思路就是用一个比较好的模型来指导训练我们要训练的模型，比如像下边这个用Qwen1.5B(Reference Model)来指导训练Qwen0.5B(Training Model)。思路就是训练Training Model过程中计算出token的loss之后，跟Reference的loss进行比较，差值记为diff，然后drop掉百分之五十的token也就是把他们loss改成0反向传播就不管他们了，只继续训练差值大的百分之五十就行，这样提高一些训练效率。

图解LLM训练和推理的秘密-1 - 知乎 (zhihu.com)

```
import torch
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence
from transformers import AutoModelForCausalLM, AutoTokenizer
from datasets import load_dataset  # New import for datasets library

# Load the reference model and tokenizer
reference_tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-1.5B")
reference_model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen2.5-1.5B")

# Load the tinyLlama model and tokenizer
tinylama_tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B")
tinylama_model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen2.5-0.5B")

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
reference_model.to(device)
tinylama_model.to(device)

# New: Load the Wikimedia dataset
dataset_name = 'wikitext'  # You can also use 'wiki40b' or other datasets
dataset_version = 'wikitext-103-raw-v1'  # For 'wikitext' dataset
split = 'train'  # You can choose 'train', 'validation', or 'test'

# Load the dataset using Hugging Face Datasets
raw_datasets = load_dataset(dataset_name, dataset_version, split=split)
print(f"Loaded {len(raw_datasets)} examples from the {split} split of
{dataset_name}")

class ReferenceDataset(Dataset):
    def __init__(self, datasets, tokenizer, block_size, reference_model, device):
        self.examples = []
        self.reference_losses = []
        self.tokenizer = tokenizer
        self.reference_model = reference_model
        self.device = device
        self.block_size = block_size
        self.prepare_dataset(datasets)

    def prepare_dataset(self, datasets):
        self.reference_model.eval()
        with torch.no_grad():
            for example in datasets:
                text = example['text']
                # Tokenize and split into blocks
                tokens = self.tokenizer.encode(text, add_special_tokens=False)
                for i in range(0, len(tokens), self.block_size):
                    block_tokens = tokens[i:i + self.block_size]
                    input_ids = torch.tensor(block_tokens,
dtype=torch.long).unsqueeze(0).to(self.device)
                    labels = input_ids.clone()

                    # Compute reference loss
                    outputs = self.reference_model(input_ids)
                    logits = outputs.logits
                    shift_logits = logits[..., :-1, :].contiguous()
                    shift_labels = labels[..., 1:].contiguous()
                    loss_fct = torch.nn.CrossEntropyLoss(reduction='none')
                    loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)),
shift_labels.view(-1))
                    loss = loss.view(shift_labels.size())  # Shape: [1, seq_len -
1]

                    # Store input_ids and reference loss
                    self.examples.append(input_ids.squeeze(0).cpu())
                    self.reference_losses.append(loss.squeeze(0).cpu())

    def __len__(self):
        return len(self.examples)

    def __getitem__(self, idx):
        return self.examples[idx], self.reference_losses[idx]

def collate_fn(batch):
    input_ids_list, ref_losses_list = zip(*batch)
    input_ids_padded = pad_sequence(input_ids_list, batch_first=True,
padding_value=reference_tokenizer.pad_token_id)
    ref_losses_padded = pad_sequence(ref_losses_list, batch_first=True,
padding_value=0.0)
    return input_ids_padded, ref_losses_padded

# Hyperparameters
block_size = 128  # Maximum sequence length for each example
batch_size = 4
num_epochs = 3
learning_rate = 1e-5

# Prepare the dataset and dataloader
print("Preparing the dataset...")
dataset = ReferenceDataset(raw_datasets, reference_tokenizer, block_size,
reference_model, device)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
collate_fn=collate_fn)
print("Dataset preparation complete.")

# Optimizer
optimizer = torch.optim.AdamW(tinylama_model.parameters(), lr=learning_rate)
```

```
# Training loop
tinylama_model.train()
for epoch in range(num_epochs):
    print(f"Starting epoch {epoch+1}/{num_epochs}")
    for batch_idx, (input_ids, ref_losses) in enumerate(dataloader):
        input_ids = input_ids.to(device)
        ref_losses = ref_losses.to(device)

        # Prepare inputs and labels for tinyLlama
        inputs = input_ids[:, :-1]  # Exclude last token
        labels = input_ids[:, 1:]   # Exclude first token

        # Forward pass through tinyLlama
        outputs = tinylama_model(inputs, labels=labels)
        logits = outputs.logits
        shift_logits = logits.contiguous()
        shift_labels = labels.contiguous()

        # Compute training loss per token
        loss_fct = torch.nn.CrossEntropyLoss(reduction='none')
        training_loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)),
shift_labels.view(-1))
        training_loss = training_loss.view(shift_labels.size())  # Shape:
[batch_size, seq_len - 1]

        # Compute excess loss: training loss minus reference loss
        excess_loss = training_loss - ref_losses

        # Flatten excess loss to select top 30%
        flat_excess_loss = excess_loss.view(-1)
        num_tokens = flat_excess_loss.size(0)
        top_k = int(num_tokens * 0.3)

        if top_k == 0:
            # If the number of tokens is too small, default to keeping all tokens
            mask = torch.ones_like(flat_excess_loss)
        else:
            # Select indices of top 30% excess losses
            top_values, top_indices = torch.topk(flat_excess_loss, k=top_k)
            mask = torch.zeros_like(flat_excess_loss)
            mask[top_indices] = 1.0

        mask = mask.view(excess_loss.size())  # Shape: [batch_size, seq_len - 1]

        # Zero out losses not in top 30%
        final_loss = (training_loss * mask).sum() / mask.sum()

        # Backpropagation
        optimizer.zero_grad()
        final_loss.backward()
        optimizer.step()

        if (batch_idx + 1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}], Batch
[{batch_idx+1}/{len(dataloader)}], Loss: {final_loss.item():.4f}")

    print(f"Epoch {epoch+1} complete.")

# Save the trained tinyLlama model
tinylama_model.save_pretrained('path/to/save/tinylama_model')
tinylama_tokenizer.save_pretrained('path/to/save/tinylama_tokenizer')
```