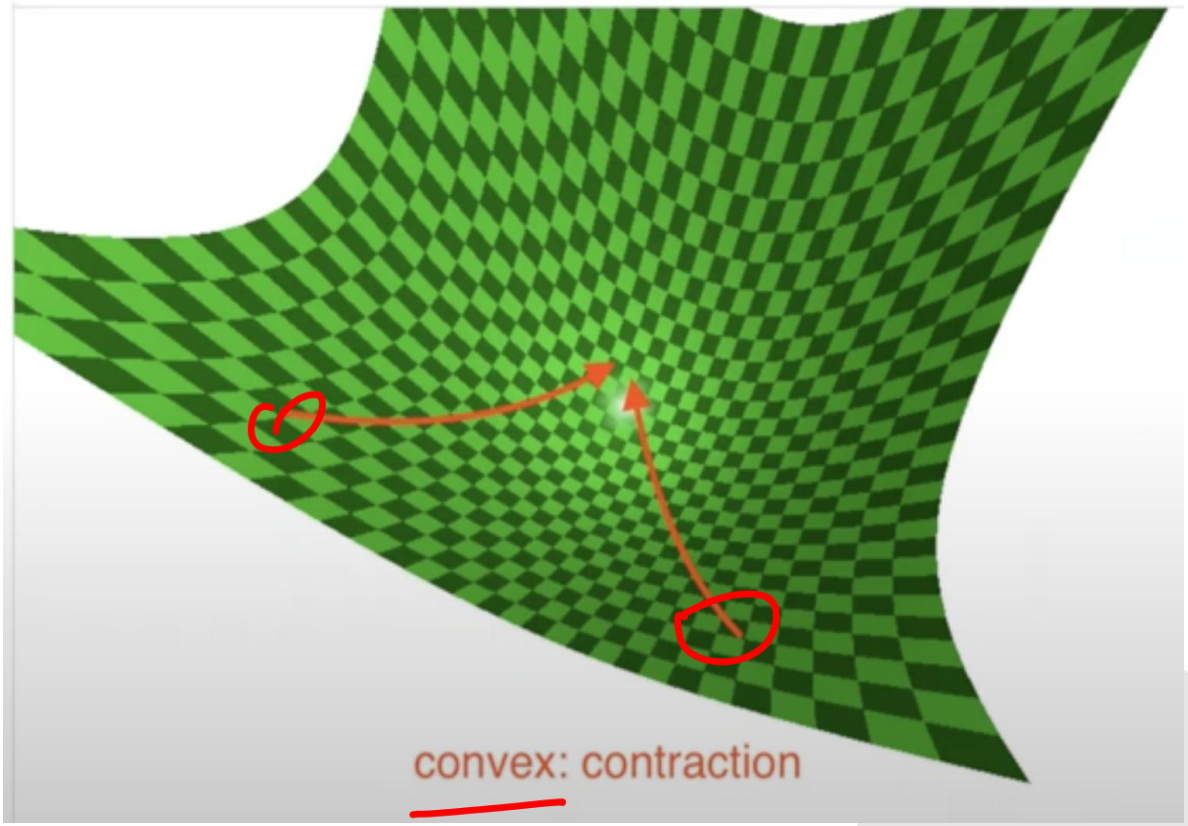


Why do experts not collapse in practice?



Why do experts not collapse in practice?





Pros and Cons of Sparse MoE Layer

Pros

👍 Increased model parameters

👍 Efficient pretraining due to conditional (sparse) computation

👍 Faster inference

Cons

👎 High memory requirement - all parameters need to be loaded in vRAM (GPU memory)

Pros and Cons of Sparse MoE Layer

Pros

👍 Increased model parameters

👍 Efficient pretraining due to conditional (sparse) computation

👍 Faster inference

Cons

👎 High memory requirement - all parameters need to be loaded in vRAM (GPU memory)

Can we exploit Parallelism?

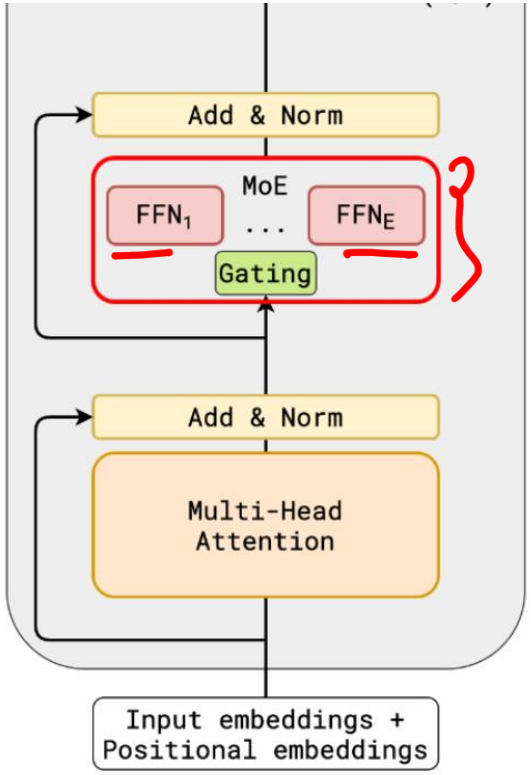
Parallelism ...

- Data Parallelism / Fully Sharded Data Parallelism
- Tensor Parallelism (w/ sequence parallelism)
- Context Parallelism
- Pipeline Parallelism

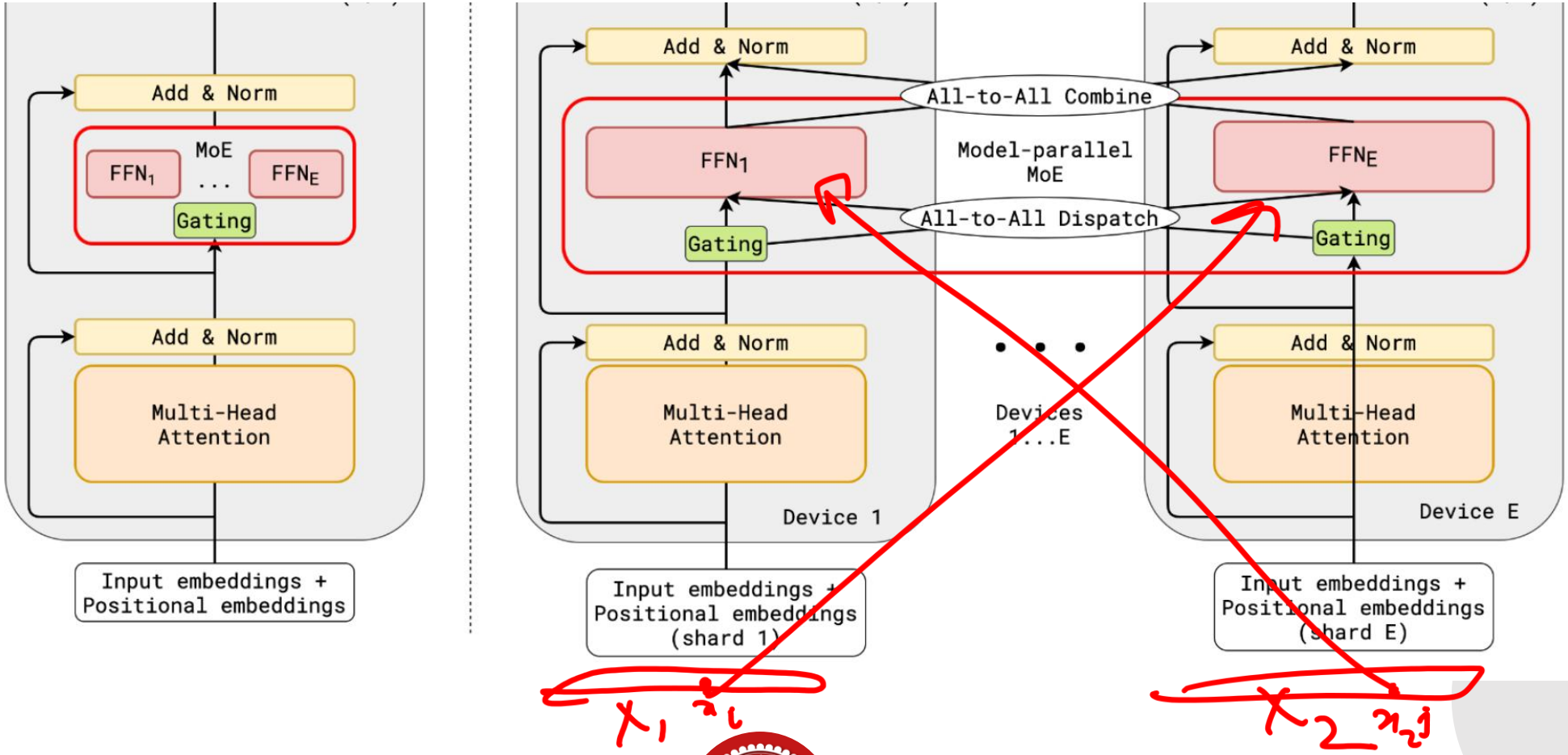
Parallelism ...

- Data Parallelism / Fully Sharded Data Parallelism
- Tensor Parallelism (w/ sequence parallelism)
- Context Parallelism
- Pipeline Parallelism
- **Expert Parallelism?**

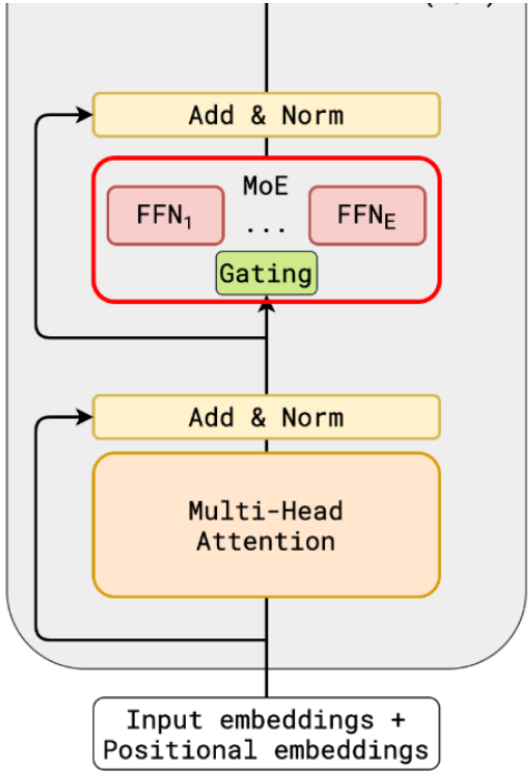
Expert Parallel for Sparse MoEs

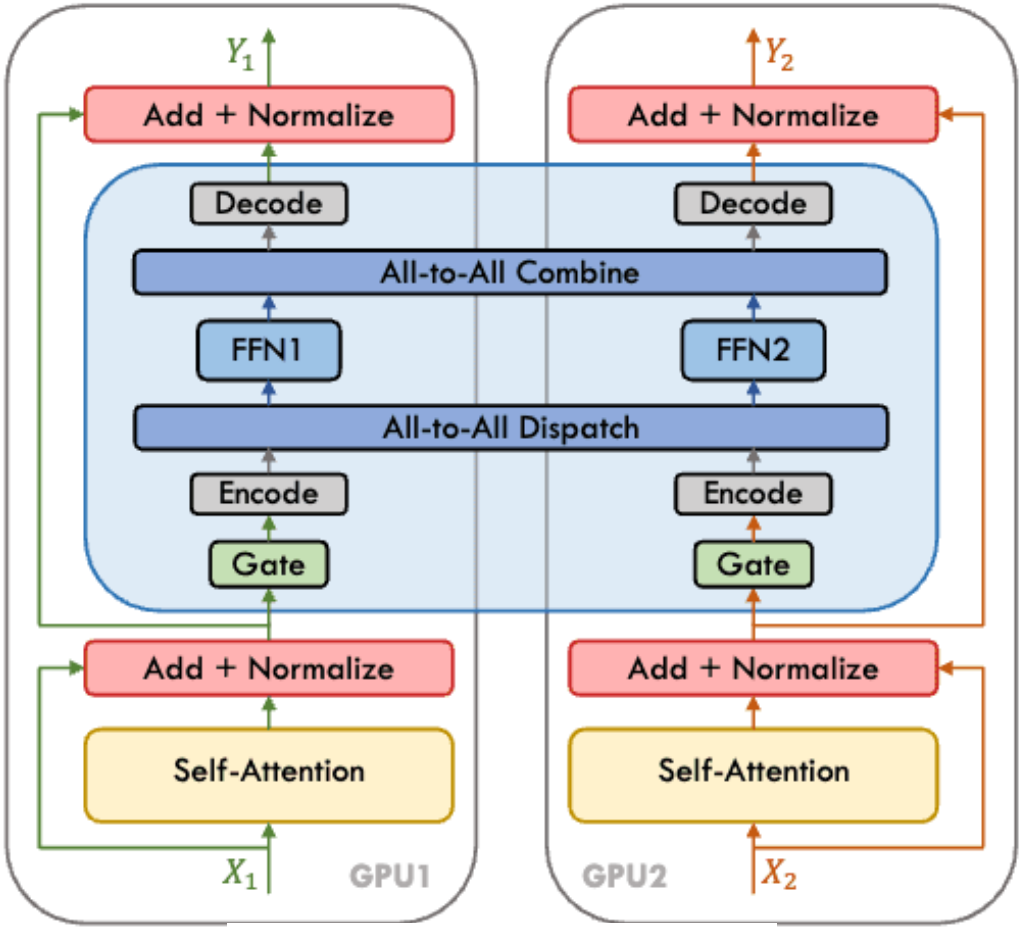


Expert Parallel for Sparse MoEs



Expert Parallel for Sparse MoEs

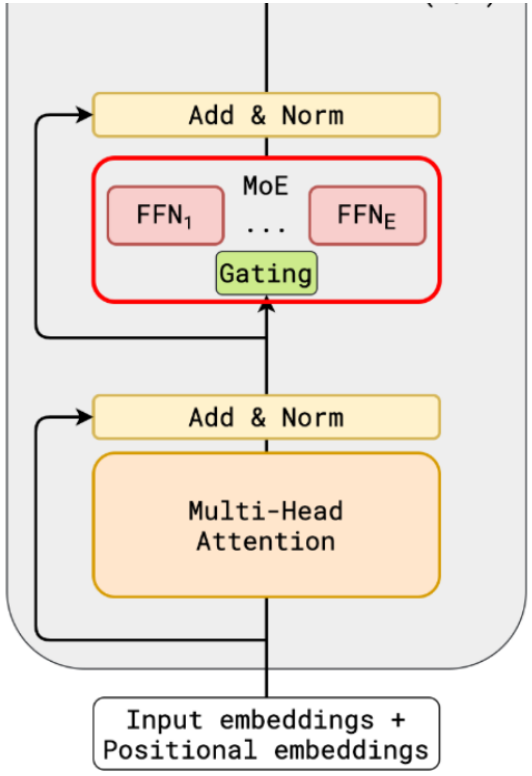




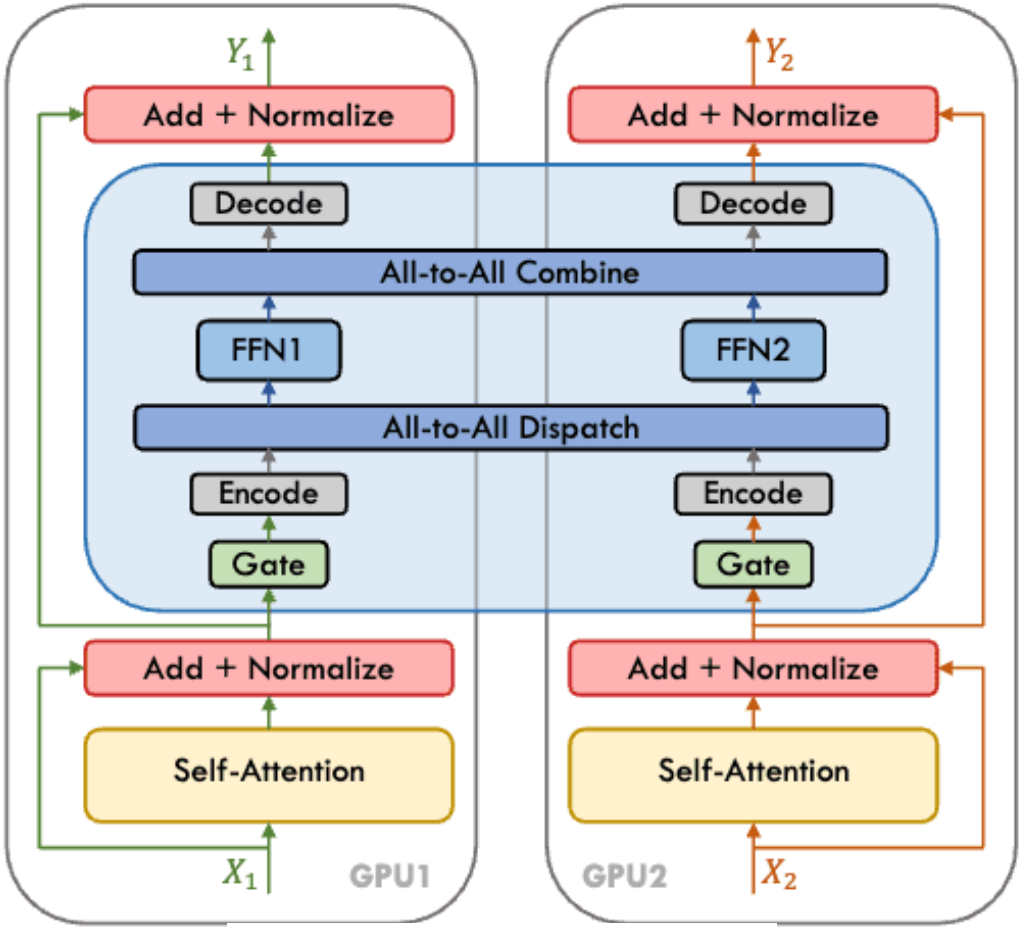
Data + Expert Parallelism

Used in conjunction with other parallelism strategies

Expert Parallel for Sparse MoEs



But what if all tokens are routed to single expert?



Data + Expert Parallelism

Pros and Cons of Sparse MoE Layer

Pros

👍 Increased model parameters

👍 Efficient pretraining due to conditional (sparse) computation

👍 Faster inference

Cons

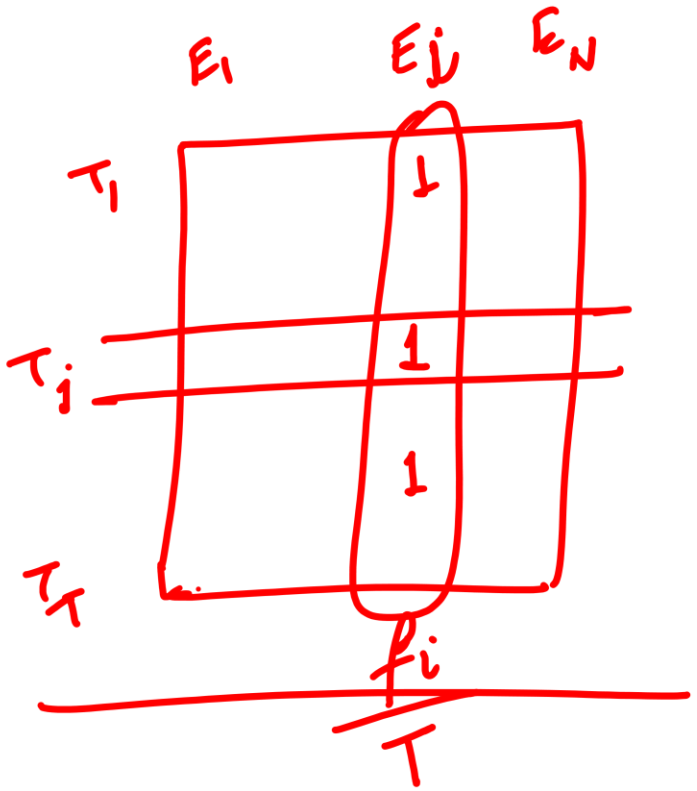
👎 High memory requirement - all parameters need to be loaded in vRAM (GPU memory)

👎 Unstable training

- 😟 Router collapse– router sends all tokens to the same expert
- 😟 May diverge

Load Balancing Loss

- N experts; T tokens in a batch $\mathfrak{B}$
- f_i: Fraction of tokens dispatched to expert i

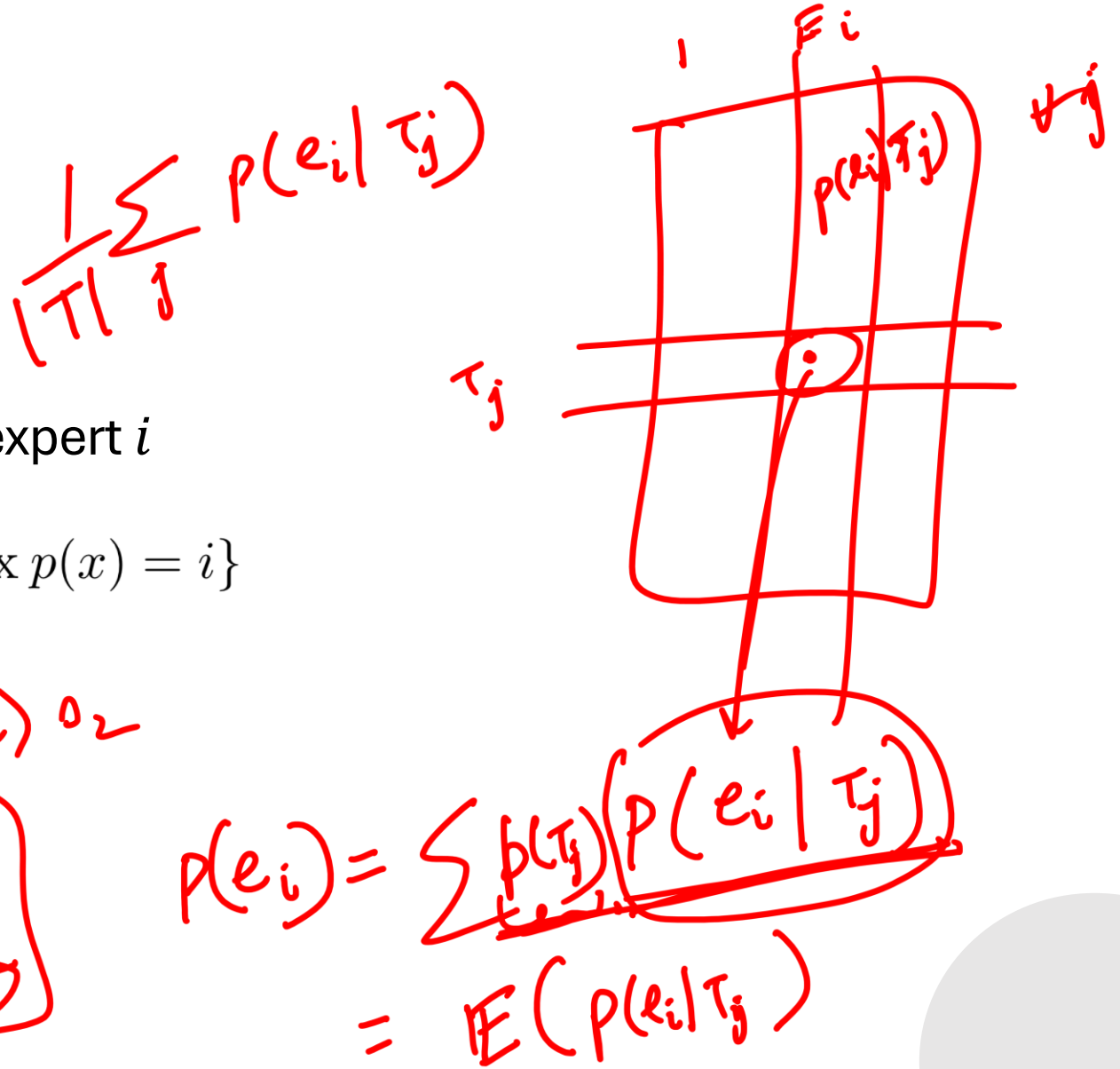


Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity
https://www.youtube.com/watch?v=U8J32Z3qV8s&t=2816s

Load Balancing Loss

- N experts; T tokens in a batch $\mathfrak{B}$
- f_i: Fraction of tokens dispatched to expert i

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



Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity
https://www.youtube.com/watch?v=U8J32Z3qV8s&t=2816s

Load Balancing Loss

- N experts; T tokens in a batch $\mathfrak{B}$
- f_i: Fraction of tokens dispatched to expert i

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

- P_i: Expected Probability of selecting expert i

Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity
https://www.youtube.com/watch?v=U8J32Z3qV8s&t=2816s

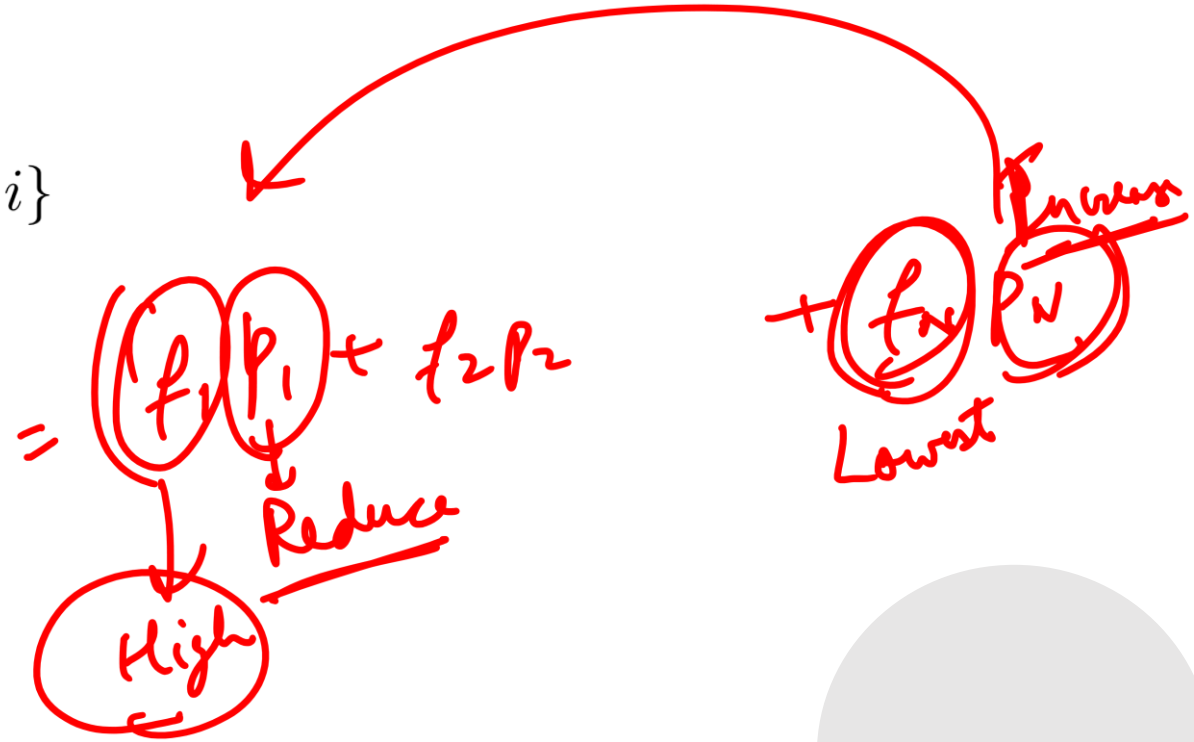
Load Balancing Loss

- N experts; T tokens in a batch $\mathfrak{B}$
- f_i: Fraction of tokens dispatched to expert i

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

- P_i: Expected Probability of selecting expert i

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



Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity
https://www.youtube.com/watch?v=U8J32Z3qV8s&t=2816s

Load Balancing Loss

- N experts; T tokens in a batch $\mathfrak{B}$
- f_i: Fraction of tokens dispatched to expert i

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

- P_i: Expected Probability of selecting expert i

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

Using sample mean as an empirical estimate

Content credits: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity
https://www.youtube.com/watch?v=U8J32Z3qV8s&t=2816s

Load Balancing Loss

- N experts; T tokens in a batch $\mathfrak{B}$
- f_i: Fraction of tokens dispatched to expert i

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

- P_i: Expected Probability of selecting expert i

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

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

Load Balancing Loss

- N experts; T tokens in a batch $\mathfrak{B}$
- f_i: Fraction of tokens dispatched to expert i

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

- P_i: Expected Probability of selecting expert i

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

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

👍 Prevents router collapse

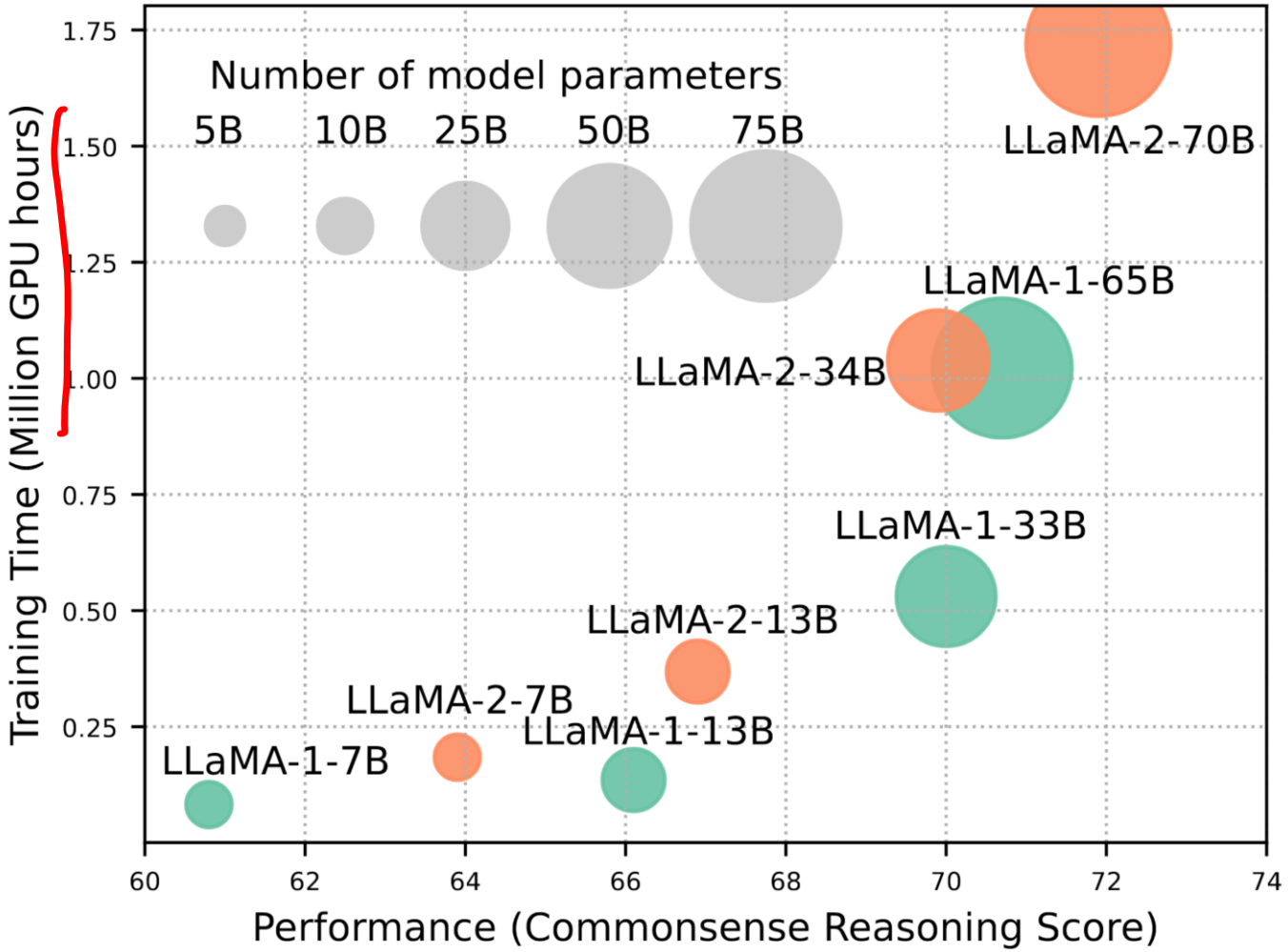
👍 Improves training efficiency by using all the devices equally (remember that each expert is on a separate device)

Homework!

- Read the following papers –
  - Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity
  - Mixtral of Experts
  - MiniMax-01: Scaling Foundation Models with Lightning Attention

Summary

Training Resources vs Performance

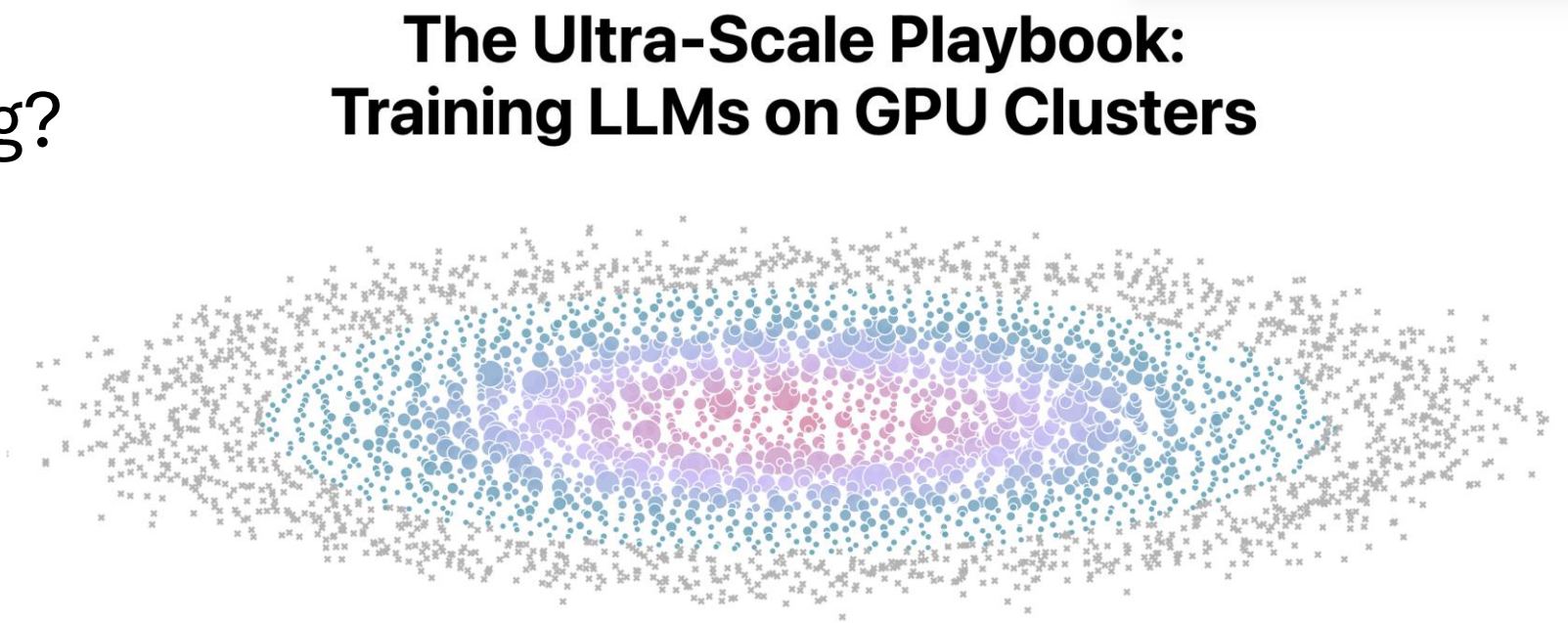


- Based on Nvidia A100 80GB GPU

https://huggingface.co/spaces/optimum/llm-perf-leaderboard

Efficient LLMs

- How to scale training?
  - *Data Parallelism*
  - *Tensor Parallelism*
  - *Context Parallelism*
  - *Pipeline Parallelism*



We ran over 4,000 scaling experiments on up to 512 GPUs and measured throughput (size of markers) and GPU utilization (color of markers). Note that both are normalized per model size in this visualization.

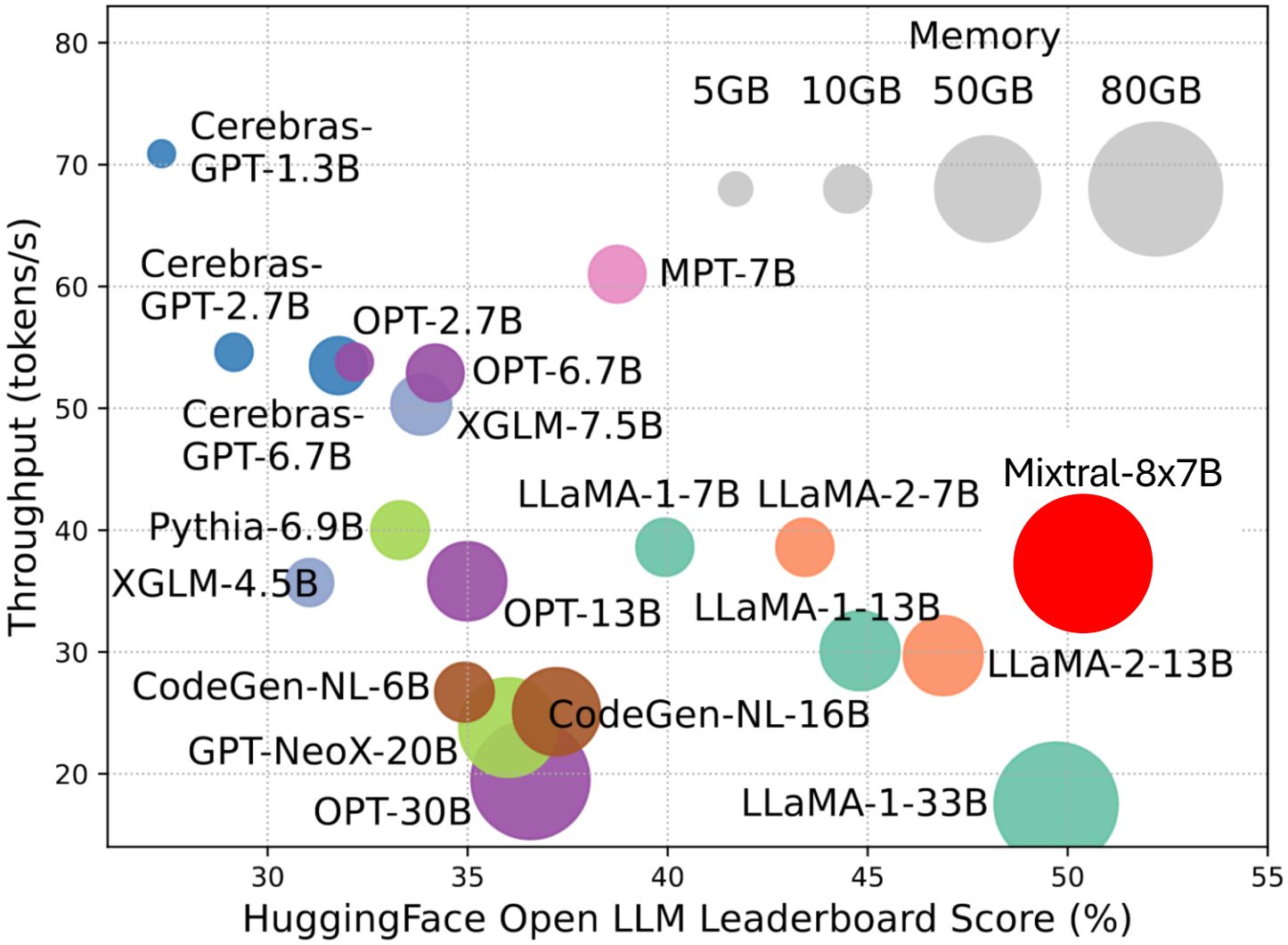
Summary

Efficient Training

1. Parallelism for Scale -
   - *Distribute model & data to fit massive training*
   - *DP, ZeRO-1/2/3, TP (w/ SP), CP, PP*

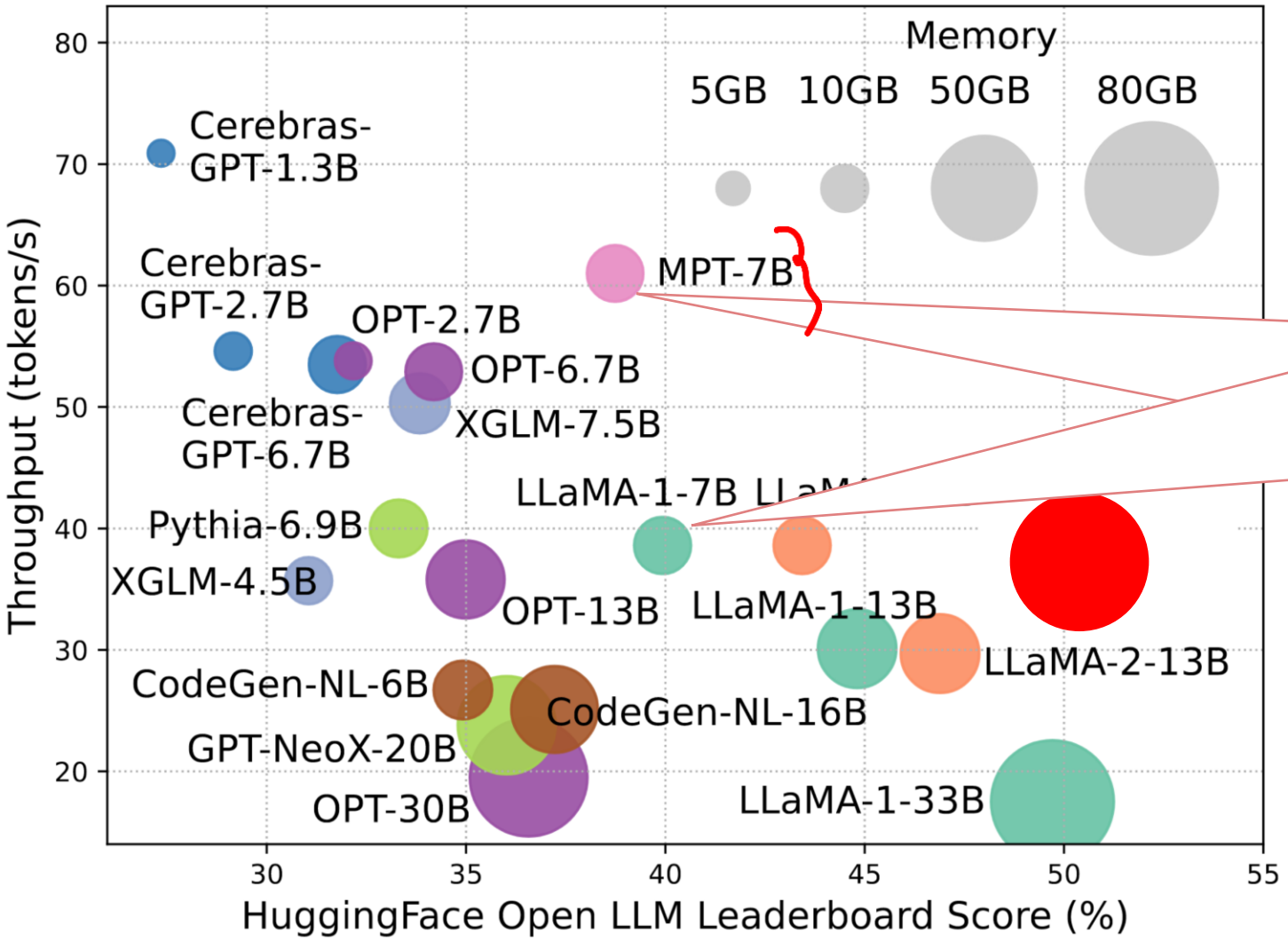
Inference Throughput vs Performance

Inference Throughput vs Performance



- On Nvidia A100 80GB GPU;
- 16-bit quantized
- Batch Size - 1
- Prompt size of 256
- Generating 1000 tokens

Inference Throughput vs Performance



- Similar performance, different throughput! How?
- Efficient implementation –
  - Fused kernel for attention

Summary

Efficient Training

1. Parallelism for Scale -
   - *Distribute model & data to fit massive training*
   - *DP, ZeRO-1/2/3, TP (w/ SP), CP, PP*

2. Efficient Implementations
   - *Flash Attention*
   - *Liger Kernels*

Summary

Efficient Training

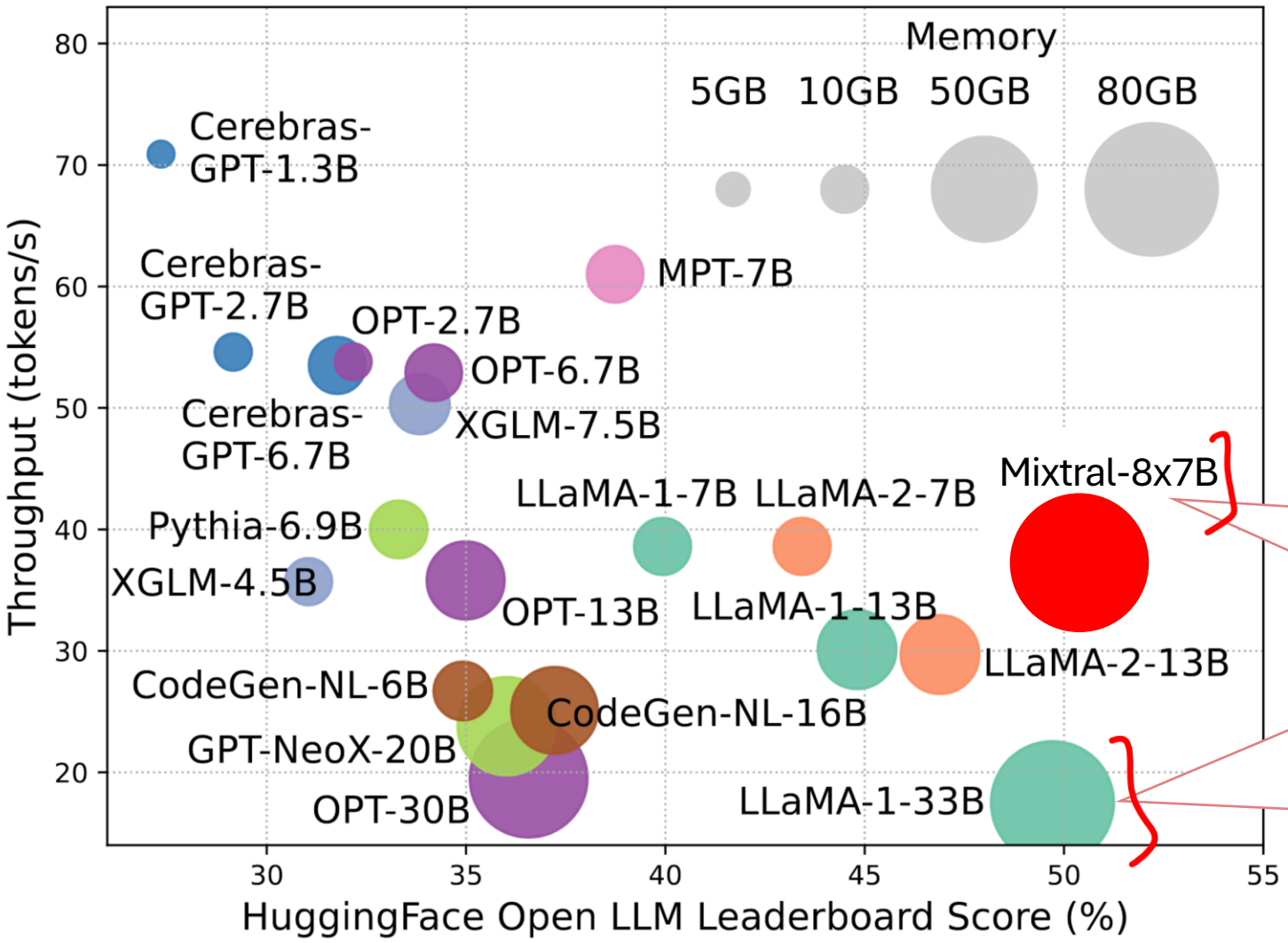
1. Parallelism for Scale -
   - *Distribute model & data to fit massive training*
   - *DP, ZeRO-1/2/3, TP (w/ SP), CP, PP*
2. Efficient Implementations
   - *Flash Attention*
   - *Liger Kernels*

Efficient Inference

1. KV Caching
2. vLLMs and paged attention
3. Flash decoding
4. Speculative decoding

Homework!

Inference Throughput vs Performance



- Similar performance, different throughput! How?
- Efficient design –
  - MoE

Summary

Efficient Training

1. Parallelism for Scale -
   - *Distribute model & data to fit massive training*
   - *DP, ZeRO-1/2/3, TP (w/ SP), CP, PP*
2. Efficient Implementations
   - *Flash Attention*
   - *Liger Kernels*

Efficient Inference

1. KV Caching
2. vLLMs and paged attention
3. Flash decoding
4. Speculative decoding

Homework!

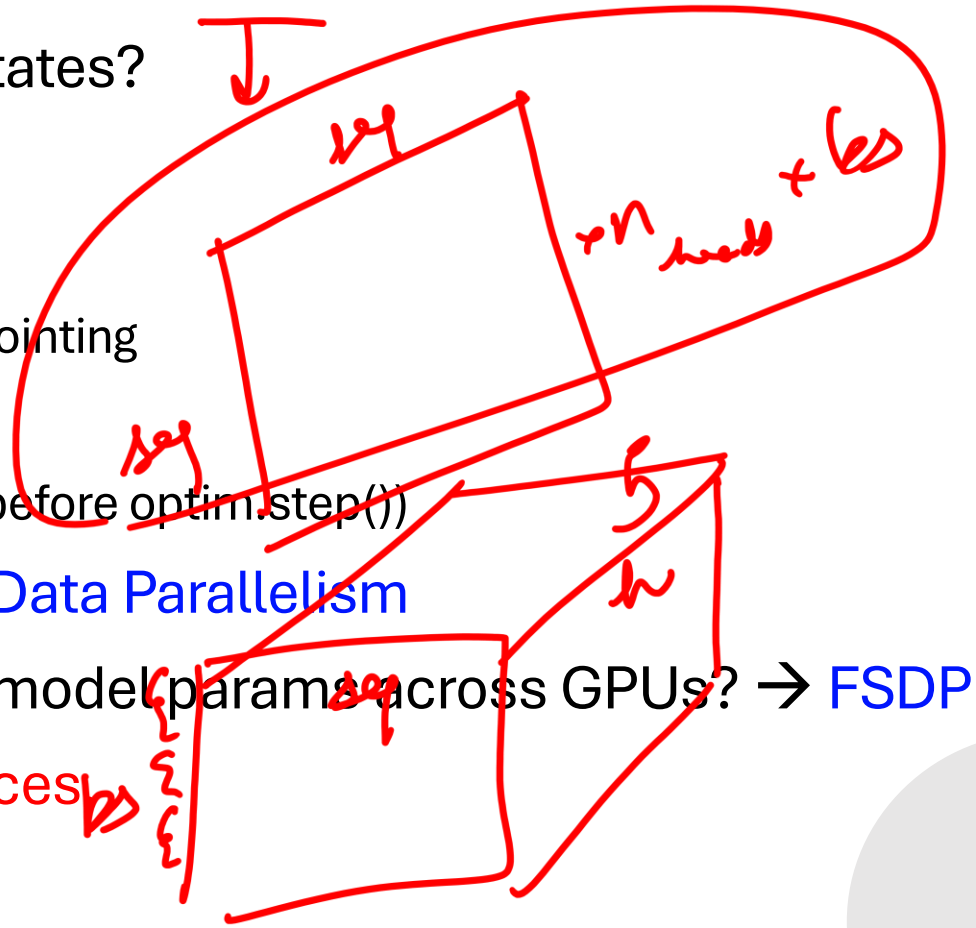
Efficient Design

1. MoE as a Layer
   1. Sparsification
   2. Expert Parallelism
   3. Load Balancing Loss
2. MoE Models
   1. Switch Transformers
   2. Mixtral of Experts
   3. Minimax

Homework!

Recap

- How to train big models on big data?
- What's in the GPU memory during training?
  i. Model weights ; ii. Param gradients iii. Optim states iv. Activations
- What is the size of params / grads / optim states?
- What is the size of activations?
- How to reduce activation memory?
  - Activation re-computation *aka* Gradient checkpointing
- How to increase batch size?
  - Gradient accumulation (run fwd / bwd *k* times before optim.step())
- Can we parallelize grad. Accumulation? → Data Parallelism
- Can we shard the optim. states, grads, and model params across GPUs? → FSDP
- Still not good for big models & large sequences



Recap

- Can we split activations of one input across GPUs?
- Split along the hidden dimension -- let's exploit the distributive property of matrix multiplication! → Tensor Parallelism
- TP: both model weights & activations are split across GPUs!
- TP: LayerNorm & Dropout – same computation on same data - wastage of resources :-(
- Combine TP with Sequence Parallel – *reduce & scatter* along seq. length dimension
- How to handle very very long sequences?
- Context Parallel → split sequence into chunks & process each chunk on a different GPU (same weights but different activations on each GPU)
- How to apply attention on a sequence split on multiple GPUs? → Ring Attention
- TP: Communication overhead beyond a node is prohibitive.
- How to handle very large models? → Pipeline Parallelism