Lecture 25: Mixture of Experts & Efficiency

Scaling Efficiently with Sparse Models 🚀

PSYC 51.07: Models of Language and Communication

Week 9

Today's Journey 🗺️

What we'll cover

1. **The Scaling Problem**: Why bigger isn't always better
2. **Mixture of Experts**: Sparse activation for efficiency
3. **How MoE Works**: Routing, load balancing, training
4. **Real-World MoE**: Mixtral and production systems
5. **Other Efficiency Techniques**: Quantization, pruning, distillation

The Scaling Dilemma 📈

Larger models perform better... but at what cost?

Training a 175B parameter model (GPT-3 scale):

- 💰 Cost: $4-12 million in compute
- 🔥 Energy: Equivalent to 120 homes for a year
- ⏳ Time: Weeks to months on thousands of GPUs
- 🐌 Inference: Slow and expensive ($0.002-0.02 per 1K tokens)
- 🌍 Environmental: Massive carbon footprint

The Challenge

Can we get the benefits of scale without the full computational cost?

Key Insight: Not all parameters need to be active for every input!

Dense vs Sparse Models 🎯

Dense Models (e.g., GPT-3):

```
1Input Token
2    ↓
3[████████████████████]  ← All neurons active
4[████████████████████]  ← 100% of parameters used
5[████████████████████]  ← Every forward pass
6    ↓
7Output
```

- 175B parameters, 175B active
- High compute per token
- Simple architecture

Sparse Models (MoE):

What is Mixture of Experts? 🧠

Mixture of Experts (MoE)
A neural network architecture where different "expert" sub-networks specialize in different parts of the input space, with a gating mechanism that routes inputs to the appropriate experts.

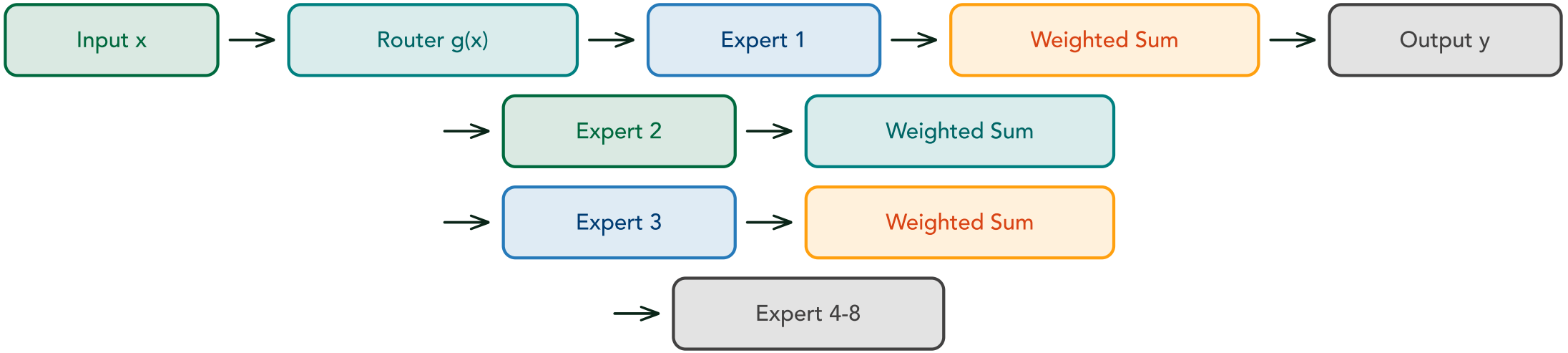
Key Components:

1. **Experts**: Multiple specialized feed-forward networks
2. **Router/Gate**: Learned function that assigns inputs to experts
3. **Sparse Activation**: Only top-k experts process each input

Intuition:

- Different experts become good at different things
- Math expert, code expert, language expert, etc.

MoE Architecture 🏗️



Concrete Example: Processing "def factorial(n):"

```
1# Router computes scores for each expert
2router_logits = router(token_embedding)  # Shape: (8,) for 8 experts
3# [0.1, 0.9, 0.3, 0.2, 0.1, 0.1, 0.1, 0.1]  # Expert 2 (Code) scores highest!
4
5# Select top-2 experts
6top_k_indices = [1, 2]   # Expert 2 (Code) and Expert 3 (Language)
```

The Router Mechanism 🎯

Step-by-step routing for a single token:

```
import torch.nn.functional as F

def route_token(x, router_weights, num_experts=8, k=2):
    """Route a token to top-k experts"""
    # Step 1: Compute routing scores (linear projection)
    # router_weights: (hidden_dim, num_experts)
    logits = x @ router_weights  # Shape: (num_experts,)
    # Example: [-0.5, 2.1, 1.3, 0.2, -0.1, 0.0, -0.3, 0.1]

     # Step 2: Convert to probabilities
    probs = F.softmax(logits, dim=-1)
    # Example: [0.04, 0.52, 0.24, 0.08, 0.03, 0.03, 0.03, 0.03]

    # Step 3: Select top-k experts
    top_k_probs, top_k_indices = torch.topk(probs, k)
    # indices: [1, 2], probs: [0.52, 0.24]
```

The Router Mechanism 🎯

```
21
22    return top_k_indices, top_k_probs
23    # Expert 1 handles 68%, Expert 2 handles 32%
```

...continued

MoE in PyTorch (Simplified) 💻

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MoELayer(nn.Module):
    def __init__(self, d_model, num_experts, expert_capacity, k=2):
        super().__init__()
        self.num_experts = num_experts
        self.k = k  # Top-k routing

        # Router
        self.gate = nn.Linear(d_model, num_experts)

        # Experts (simple FFN for each)
        self.experts = nn.ModuleList([
            nn.Sequential(
                nn.Linear(d_model, 4 * d_model),
                nn.ReLU(),
```

MoE in PyTorch (Simplified) 💻

```
            for _ in range(num_experts)
        ])

    def forward(self, x):
        # x: (batch_size, seq_len, d_model)
        batch_size, seq_len, d_model = x.shape

        # Compute routing scores
        router_logits = self.gate(x)  # (batch, seq, num_experts)
        router_probs = F.softmax(router_logits, dim=-1)

        # Select top-k experts
        top_k_probs, top_k_indices = torch.topk(router_probs, self.k, dim=-1)
        # top_k_probs: (batch, seq, k)
        # top_k_indices: (batch, seq, k)
```

...continued

MoE Forward Pass (cont.) 💻

```
# Initialize output
        output = torch.zeros_like(x)

        # Route to experts
        for i in range(self.k):
            # Get expert indices for this position
            expert_idx = top_k_indices[:, :, i]  # (batch, seq)
            expert_weight = top_k_probs[:, :, i]  # (batch, seq)

             # Process through each expert
             for expert_id in range(self.num_experts):
                 # Mask for tokens routed to this expert
                 mask = (expert_idx == expert_id)

                 if mask.any():
                     # Get tokens for this expert
                     expert_input = x[mask]

```

MoE Forward Pass (cont.) 💻

```
21
22                      # Add weighted output
23                      output[mask] += expert_weight[mask].unsqueeze(-1) * expert
24
25          return output
```

...continued

Note: This is simplified. Production implementations handle batching and load balancing more efficiently.

Load Balancing Problem ⚖️

Problem: Without balancing, some experts get overused!

```
1Expert Usage Distribution (Unbalanced):
2
3E1: ████████████████████████████████ 40%  ← Overloaded!
4E2: ██████████████████               22%
5E3: ████████████                     15%
6E4: ████████                         10%
7E5: ████                              5%
8E6: ███                               4%
9E7: ██                                3%  ← Undertrained
10E8: ▌                                 1%  ← "Dead" expert
```

Desired (Balanced):

```
1E1: ████████████  12.5%
```

Load Balancing Solutions 🔧

Solution 1: Auxiliary Loss (Penalize Imbalance)

```
def compute_load_balance_loss(router_probs, expert_assignments, alpha=0.01):
    """Add penalty to main loss for imbalanced routing"""
    num_experts = router_probs.shape[-1]

    # f_i: fraction of tokens actually sent to expert i
    tokens_per_expert = expert_assignments.sum(dim=0)  # Count per expert
    f = tokens_per_expert / tokens_per_expert.sum()    # [0.4, 0.22, ...]

    # P_i: average routing probability for expert i
     P = router_probs.mean(dim=0)                       # [0.35, 0.2, ...]

     # Loss: encourages f and P to both be uniform (1/N each)
     aux_loss = alpha * num_experts * (f * P).sum()
     return aux_loss  # Added to main training loss
```

Solution 2: Expert Capacity Limits

Training Instability ⚠️

Challenges in training MoE:

1. **Routing Collapse**
    - All tokens routed to one or few experts
- *Solution:* Load balancing loss, initialization
2. **Expert Imbalance**
    - Some experts undertrained
- *Solution:* Balanced batching, capacity limits
3. **High Variance Gradients**
    - Discrete routing decisions
- *Solution:* Softmax gating, larger batch sizes

Memory and Communication 💾

MoE Memory Requirements:

Challenge

All experts must be in memory, even if only 2 are active!

Example: Mixtral 8x7B

- 8 experts × 7B parameters each = 56B total
- But only 2 experts active → 14B active parameters
- **Memory:** Need to store all 56B parameters
- **Compute:** Only process 14B parameters per token

Solutions:

1. **Expert Parallelism**: Distribute experts across GPUs

Mixtral 8x7B 🎬

Mixtral (Jiang et al., 2024)

A state-of-the-art sparse MoE model from Mistral AI with 8 experts, each 7B parameters.

Architecture:

- **Total parameters**: 47B (8 experts × 7B, minus shared layers)
- **Active parameters**: 13B (only 2 experts active per token)
- **Layers**: 32 transformer blocks
- **Context**: 32K tokens
- **Vocabulary**: 32K tokens (SentencePiece)

Training:

- Open weights (Apache 2.0 license)

Mixtral Performance 📊

Comparison with dense models:

Model	Total Params	Active Params	MMLU Score	Speed vs 70B
Llama 2 13B	13B	13B	55.0	5× faster
Mixtral 8x7B	**47B**	**13B**	**70.6**	**5× faster**
Llama 2 70B	70B	70B	69.7	1× (baseline)
GPT-3.5	~175B	~175B	70.0	N/A (API)

The Magic of MoE:

```
Mixtral achieves:
┌──────────────────────────────────────────────────────┐
  Quality of 70B model (MMLU: 70.6 vs 69.7)
```

What Do Experts Learn? 🔬

Experts naturally specialize without explicit supervision!

Analyzed routing patterns in Mixtral:

```
1Token Type                    → Most Active Experts
2──────────────────────────────────────────────────
3"def", "class", "import" → Expert 2 (Code)
4"la", "le", "français"   → Expert 5 (French)
5"∑", "∫", "theorem"      → Expert 7 (Math)
6"the", "is", "and"       → Expert 1 (Common words)
7"neural", "gradient"     → Expert 3 (Technical)
```

Concrete Example: Sentence routing

```
1"The neural network learns via backpropagation"
2 |    |      |       |     |    |
```

Model Compression Methods 🗜️

Beyond MoE: Making models smaller and faster

1. Quantization

```
1# Original: 32-bit float (4 bytes/param)
2weight = 0.123456789  # Full precision
3
4# INT8: 8-bit integer (1 byte/param)
5weight_int8 = 31  # Scaled + quantized
6# 4× memory reduction!
7
8# INT4: 4-bit (0.5 byte/param)
9# 8× memory reduction!
```

Llama 2 7B Memory:

- FP32: 28 GB

Inference Optimizations ⚡

Making generation faster:

1. KV Cache (Essential)

```
1# Without cache: Recompute all attention
2# Token 100 attends to tokens 1-99
3# = O(n²) attention per token!
4
5# With cache: Store previous K,V
6cache = {}
7for token in sequence:
8    k, v = compute_kv(token)
9    cache[pos] = (k, v)  # Store!
10    # Only compute attention once
```

2. Flash Attention

State Space Models: Mamba

Alternative to Transformers with linear-time complexity

Transformer Attention: O(n²)

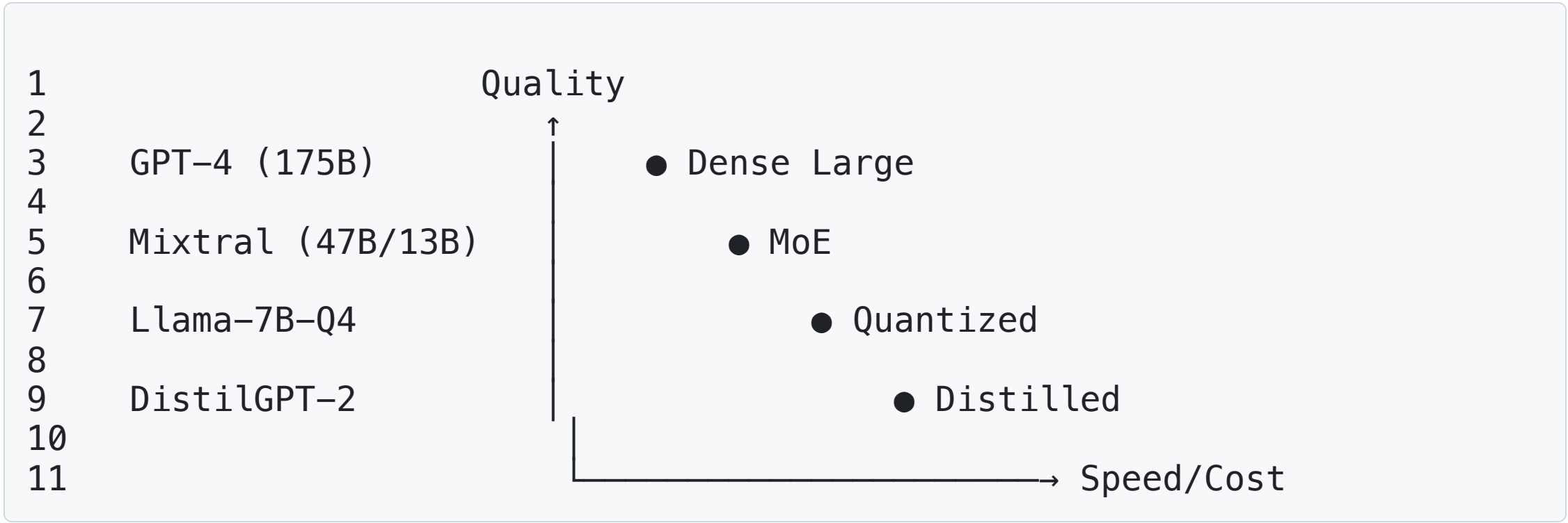
```
1Sequence length:   1K   4K   16K   64K
2Compute (relative): 1   16   256   4096
3                     ↑
4             Gets expensive fast!
```

Mamba SSM: O(n)

```
1Sequence length:   1K   4K   16K   64K
2Compute (relative): 1    4    16    64
3                          ↑
4             Linear scaling!
```

Efficiency Trade-off Landscape 🗺️

Choosing the Right Technique for Your Use Case:



Technique	Best For	Trade-off
Dense Large	Maximum quality	Expensive, slow

Environmental Impact 🌍

The carbon cost of large language models:

GPT-3	**502**	**112 cars for 1 year**
Llama 2	~539	120 cars for 1 year

Factors affecting carbon footprint:

- Model size (more parameters = more compute)
- Training time
- Hardware efficiency (newer GPUs more efficient)
- Energy source (coal vs solar)
- Location of data center

Source: Strubell et al. (2019) - "Energy and Policy Considerations for Deep Learning in

Democratizing Access 🔓

Should powerful AI be accessible to everyone, or only large organizations?

Current reality:

- Training GPT-3 scale model: $4-12M
- Requires 1000+ GPUs
- Only big tech companies can afford
- Creates AI "haves" and "have-nots"

Efficiency enables democratization:

- ✅ Smaller models on consumer hardware
- ✅ Open-source weights (Llama, Mixtral, Mistral)
- ✅ Efficient fine-tuning (LoRA, QLoRA)

Open vs Closed Models 🤔

Closed (GPT-4, Claude):

- ✅ Better safety control
- ✅ Monetization easier
- ✅ Protect IP
- ✅ Can update/improve
- ❌ No transparency
- ❌ Vendor lock-in
- ❌ Limited customization
- ❌ Privacy concerns

Open (Llama, Mixtral):

Future of Efficient LLMs 🔮

Where is the field heading?

1. **Hybrid Architectures**
    - Combine MoE + attention + SSMs
- Use different mechanisms for different parts
- Dynamic architecture selection

2. **Conditional Computation**
    - Adjust depth/width based on input difficulty
- Early exit for easy examples
- More compute for hard problems

3. **Neural Architecture Search**
    - Automatically find efficient architectures

Key Takeaways 🔑

1. **MoE enables efficient scaling**
    - More parameters, fewer active per token
- Better quality/compute ratio
2. **Challenges exist but solvable**
    - Load balancing, training instability
- Solutions: auxiliary losses, capacity limits
3. **Mixtral proves MoE works at scale**
    - 70B-quality with 13B-cost
- Open weights accelerate adoption
4. **Many paths to efficiency**
    - Quantization, pruning, distillation

Readings 📖

Required:

1. **Shazeer et al. (2017)**: Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer
   [arXiv]
2. **Jiang et al. (2024)**: Mixtral of Experts
   [arXiv]

Recommended:

- Fedus et al. (2022): Switch Transformers [arXiv]
- Gu & Dao (2023): Mamba [arXiv]
- Dao et al. (2022): FlashAttention [arXiv]
- Strubell et al. (2019): Energy and Policy Considerations [arXiv]

Questions? 💬

Next: Ethics, Bias, and Safety!