BI的设计

在ShortGPT中，区块影响力（Block Influence, BI）的设计核心是通过量化模型各层的隐性状态变化来评估其重要性，并基于此进行冗余层的剪枝。具体设计如下：

1. **BI指标的定义**
   BI通过测量连续层输出之间的隐性状态变化程度来评估每层的重要性。具体而言，它基于层输出的点积或余弦相似度计算，公式为：
   $$BI = 1 - \text{cosine_similarity}(h_{l}, h_{l+1})$$其中，h_l和h_{l+1}表示相邻层的隐状态。BI值越高，表明该层对模型输出的影响越大，反之则越冗余。

2. **冗余层的识别与剪枝**

   - 首先对模型所有层的BI分数进行排序，选择BI值较低的层作为冗余层。例如，在Llama-2-7B模型中，剪枝了层27、26、25等共9层（占总层数28%）。
   - 实验表明，即使剪除25%的层（如Llama-2-13B剪去10层），模型仍能保持95%的MMLU基准性能，同时减少参数和计算量。
   - 剪枝策略与量化方法正交，可结合使用以进一步优化部署效率。

3. **实现细节**

   - 在HuggingFace模型（如Llama、Qwen）中，通过封装模型收集各层的隐状态，动态计算BI分数。
   - 实际剪枝时，需使用特定数据集（如Wikitext）计算BI值，确保数据驱动的决策。
   - 代码实现中提供了剪枝脚本（如 `short_qwen_xin.sh` ）和推理验证流程，支持自定义剪枝比例。

4. **对比与改进**

   - 相比传统剪枝方法（如基于权重幅度的剪枝），BI更直接反映层对模型功能的影响，因此效果更优。
   - 后续研究（如FoldGPT）指出BI的局限性，并提出可学习的门控系数替代静态BI分数，通过微调进一步提升剪枝效果。

总结来看，ShortGPT的BI设计通过动态评估层间状态变化，简化了模型压缩流程，同时揭示了LLM在深度维度的高度冗余，为高效训练与部署提供了新思路。

BI的示例代码

以下是基于ShortGPT中**区块影响力（BI）设计**的核心代码示例，展示了如何实现隐状态相似度计算、冗余层识别与模型剪枝的关键步骤。代码基于PyTorch和HuggingFace Transformers库实现：

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
from sklearn.metrics.pairwise import cosine_similarity

# 工具函数：计算层间余弦相似度
def calculate_bi(hidden_states):
    bi_scores = []
    for i in range(len(hidden_states)-1):
        # 将隐状态展平为向量 [batch_size * seq_len, hidden_dim]
        h_current = hidden_states[i].view(-1, hidden_states[i].size(-1
        h_next = hidden_states[i+1].view(-1, hidden_states[i+1].size(-

        # 计算余弦相似度 (batch级平均)
        sim = cosine_similarity(h_current, h_next).mean()
        bi = 1 - sim  # BI分数 = 1 - 相似度
        bi_scores.append(bi)
    return bi_scores

# Step 1: 加载模型并捕获中间层输出
model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")

# 注册钩子捕获所有层的输出
hidden_states = []
def hook_fn(module, input, output):
    hidden_states.append(output[0])  # 保存隐状态（假设为tuple输出）

# 为每个Transformer层注册钩子
for layer in model.model.layers:  # 注：不同模型层路径可能不同
    layer.register_forward_hook(hook_fn)

# Step 2: 用样例数据计算BI分数
input_text = "The capital of France is"
```

```
inputs = tokenizer(input_text, return_tensors="pt")
with torch.no_grad():
    outputs = model(**inputs)

# 计算所有相邻层的BI分数
bi_scores = calculate_bi(hidden_states)

# Step 3: 识别冗余层（示例：剪除最低BI的25%层）
total_layers = len(model.model.layers)
prune_ratio = 0.25
num_prune = int(total_layers * prune_ratio)

# 按BI分数升序排列（低BI=高冗余）
sorted_indices = sorted(range(len(bi_scores)), key=lambda k: bi_scores
prune_indices = sorted_indices[:num_prune]  # 要剪除的层索引

# Step 4: 执行模型剪枝
class PrunedModel(torch.nn.Module):
    def __init__(self, original_model, keep_indices):
        super().__init__()
        self.model = original_model
        self.keep_indices = sorted(keep_indices)  # 保留层索引（需按顺序
        # 替换原始层为保留的子集
        self.model.model.layers = torch.nn.ModuleList(
            [self.model.model.layers[i] for i in self.keep_indices]
        )
        # 重要：更新模型配置中的层数
        self.model.config.num_hidden_layers = len(self.keep_indices)

    def forward(self, **inputs):
        return self.model(**inputs)

# 生成保留层索引（排除prune_indices）
all_indices = set(range(total_layers))
keep_indices = list(all_indices - set(prune_indices))

# 创建剪枝后的模型
pruned_model = PrunedModel(model, keep_indices)

# 验证剪枝效果
print(f"原始层数: {total_layers}, 剪枝后层数: {len(keep_indices)}")
# 示例输出：原始层数: 32, 剪枝后层数: 24
```

关键说明：

1. **隐状态捕获**：通过PyTorch的 `register_forward_hook` 捕获每层输出，实际使用时需要根据模型结构调整层路径（如GPT-J的 `transformer.h` ，Llama的 `model.layers` ）

2. **BI计算优化**：

```
# 实际实现中可批量处理多个数据样本：
from tqdm import tqdm
dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="tra

bi_scores = [0] * (total_layers - 1)
for batch in tqdm(dataset.select(range(100))):  # 用100个样本计算
    inputs = tokenizer(batch["text"], return_tensors="pt", truncat
    hidden_states.clear()
    _ = model(**inputs)
    scores = calculate_bi(hidden_states)
    bi_scores = [s + scores[i] for i, s in enumerate(bi_scores)]

# 平均BI分数
bi_scores = [s / 100 for s in bi_scores]
```

3. **剪枝扩展**：可与量化结合（参考LLM.int8()），代码可扩展为：

```
from bitsandbytes import quantize_blockwise
pruned_model = torch.quantization.quantize_dynamic(
    pruned_model, {torch.nn.Linear}, dtype=torch.qint8
)
```

该示例展示了BI设计的核心逻辑，完整实现需根据具体模型结构调整钩子注册位置和层索引处理。实际应用时建议参考ShortGPT官方仓库的剪枝脚本（如 `prune_llama.py` ）。