A Deep Dive into Mixture of Experts (MoE) for Large Language Models

> This guide provides a comprehensive overview of the Mixture of Experts (MoE) architecture, a key technique for scaling neural networks to hundreds of billions or even trillions of parameters. We will deconstruct the seminal paper "Outrageously Large Neural Networks," explore the core concepts of conditional computation, sparse gating, and load balancing, and connect them to modern implementations in state-of-the-art models like Mixtral. The guide concludes with a curated set of theoretical and practical interview questions, complete with detailed answers and Python/PyTorch code implementations, to prepare you for rigorous technical interviews.

Knowledge Section

1. Introduction: The Scaling Challenge and Conditional Computation

The success of deep learning has been fueled by scaling: more data and larger models consistently yield better performance. However, traditional dense models face a critical bottleneck. In a dense architecture, every parameter is activated for every single input token. This means that as the model size (number of parameters) and training data grow, the computational cost increases quadratically, quickly becoming infeasible.

Conditional Computation offers a powerful solution. Instead of activating the entire network for every input, conditional computation selectively activates only certain parts of the model. This allows for a massive increase in the model's total number of parameters (its "capacity" to store knowledge) without a proportional increase in the computational cost (FLOPs) required for processing each input. The Sparsely-Gated Mixture of Experts layer is a highly successful implementation of this principle.

2. The Sparsely-Gated Mixture of Experts (MoE) Layer

An MoE layer replaces a standard component of a neural network (like a feed-forward block) with a more sophisticated structure. Imagine having a team of specialized "expert" networks.

For any given input, instead of consulting every expert, you have a "gating network" or "router" that intelligently selects a small, relevant subset of experts to process the input. The final output is then a weighted combination of the outputs from these selected experts.

The MoE layer, as proposed in the 2017 paper, consists of two main components:

1. **Expert Networks (E_i):** A set of n simpler neural networks, typically feed-forward networks (FFNs). In the original paper, they were placed between LSTM layers, but in modern Transformers, they replace the FFN block. Each expert has its own distinct set of parameters.
2. **Gating Network (G):** A trainable routing network that determines which experts to activate for a given input. It takes the input token's representation and outputs a probability distribution over the experts.

2.1. Mathematical Formulation

Given an input vector x, the output of an MoE layer, y, is a weighted sum of the expert outputs. The weights are provided by the gating network G.

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

Where:

- n is the total number of experts.
- $E_i(x)$ is the output of the i-th expert network.
- $G(x)_i$ is the weight (gate value) assigned by the gating network to the i-th expert.

Crucially, for this to be computationally efficient, the gating vector $G(x)$ must be **sparse**. This means most of its values, $G(x)_i$, are zero. If $G(x)_i = 0$, there is no need to compute the output of the corresponding expert $E_i(x)$, saving significant computation.

2.2. The Noisy Top-K Gating Mechanism

A simple softmax gate would produce dense weights, requiring every expert to be computed. To enforce sparsity, the authors introduced **Noisy Top-K Gating**.

The process is as follows:

1. **Compute pre-gate logits:** The input x is passed through a linear layer to produce logits for each expert.

$$h(x) = x \cdot W_g$$

2. **Add tunable noise:** To improve load balancing and encourage exploration during training, a controllable amount of Gaussian noise is added to the logits.

$$H(x)_i = h(x)_i + \text{StandardNormal}() \cdot \text{Softplus}((x \cdot W_{\text{noise}})_i)$$

   The `Softplus` function ensures the noise scale is always positive. This noise injection is a form of exploration, preventing the gating network from collapsing into a state where it only ever picks a few popular experts.
3. **Select Top-K Experts:** Only the logits with the top k values are kept; the rest are set to $-\infty$. This is the hard sparsity-inducing step.

$$\text{KeepTopK}(v, k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v \\ -\infty & \text{otherwise.} \end{cases}$$

4. **Apply Softmax:** A standard softmax function is applied to the resulting sparse vector of logits. This converts the logits into a valid probability distribution over the selected k experts. The experts whose logits were set to $-\infty$ will have a final gate value of 0.

$$G(x) = \text{Softmax}(\text{KeepTopK}(H(x), k))$$

   Typically, k is a small number like 1, 2, or 4, even when the total number of experts n is in the thousands. This ensures that only a tiny fraction of the model's parameters are used for any given token.

3. Training Challenges and Solutions

Implementing MoE layers introduces unique challenges not present in dense models.

3.1. The Load Balancing Problem

A major issue is that the gating network is a feedback loop. It learns to send tokens to the

experts that give the best results. This can lead to a "rich-get-richer" dynamic where a few experts are heavily favored and trained, while others are neglected and remain underdeveloped. This imbalance negates the benefit of having a large number of diverse experts.

To counteract this, an **auxiliary loss function** is added to the main model loss. This loss encourages the gating network to distribute the load more evenly across all experts. The total loss becomes:

$$L = L_{\text{task}} + w_{\text{aux}} \cdot L_{\text{aux}}$$

The auxiliary loss, L_{aux}, is designed to equalize the importance of each expert over a training batch. The "importance" of an expert is the sum of the gate values it receives across all tokens in a batch.

Let B be the set of tokens in a training batch. The importance of expert i is:

$$\text{Importance}_i = \sum_{x \in B} G(x)_i$$

The auxiliary loss is then defined as the squared coefficient of variation (CV) of these importance values across all experts:

$$L_{\text{aux}} = (\text{CV}(\text{Importance}))^2 = \left(\frac{\text{std}(\text{Importance})}{\text{mean}(\text{Importance})} \right)^2$$

Minimizing this loss pushes the standard deviation of expert importance towards zero, meaning all experts receive a similar amount of training signal over a batch. w_{aux} is a tunable hyperparameter that balances the primary task loss with this load balancing objective.

3.2. The Shrinking Batch Problem

Conditional computation leads to a practical implementation challenge. Imagine a batch of 512 tokens, a total of 64 experts ($n = 64$), and we select the top 2 experts for each token ($k = 2$). Each expert will, on average, only receive $\frac{512 \times 2}{64} = 16$ tokens. This drastically reduced, or "shrunken," batch size per expert is inefficient for modern hardware like GPUs, which are optimized for large, parallel matrix multiplications.

The solution involves a combination of data and model parallelism:

- **Model Parallelism:** The experts themselves are distributed across multiple devices (e.g., GPUs). For instance, if you have 8 GPUs, each GPU can hold 8 out of the 64 experts.
- **Data Parallelism:** The input data batch is present on every device. Each device processes the full batch of tokens through its local experts. All-to-all communication is then required to send each token's representation to the device holding its assigned expert and to gather the results. This approach ensures that each expert still processes a large number of tokens aggregated from the batches on all devices, thus solving the shrinking batch problem.

4. MoE in Modern LLMs (e.g., Mixtral 8x7B)

While the original paper applied MoE to LSTMs, its most impactful application today is within the **Transformer architecture**. In a standard Transformer block, the MoE layer replaces the feed-forward network (FFN) sub-layer.

Mixtral 8x7B: A Case Study (as of late 2023)

Mistral AI's Mixtral 8x7B is a prominent example of a high-performing open-source MoE model. Its architecture highlights modern MoE design:

- **Total vs. Active Parameters:** The model has 8 experts. The "8x7B" naming suggests that each expert is around 7 billion parameters. However, many parameters (like the attention layers) are shared across experts. The total parameter count is closer to 47B, not 56B.
- **Sparse Activation:** For each token at each MoE layer, the router selects the **top 2** experts. This means that during inference, only about 13B parameters are active per token (the shared parameters plus the parameters of two 7B-class experts). This is why it has the speed and cost of a 13B dense model but the knowledge capacity of a much larger 47B model.
- **Placement:** The MoE layers are used in place of the FFN layers in the Transformer blocks. This is done for every other layer in the model's architecture.

The structure of a Transformer block with an MoE layer looks like this:

1. Input `x`
2. `x = x + MultiHeadSelfAttention(LayerNorm(x))`

3. `x = x + MoELayer(LayerNorm(x))` *(Replaces the FFN)*
4. Output `x`

5. Pros and Cons of Mixture of Experts

Advantages

- **Scalable Capacity:** Allows for a massive increase in the number of model parameters without a proportional increase in inference cost.
- **Fast Inference:** For a given parameter count, MoE models are significantly faster at inference than dense models because they only use a fraction of their parameters per token.
- **Specialization:** Experts can learn to specialize in different aspects of the data (e.g., different languages, topics, or syntactic structures), potentially leading to better performance.

Disadvantages

- **High VRAM Requirement:** During training and inference, all parameters (all experts) must be loaded into memory (VRAM), even though only a fraction are used at a time. A model like Mixtral 8x7B requires significant VRAM to load all ~47B parameters.
- **Training Instability:** Training MoE models can be more challenging. The auxiliary loss is critical, and tuning its weight (w_{aux}) can be difficult. Poor tuning can lead to load balancing issues or convergence problems.
- **Implementation Complexity:** The routing logic, communication overhead (all-to-all), and parallel strategies make MoE models more complex to implement and optimize than dense models.

Interview Questions

Theoretical Questions

Question 1: What is Mixture of Experts (MoE) and how does it fundamentally differ from a standard dense model?

Answer:
A Mixture of Experts (MoE) is a neural network architecture based on the principle of **conditional computation**. It consists of a set of "expert" sub-networks and a "gating network" or "router." For each input, the gating network selects a small subset of experts to process it, and their outputs are combined to produce the final result.

The fundamental difference from a dense model lies in parameter activation:

- **Dense Model:** For every input token, **all** parameters of the model are activated and used in the computation. The computational cost (FLOPs) is directly proportional to the total number of parameters.
- **MoE Model:** For every input token, only a **small fraction** of the parameters are activated—specifically, the parameters of the shared components (like attention layers) and the few experts selected by the gating network. This decouples the total parameter count from the computational cost per token, allowing for models with extremely high parameter counts (knowledge capacity) but relatively low inference cost.

In essence, dense models use all their knowledge for every problem, while MoE models learn to consult only the most relevant specialists for each specific problem.

Question 2: Explain the "load balancing" problem in MoE and derive the auxiliary loss function used to address it.

Answer:
The load balancing problem arises because the gating network, which is trained alongside the experts, can develop a bias towards a few "favorite" experts. This creates a self-reinforcing loop: certain experts receive more training data, become better, and are thus chosen even more frequently by the gate. Other experts are starved of data, remain undertrained, and are rarely selected. This imbalance undermines the very purpose of having a large, diverse set of

experts.

To solve this, an **auxiliary loss function** is added to the main task loss during training. This loss encourages the gating network to distribute tokens (the "load") evenly across all experts.

Derivation and Explanation:
The goal is to ensure that over a batch of data, each expert receives a roughly equal amount of attention from the gating network.

1. **Define Expert Importance:** First, we define the "importance" of an expert i over a batch of tokens B as the sum of the gate values it receives for all tokens in that batch.

$$\text{Importance}_i(B) = \sum_{x \in B} G(x)_i$$

   Here, $G(x)_i$ is the gate's output weight for expert i on input x.
2. **Goal: Equal Importance:** If the load is perfectly balanced, the importance value for every expert would be the same. This means the variance (or standard deviation) of the set of importance values `[Importance_1, Importance_2, ..., Importance_n]` would be zero.
3. **Formulate the Loss:** A common way to penalize variance is to use the **Coefficient of Variation (CV)**, which is the standard deviation divided by the mean. The auxiliary loss is typically defined as the square of the CV of the experts' importance values.

$$L_{\text{aux}} = w_{\text{aux}} \cdot (\text{CV}(\text{Importance}))^2 = w_{\text{aux}} \cdot \left(\frac{\sqrt{\text{Var}(\text{Importance})}}{\text{Mean}(\text{Importance})} \right)^2$$

   where w_{aux} is a tunable hyperparameter. Adding this term to the main loss function penalizes the model whenever the distribution of work across experts becomes uneven, pushing the training dynamics toward a more balanced state.

Question 3: How is an MoE layer typically integrated into a modern Transformer architecture like GPT or Mixtral? Which component does it replace?

Answer:
In modern Transformer architectures, the MoE layer is used as a direct, drop-in replacement for the **Feed-Forward Network (FFN)** sub-layer within a Transformer block.

A standard Transformer block consists of two main sub-layers:

1. A Multi-Head Self-Attention (MHSA) layer.
2. A position-wise Feed-Forward Network (FFN), which is typically a two-layer MLP.

The data flow in a standard block is: `Attention -> Add & Norm -> FFN -> Add & Norm`.

To integrate MoE, the FFN is replaced with the MoE layer. The MoE layer itself contains multiple FFNs (the experts) and a gating network to route to them. The data flow becomes:

Transformer Block with MoE:

1. Input `x`
2. `attention_output = MultiHeadSelfAttention(LayerNorm(x))`
3. `x = x + attention_output`
4. `moe_output = MoELayer(LayerNorm(x))` **(This is the replacement)**
5. `x = x + moe_output`
6. Output `x`

So, the MoE layer takes the place of the dense FFN. This is a powerful strategic choice because the FFN component typically accounts for about two-thirds of a Transformer's parameters. Replacing it with a sparse MoE layer allows for a massive increase in parameters in that part of the model while keeping the per-token computation low. In models like Mixtral, this replacement is done in alternating layers to balance model capacity and training stability.

Question 4: Explain the "shrinking batch problem" in MoE and discuss the technical solutions used to mitigate it.

Answer:
The "shrinking batch problem" is a critical performance issue in naive MoE implementations. Modern hardware like GPUs achieves high throughput by performing operations on large

tensors (batches) in parallel. In an MoE model, the gating network splits the input batch of tokens among many experts.

For example, if you have a batch of 1024 tokens, 128 experts, and you route each token to 2 experts ($k = 2$), each expert, on average, will only process $\frac{1024 \times 2}{128} = 16$ tokens. This tiny "batch" per expert is extremely inefficient for a GPU, leading to severe under-utilization of the hardware and slow training times.

Solutions:
The primary solution involves a clever combination of **model parallelism** and **data parallelism**, often orchestrated with an **all-to-all communication** primitive.

1. **Model Parallelism:** The experts are distributed across multiple processing units (e.g., GPUs). If you have 8 GPUs and 128 experts, each GPU can hold $128/8 = 16$ experts. All parameters for these 16 experts reside on their assigned GPU.
2. **Data Parallelism:** The full batch of input tokens is replicated on *each* GPU.
3. **Computation and Communication Flow:**
    - **Step 1 (Gating):** Each GPU computes the gating decisions for the *entire* batch of 1024 tokens. Now, each GPU knows which two experts each token needs to be sent to.
    - **Step 2 (All-to-All Communication):** An `all-to-all` communication operation is performed. Each GPU partitions its batch of 1024 tokens into 8 chunks, one for each destination GPU. GPU `i` sends the chunk of tokens destined for experts on GPU `j` to GPU `j`. After this step, each GPU has a new batch of tokens, where all tokens in this new batch are designated for the experts residing locally on that GPU. This re-shuffled batch is now large enough for efficient computation.
    - **Step 3 (Expert Computation):** Each GPU processes its re-shuffled batch of tokens through its local experts.
    - **Step 4 (Reverse All-to-All):** A second `all-to-all` operation is performed to send the results from the experts back to the original GPUs, so that each token's representation is back on the GPU where it started.

This sophisticated strategy ensures that each expert processes a large batch of tokens aggregated from all devices, solving the shrinking batch problem and allowing MoE models to

train efficiently at scale.

Practical & Coding Questions

Question 5: Implement a basic MoE layer from scratch in PyTorch. Your implementation should include a gating network and a list of experts, and show how an input batch is processed.

Answer:

Here is a complete, commented PyTorch implementation of a basic MoE layer.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt

# A simple Feed-Forward Network to be used as an expert
class Expert(nn.Module):
    def __init__(self, d_model, d_hidden):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_hidden),
            nn.ReLU(),
            nn.Linear(d_hidden, d_model)
        )

    def forward(self, x):
        return self.net(x)

# The main MoE Layer
class MoELayer(nn.Module):
    """
    A basic Mixture of Experts layer.

    Args:
        d_model (int): The hidden dimension of the input and output.
        num_experts (int): The total number of experts.
        top_k (int): The number of experts to route each token to.
        d_hidden (int): The hidden dimension of each expert's FFN.
    """
    def __init__(self, d_model, num_experts, top_k, d_hidden):
        super().__init__()
        self.d_model = d_model
        self.num_experts = num_experts
        self.top_k = top_k

        # Create a list of expert networks
        self.experts = nn.ModuleList([Expert(d_model, d_hidden) for _ in range(num_experts
```

```
        # The gating network (router)
        self.gate = nn.Linear(d_model, num_experts)

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

        # Reshape for gating: (batch_size * seq_len, d_model)
        x_reshaped = x.reshape(-1, d_model)

        # 1. Gating: Get logits for each expert
        # gate_logits shape: (batch_size * seq_len, num_experts)
        gate_logits = self.gate(x_reshaped)

        # 2. Select Top-K experts
        # Find the top-k scores and their indices
        # top_k_weights and top_k_indices shape: (batch_size * seq_len, top_k)
        top_k_weights, top_k_indices = torch.topk(gate_logits, self.top_k, dim=-1)

        # Apply softmax to the top-k scores to get routing weights
        routing_weights = F.softmax(top_k_weights, dim=-1)

        # 3. Route tokens to experts and aggregate results
        # We need to create a final output tensor of the same shape as the input
        final_output = torch.zeros_like(x_reshaped)

        # To make this efficient, we create a flat list of tokens and their expert assignm
        # flat_top_k_indices shape: (batch_size * seq_len * top_k)
        flat_top_k_indices = top_k_indices.flatten()

        # Create a combined batch for processing, mapping each token to its chosen expert
        # This is a simplified way to handle batching for demonstration.
        # In a real system, this is where complex all-to-all communication would happen.

        # Create a sparse binary mask to select tokens for each expert
        # This mask helps us select which tokens go to which expert.
        # It's a boolean tensor of shape (batch_size * seq_len, num_experts)
        expert_mask = F.one_hot(top_k_indices, num_classes=self.num_experts).sum(dim=1)
        expert_mask = expert_mask.bool()
```

```
        # Initialize an empty tensor for the outputs of all experts
        expert_outputs = torch.zeros_like(x_reshaped)

        # Loop through each expert and process the assigned tokens
        for i in range(self.num_experts):
            # Find which tokens are assigned to this expert
            idx, = torch.where(expert_mask[:, i])

            if idx.numel() > 0:
                # Select the tokens for the current expert
                expert_input = x_reshaped[idx]
                # Process through the expert
                expert_output = self.experts[i](expert_input)
                # Store the output
                expert_outputs.index_add_(0, idx, expert_output)

        # 4. Weight the expert outputs by the routing weights
        # We need to expand routing_weights and top_k_indices to gather correctly
        expanded_routing_weights = routing_weights.flatten()

        # Scatter the weighted outputs back into the final output tensor
        # This is a more complex gather/scatter operation in practice
        # For simplicity, we'll iterate
        for i in range(x_reshaped.size(0)): # Iterate over each token
            for j in range(self.top_k):
                expert_idx = top_k_indices[i, j]
                weight = routing_weights[i, j]

                # In our simplified batched approach, expert_outputs contains the sum of o
                # This part is tricky to implement efficiently without custom kernels.
                # Here's a conceptual, though inefficient, way:
                # final_output[i] += weight * self.experts[expert_idx](x_reshaped[i])

        # A more efficient, but complex, way to perform the weighted sum:
        # We can create a dense tensor of expert outputs and then multiply by weights.
        # This is memory-intensive but vectorized.
        # Let's stick to a clearer, if less optimal, approach for this example.
```

```
        # Let's refine the logic to be more PyTorch-idiomatic
        final_output = torch.zeros_like(x_reshaped)
        # Create a flat index for batch items
        flat_indices = torch.arange(x_reshaped.size(0)).repeat_interleave(self.top_k)

        # Gather the expert outputs corresponding to the chosen experts for each token
        # This is complex. A simpler loop is clearer for demonstration.
        temp_outputs = torch.zeros_like(x_reshaped)
        for i in range(x_reshaped.size(0)):
            for j in range(self.top_k):
                expert_idx = top_k_indices[i, j]
                weight = routing_weights[i, j]
                temp_outputs[i] += self.experts[expert_idx](x_reshaped[i]) * weight

        final_output = temp_outputs

        # Reshape back to the original input shape
        return final_output.view(batch_size, seq_len, d_model)


# --- Example Usage ---
d_model = 512
d_hidden = 2048
num_experts = 8
top_k = 2
batch_size = 4
seq_len = 10

# Create a dummy input tensor
input_tensor = torch.randn(batch_size, seq_len, d_model)

# Instantiate the MoE layer
moe_layer = MoELayer(d_model=d_model, num_experts=num_experts, top_k=top_k, d_hidden=d_hi

# Get the output
output = moe_layer(input_tensor)

print("Input shape:", input_tensor.shape)
print("Output shape:", output.shape)
```

```
assert input_tensor.shape == output.shape
print("\nMoE layer executed successfully!")
```

Question 6: Using Python and Matplotlib, create a visualization that explains the effect of the load balancing auxiliary loss. Show how expert utilization can be skewed without it and balanced with it.

Answer:

This code simulates the expert selection process over several training steps. It demonstrates how a load-balancing loss term pushes the gating network to distribute its choices more evenly across all available experts.

```python
import numpy as np
import matplotlib.pyplot as plt

# --- Simulation Parameters ---
num_experts = 16
num_steps = 200  # Number of simulated training steps
batch_size = 1024 # Tokens per step
top_k = 2
learning_rate = 0.1
w_aux = 0.01 # Weight for the auxiliary loss

# Initialize expert logits (representing the gating network's preference)
expert_logits = np.zeros(num_experts)

def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

def simulate_training(use_aux_loss):
    """Simulates the evolution of expert selection over time."""
    logits_history = []
    expert_counts = np.zeros(num_experts)
    current_logits = np.copy(expert_logits)

    for step in range(num_steps):
        # Simulate gating for a batch
        # For simplicity, we assume the gate's choice is driven by its current logits
        gate_probs = softmax(current_logits)

        # Simulate selecting top_k experts for each token in the batch
        # We can model this by sampling from the distribution
        choices = np.random.choice(
            num_experts,
            size=(batch_size * top_k),
            p=gate_probs
        )

        # Count how many times each expert was chosen in this step
        step_counts = np.bincount(choices, minlength=num_experts)
```

```
        expert_counts += step_counts

        # --- GRADIENT UPDATE ---
        # Simulate the gradient from the main task loss.
        # This gradient will favor experts that are chosen (positive gradient for chosen e
        # We simulate this by giving a positive reward to chosen experts.
        grad_task = step_counts / step_counts.sum()

        # Calculate update for logits
        update = grad_task

        if use_aux_loss:
            # --- AUXILIARY LOSS GRADIENT ---
            # The loss encourages counts to be equal. The gradient will push down
            # the logits of over-used experts and push up the logits of under-used experts

            # Simplified gradient of the load balancing loss
            # It's proportional to the deviation from the mean count.
            mean_count = expert_counts.mean()
            grad_aux = expert_counts - mean_count

            # Normalize the aux gradient
            grad_aux /= (np.std(grad_aux) + 1e-6)

            # Combine gradients
            update -= w_aux * grad_aux

        # Update the logits (simple gradient ascent)
        current_logits += learning_rate * update
        logits_history.append(softmax(current_logits))

    return np.array(logits_history)

# --- Run Simulations ---
history_no_loss = simulate_training(use_aux_loss=False)
history_with_loss = simulate_training(use_aux_loss=True)

# --- Plotting ---
fig, axes = plt.subplots(2, 1, figsize=(12, 10), sharex=True)
```

```
fig.suptitle("Effect of Load Balancing Auxiliary Loss on Expert Utilization", fontsize=16

# Plot 1: Without Auxiliary Loss
ax1 = axes[0]
im1 = ax1.imshow(history_no_loss.T, aspect='auto', cmap='viridis', origin='lower')
ax1.set_title("Without Load Balancing Loss (Skewed Utilization)")
ax1.set_ylabel("Expert ID")
fig.colorbar(im1, ax=ax1, label="Selection Probability")

# Plot 2: With Auxiliary Loss
ax2 = axes[1]
im2 = ax2.imshow(history_with_loss.T, aspect='auto', cmap='viridis', origin='lower')
ax2.set_title("With Load Balancing Loss (Balanced Utilization)")
ax2.set_xlabel("Training Step")
ax2.set_ylabel("Expert ID")
fig.colorbar(im2, ax=ax2, label="Selection Probability")

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

Explanation of the Visualization:

- **Top Plot (Without Load Balancing Loss):** You will see that over time, a few horizontal lines (representing specific experts) become bright yellow, while most others remain dark purple. This illustrates the "rich-get-richer" problem: the gating network quickly learns to favor a small subset of experts, ignoring the rest.
- **Bottom Plot (With Load Balancing Loss):** The colors are much more evenly distributed across all experts throughout the training steps. While there are still fluctuations, no single expert is consistently favored or starved. The auxiliary loss successfully forces the model to explore and utilize all its experts, leading to a more stable and balanced training dynamic. This is the desired behavior for an MoE model.

References

- Shazeer, Noam, et al. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." *arXiv preprint arXiv:1701.06538* (2017).
- Helwan, Abdul Kader. "Mixture of Experts-Introduction." *Medium*, 2021.
- Jain, Sm. "Understanding the Mixture-of-Experts Model in Deep Learning." *Medium*, 2021.