Resources

1. https://medium.com/@palanikalyan27/building-your-own-llm-from-scratch-a-comprehensive-guide-7e38d9624d47

```
#install
!pip -q install torch numpy transformers datasets tokenizers wandb tqdm
# Install Flask and other required libraries
```

Downloading Dataset

This downloads data from wikpedia

```
from datasets import load_dataset
# Load Wikitext dataset (a smaller dataset for demonstration)
dataset = load_dataset("wikitext", "wikitext-2-raw-v1")
print(f"Train set size: {len(dataset['train'])}")
print(f"Sample text: {dataset['train'][0]['text'][:200]}")
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your sett
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to a
  warnings.warn(
Train set size: 36718
Sample text:
```

Build Tokenizer

```
from tokenizers import ByteLevelBPETokenizer
import os


# Initialize a tokenizer
tokenizer = ByteLevelBPETokenizer()
# Prepare training files
def get_training_corpus():
    for i in range(0, len(dataset["train"])):
        yield dataset["train"][i]["text"]
# Train the tokenizer
tokenizer.train_from_iterator(
```

```
tokenizer.train_from_iterator(
    get_training_corpus(),
    vocab_size=30000,
    min_frequency=2,
    special_tokens=["<s>", "<pad>", "</s>", "<unk>", "<mask>"]
)
#Create the directory if it doesn't exist
os.makedirs("tokenizer", exist_ok=True)

# save the model files (merges.txt and vocab.json)
tokenizer.save("tokenizer/tokenizer.json")
```

Double-click (or enter) to edit

Load

Double-click (or enter) to edit

```
from transformers import PreTrainedTokenizerFast
# Load the trained tokenizer
tokenizer = PreTrainedTokenizerFast(tokenizer_file="tokenizer/tokeniz
tokenizer.pad_token = "<pad>"
tokenizer.eos_token = "</s>"
tokenizer.bos_token = "<s>"
tokenizer.unk_token = "<unk>"
# Define maximum sequence length
max_length = 128
# Tokenize function
def tokenize_function(examples):
    return tokenizer(
        examples["text"],
        padding="max_length",
        truncation=True,
        max_length=max_length,
        return_tensors="pt"
    )
# Apply tokenization to the dataset
tokenized_datasets = dataset.map(
    tokenize_function,
    batched=True,
    remove_columns=["text"]
)
# Prepare for training
tokenized_datasets.set_format("torch")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
```

```
last)
/tmp/ipython-input-2945183752.py in <cell line: 0>()
----> 1 from transformers import PreTrainedTokenizerFast
      2 # Load the trained tokenizer
      3 tokenizer = PreTrainedTokenizerFast(tokenizer_file="tokenizer/
tokenizer.json")
      4 tokenizer.pad_token = "<pad>"
      5 tokenizer.eos_token = "</s>"

8 frames

/usr/local/lib/python3.12/dist-packages/torch/fx/experimental/
const_fold.py in <module>
     15
     16
---> 17 class FoldedGraphModule(torch.fx.GraphModule):
     18     """
     19     FoldedGraphModule is a GraphModule which also contains
another
```

Building the Model Architecture

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        super().__init__()
        self.q = nn.Linear(embed_dim, head_dim)
        self.k = nn.Linear(embed_dim, head_dim)
        self.v = nn.Linear(embed_dim, head_dim)

    def forward(self, hidden_state):
        attn_outputs = self._attention(
            self.q(hidden_state),
            self.k(hidden_state),
            self.v(hidden_state)
        )
        return attn_outputs

    def _attention(self, query, key, value):
        # Scaled dot-product attention
        attn_scores = torch.matmul(query, key.transpose(-2, -1)) / (k

        # Create causal mask (lower triangular)
        seq_length = query.size(1)
        causal_mask = torch.triu(torch.ones(seq_length, seq_length),
        causal_mask = causal_mask.to(query.device)

        # Apply causal mask by setting masked positions to -inf
        attn_scores = attn_scores.masked_fill(causal_mask, -1e10)
```

```
        attn_scores = attn_scores.masked_fill(causal_mask, -1e10)

        # Apply softmax to get attention weights
        attn_weights = F.softmax(attn_scores, dim=-1)

        # Apply attention weights to values
        return torch.matmul(attn_weights, value)
class MultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        embed_dim = config.hidden_size
        num_heads = config.num_attention_heads
        head_dim = embed_dim // num_heads

        self.heads = nn.ModuleList(
            [AttentionHead(embed_dim, head_dim) for _ in range(num_hea
        )
        self.output_linear = nn.Linear(embed_dim, embed_dim)

    def forward(self, hidden_states):
        head_outputs = [head(hidden_states) for head in self.heads]
        concatenated = torch.cat(head_outputs, dim=-1)
        return self.output_linear(concatenated)
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.linear1 = nn.Linear(config.hidden_size, config.intermedi
        self.linear2 = nn.Linear(config.intermediate_size, config.hid
        self.activation = nn.GELU()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.dropout(x)
        x = self.linear2(x)
        return x
class TransformerBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.attention = MultiHeadAttention(config)
        self.layer_norm1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm2 = nn.LayerNorm(config.hidden_size)
        self.feed_forward = FeedForward(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        # Self-attention with residual connection and layer norm
        residual = x
        x = self.layer_norm1(x)
        x = self.attention(x)
```

```
        x = self.dropout(x)
        x = x + residual

        # Feed-forward with residual connection and layer norm
        residual = x
        x = self.layer_norm2(x)
        x = self.feed_forward(x)
        x = self.dropout(x)
        x = x + residual

        return x
class GPTConfig:
    def __init__(
        self,
        vocab_size=30000,
        hidden_size=768,
        num_hidden_layers=12,
        num_attention_heads=12,
        intermediate_size=3072,
        hidden_dropout_prob=0.1,
        max_position_embeddings=512,
    ):
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.num_hidden_layers = num_hidden_layers
        self.num_attention_heads = num_attention_heads
        self.intermediate_size = intermediate_size
        self.hidden_dropout_prob = hidden_dropout_prob
        self.max_position_embeddings = max_position_embeddings
class SimpleLLM(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        # Token embeddings
        self.token_embeddings = nn.Embedding(config.vocab_size, confi

        # Position embeddings
        self.position_embeddings = nn.Embedding(
            config.max_position_embeddings, config.hidden_size
        )

        # Transformer blocks
        self.transformer_blocks = nn.ModuleList(
            [TransformerBlock(config) for _ in range(config.num_hidde
        )

        # Layer norm
        self.layer_norm = nn.LayerNorm(config.hidden_size)

        # Output head
```

```
        # output head
        self.output = nn.Linear(config.hidden_size, config.vocab_size

        # Initialize weights
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding)):
            module.weight.data.normal_(mean=0.0, std=0.02)
            if isinstance(module, nn.Linear) and module.bias is not N
                module.bias.data.zero_()
        elif isinstance(module, nn.LayerNorm):
            module.bias.data.zero_()
            module.weight.data.fill_(1.0)

    def forward(self, input_ids):
        batch_size, seq_length = input_ids.size()

        # Get token embeddings
        token_embeds = self.token_embeddings(input_ids)

        # Create position IDs and embeddings
        position_ids = torch.arange(seq_length, dtype=torch.long, dev
        position_ids = position_ids.unsqueeze(0).expand(batch_size, -
        position_embeds = self.position_embeddings(position_ids)

        # Combine token and position embeddings
        x = token_embeds + position_embeds

        # Pass through transformer blocks
        for block in self.transformer_blocks:
            x = block(x)

        # Apply final layer norm
        x = self.layer_norm(x)

        # Get logits
        logits = self.output(x)

        return logits
```

Smaller model for train on modest hardware

```
# Define a smaller model configuration
config = GPTConfig(
    vocab_size=tokenizer.vocab_size,
    hidden_size=256,
    num_hidden_layers=4,
    num_attention_heads=4,
```

```
        intermediate_size=512,
        max_position_embeddings=max_length
    )
```

```
model = SimpleLLM(config)
print(f"Model parameters: {sum(p.numel() for p in model.parameters())
```

Training LLM

For Limited Hardware Training:

If you're working with limited computational resources:

```
Reduce model size: Fewer layers, smaller hidden dimensions
Use mixed precision training: Enable PyTorch's automatic mixed precision
Gradient accumulation: Update weights after multiple forward/backward pas
Train on smaller dataset: Use a subset of your data
Consider distributed training: Split across multiple GPUs if available
```

Modified training loop with these optimizations:

```
!pip install bitsandbytes
```

```
Requirement already satisfied: bitsandbytes in /usr/local/lib/python3.
Requirement already satisfied: torch<3,>=2.3 in /usr/local/lib/python3
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.1
Requirement already satisfied: packaging>=20.9 in /usr/local/lib/pytho
Requirement already satisfied: filelock in /usr/local/lib/python3.12/d
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local
Requirement already satisfied: setuptools in /usr/local/lib/python3.12
Requirement already satisfied: sympy>=1.13.3 in /usr/local/lib/python3
Requirement already satisfied: networkx>=2.5.1 in /usr/local/lib/pytho
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dis
Requirement already satisfied: fsspec>=0.8.5 in /usr/local/lib/python3
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.6.77 in /usr
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.6.77 in /u
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.6.80 in /usr
Requirement already satisfied: nvidia-cudnn-cu12==9.10.2.21 in /usr/lo
Requirement already satisfied: nvidia-cublas-cu12==12.6.4.1 in /usr/lo
Requirement already satisfied: nvidia-cufft-cu12==11.3.0.4 in /usr/loc
Requirement already satisfied: nvidia-curand-cu12==10.3.7.77 in /usr/l
Requirement already satisfied: nvidia-cusolver-cu12==11.7.1.2 in /usr/
Requirement already satisfied: nvidia-cusparse-cu12==12.5.4.2 in /usr/
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in /usr/l
Requirement already satisfied: nvidia-nccl-cu12==2.27.5 in /usr/local/
```

```
Requirement already satisfied: nvidia-nvshmem-cu12==3.3.20 in /usr/loc
Requirement already satisfied: nvidia-nvtx-cu12==12.6.77 in /usr/local
Requirement already satisfied: nvidia-nvjitlink-cu12==12.6.85 in /usr/
Requirement already satisfied: nvidia-cufile-cu12==1.11.1.6 in /usr/lo
Requirement already satisfied: triton==3.5.0 in /usr/local/lib/python3
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/py
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/pytho
```

```
import torch
import torch.nn.functional as F
from torch.cuda import amp
from torch.utils.data import DataLoader
from tqdm.auto import tqdm
import bitsandbytes as bnb
from transformers import AutoModelForCausalLM, AutoConfig

# --- INITIALIZATION ---
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# FIX: Create the actual model instance here
config = AutoConfig.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_config(config)
model.to(device)

# Initialize Optimizer BEFORE torch.compile
optimizer = bnb.optim.AdamW8bit(model.parameters(), lr=5e-5)

# OPTIMIZATION: Compile the model for fused kernels
model = torch.compile(model)

scaler = amp.GradScaler()
accumulation_steps = 4

train_dataloader = DataLoader(
    tokenized_datasets["train"],
    batch_size=16,
    shuffle=True,
    pin_memory=True, # OPTIMIZATION: Faster data transfer to GPU
    num_workers=2    # OPTIMIZATION: Use Colab's 2-core CPU for loadi
)

# --- TRAINING LOOP ---
model.train()
for epoch in range(5):
    epoch_loss = 0
    progress_bar = tqdm(train_dataloader, desc=f"Epoch {epoch+1}")

    for step, batch in enumerate(progress_bar):
        # non_blocking=True overlaps data transfer with compute
        input_ids = batch["input_ids"].to(device, non_blocking=True)
        labels = input_ids.clone()
```

```
        with amp.autocast():
            outputs = model(input_ids, labels=labels)
            loss = outputs.loss / accumulation_steps

        scaler.scale(loss).backward()

        if (step + 1) % accumulation_steps == 0:
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_no
            scaler.step(optimizer)
            scaler.update()
            optimizer.zero_grad(set_to_none=True) # OPTIMIZATION: Sav

        epoch_loss += loss.item() * accumulation_steps
        progress_bar.set_postfix({"loss": loss.item() * accumulation_
```

```
/tmp/ipython-input-4256028208.py:23: FutureWarning: `torch.cuda.amp.Gr
  scaler = amp.GradScaler()
```

Epoch 1: 100% 2295/2295 [06:38<00:00, 18.04s/it, loss=3.27]

```
/tmp/ipython-input-4256028208.py:45: FutureWarning: `torch.cuda.amp.au
  with amp.autocast():
/usr/local/lib/python3.12/dist-packages/torch/backends/cuda/__init__.p
  return torch._C._get_cublas_allow_tf32()
W0117 06:14:01.913000 904 torch/_inductor/utils.py:1558] [0/0_1] Not e
`loss_type=None` was set in the config but it is unrecognized. Using t
/tmp/ipython-input-4256028208.py:45: FutureWarning: `torch.cuda.amp.au
  with amp.autocast():
```

Epoch 2: 100% 2295/2295 [05:06<00:00, 8.01it/s, loss=3.24]

```
/tmp/ipython-input-4256028208.py:45: FutureWarning: `torch.cuda.amp.au
  with amp.autocast():
```

Epoch 3: 100% 2295/2295 [05:05<00:00, 8.06it/s, loss=1.67]

Download Model

```
import shutil
from google.colab import files

# 1. First, save the trained model instance to a directory
model.save_pretrained("trained_llm") # Uncomment if not already saved
tokenizer.save_pretrained("trained_llm")
```

```
# 2. Compress the folder into a ZIP file
shutil.make_archive('trained_llm_archive', 'zip', 'trained_llm')

# 3. Download the ZIP file to your local computer
files.download('trained_llm_archive.zip')
```

Double-click (or enter) to edit

Evaluation and Fine Tuning

```
from torch.utils.data import DataLoader
import torch
import torch.nn.functional as F
from tqdm.auto import tqdm

# 1. Prepare evaluation dataloader with optimizations for T4
eval_dataloader = DataLoader(
    tokenized_datasets["validation"],
    batch_size=16,
    pin_memory=True,  # Speeds up host-to-device transfers
    num_workers=2     # Matches standard Colab CPU core count
)

# 2. Optimized Evaluation function
def evaluate(model, dataloader):
    model.eval()
    total_loss = 0
    total_tokens = 0

    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Evaluating"):
            # Use non_blocking=True for asynchronous data transfer
            input_ids = batch["input_ids"].to(device, non_blocking=Tru
            labels = input_ids.clone()

            # Forward pass
            outputs = model(input_ids)

            # FIX: Access .logits attribute from CausalLMOutput objec
            logits = outputs.logits

            # Calculate loss using the extracted logits tensor
            # Reduction='sum' is used to accumulate total cross-entro
            loss = F.cross_entropy(
```

```
                logits.view(-1, logits.size(-1)),
                labels.view(-1),
                ignore_index=tokenizer.pad_token_id,
                reduction="sum"
            )

            # Count only non-padding tokens for accurate perplexity
            num_tokens = labels.ne(tokenizer.pad_token_id).sum().item

            total_loss += loss.item()
            total_tokens += num_tokens

    # 3. Calculate perplexity: exp(average negative log-likelihood)
    avg_loss = total_loss / total_tokens
    perplexity = torch.exp(torch.tensor(avg_loss))
    return perplexity.item()

# 4. Run Evaluation
perplexity = evaluate(model, eval_dataloader)
print(f"Validation perplexity: {perplexity:.2f}")

# # 5. Log to WandB
# if 'wandb' in globals():
#     wandb.log({"perplexity": perplexity})
```

Evaluating: 100% 235/235 [00:33<00:00, 6.91it/s]

```
import os
os.kill(os.getpid(), 9)
```

Import Previous Model

```
import os
import shutil

# Define paths
zip_path = '/content/trained_llm_archive.zip'
extract_folder = '/content/trained_llm'

#Create the destination folder if it doesn't exist
os.makedirs(extract_folder, exist_ok=True)

# Unzip the file into the folder
if os.path.exists(zip_path):
```

```
if os.path.exists(zip_path):
    shutil.unpack_archive(zip_path, extract_folder)
    print(f"Extracted to: {extract_folder}")

    # List files
    print("Files in folder:", os.listdir(extract_folder))
else:
    print(f"Error: {zip_path} not found. Please check the file path."
```

```
Extracted to: /content/trained_llm
Files in folder: ['special_tokens_map.json', 'model.safetensors', 'gen
```

Loading Model

```
import torch
import torch.fx

from transformers import AutoModelForCausalLM, AutoTokenizer

#Define paths and device
model_path = "/content/trained_llm"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Loading model on {device}...")

# Load model
model = AutoModelForCausalLM.from_pretrained(
    model_path,
    torch_dtype=torch.float16,
    device_map="auto",
    low_cpu_mem_usage=True
)

tokenizer = AutoTokenizer.from_pretrained(model_path)

if not hasattr(model, 'device'):
    model.device = device

model.eval()
print(f"Model successfully loaded on {model.device}")
```

```
Loading model on cuda...
Model successfully loaded on cuda:0
```

Text Generation

Let's implement text generation using our trained model:

```
def generate_text(model, tokenizer, prompt, max_length=100, temperatu
    model.eval()

    # Tokenize the prompt
    input_ids = tokenizer.encode(prompt, return_tensors="pt").to(devi

    # Generate tokens
    with torch.no_grad():
        for _ in range(max_length):
            # Get model predictions
            outputs = model(input_ids)
            next_token_logits = outputs[:, -1, :] / temperature

            # Sample from the distribution
            probs = F.softmax(next_token_logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)

            # Append the new token
            input_ids = torch.cat([input_ids, next_token], dim=-1)

            # Stop if EOS token is generated
            if next_token.item() == tokenizer.eos_token_id:
                break

    # Decode the generated tokens
    generated_text = tokenizer.decode(input_ids[0], skip_special_toke
    return generated_text
```

Test Text Generation

```
def generate_text(model, tokenizer, prompt, max_new_tokens=50):
    model.eval()


    inputs = tokenizer(prompt, return_tensors="pt").to(model.device)


    with torch.no_grad():
        output_ids = model.generate(
            inputs["input_ids"],
            max_new_tokens=max_new_tokens,
            pad_token_id=tokenizer.pad_token_id,
            eos_token_id=tokenizer.eos_token_id,
            do_sample=True,
            temperature=0.7,
```

```
            top_p=0.9,
            repetition_penalty=1.2
        )

    generated_tokens = output_ids[0][inputs["input_ids"].shape[1]:]
    return tokenizer.decode(generated_tokens, skip_special_tokens=Tru

# Test text generation
sample_prompt = "Artificial intelligence is"
generated_text = generate_text(model, tokenizer, sample_prompt)

print(f"Prompt: {sample_prompt}")
print(f"Generated: {generated_text}")
```

```
Prompt: Artificial intelligence is
Generated:  a large person in the late 19th century . This was found a
```

Fine-tuning for Specific Tasks

To adapt your model for specific tasks, you can fine-tune it on task-specific data:

Load Data

```
# Load a task-specific dataset (e.g., for sentiment analysis)
task_dataset = load_dataset("imdb")
```

```
# Load your tokenizer
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("trained_llm")

tokenizer.model_max_length = 512

def preprocess_function(examples):
    return tokenizer(
        examples["text"],
        padding="max_length",
        truncation=True,
        max_length=tokenizer.model_max_length  # Uses the 512 we just
    )

# Apply mapping
processed_datasets = task_dataset.map(
    preprocess_function,
    batched=True,
    remove_columns=["text"]
```

```
    remove_columns=["text"]
)
```

```
Map: 100%    25000/25000 [00:26<00:00, 1015.64 examples/s]
Map: 100%    25000/25000 [00:31<00:00, 861.15 examples/s]
```

Deployment and Practical Applications

Once you've trained your model, you can deploy it for practical use: Model Export and Optimization

Creating Simple API

Making the Flask App

```
from flask import Flask, request, jsonify, render_template_string
from google.colab import output
import threading

app = Flask(__name__)

# HTML Template for the in-notebook view
html_code = """
<!DOCTYPE html>
<html>
<head>
    <style>
        body { font-family: sans-serif; padding: 20px; background: #f(
        .container { background: white; padding: 20px; border-radius:
        textarea { width: 100%; height: 80px; margin-bottom: 10px; bo
        button { background: #34a853; color: white; border: none; pad(
        #output { margin-top: 20px; white-space: pre-wrap; background
    </style>
</head>
<body>
    <div class="container">
        <h3>LLM Inference Interface (2026)</h3>
        <textarea id="prompt" placeholder="Enter your prompt here...":
        <button onclick="generate()">Generate Text</button>
        <div id="output">Output will appear here...</div>
    </div>
```

```
    </div>

    <script>
    async function generate() {
        const prompt = document.getElementById('prompt').value;
        const outDiv = document.getElementById('output');
        outDiv.innerText = "Generating...";

        const response = await fetch('/generate', {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({prompt: prompt, max_length: 50})
        });
        const data = await response.json();
        outDiv.innerText = data.generated_text || data.error;
    }
    </script>
</body>
</html>
"""

@app.route("/")
def index():
    return render_template_string(html_code)

@app.route("/generate", methods=["POST"])
def generate():
    data = request.json
    prompt = data.get("prompt", "")
    try:
        # Use the generate_text function we fixed earlier
        res = generate_text(model, tokenizer, prompt, max_new_tokens=5
        return jsonify({"generated_text": res})
    except Exception as e:
        return jsonify({"error": str(e)}), 500

# Kill existing process on port 5000
!fuser -k 5000/tcp

# Start Flask in background
threading.Thread(target=app.run, kwargs={"host": "0.0.0.0", "port": 5
```

Generate Text Function

```
import torch

def generate_text(model, tokenizer, prompt, max_new_tokens=100, tempe
    """
```

```
    Optimized generation function for T4 GPU (2026).
    """
    model.eval()

    # Ensure the model has a .device attribute (fix for custom classe
    device = getattr(model, 'device', torch.device("cuda" if torch.cu

    # Tokenize and move to device
    inputs = tokenizer(prompt, return_tensors="pt").to(device)

    with torch.no_grad():
        output_ids = model.generate(
            inputs["input_ids"],
            max_new_tokens=max_new_tokens,
            do_sample=True,
            temperature=temperature,
            top_p=0.9,
            pad_token_id=tokenizer.pad_token_id,
            eos_token_id=tokenizer.eos_token_id,
            repetition_penalty=1.2
        )

    # Decode only the newly generated tokens (slice off the prompt)
    generated_tokens = output_ids[0][inputs["input_ids"].shape[-1]:]
    return tokenizer.decode(generated_tokens, skip_special_tokens=Tru
```

View Flask App using iframe before implementation of safety features

```
from google.colab import output

# This will create a window below this cell showing your Flask app
output.serve_kernel_port_as_iframe(5000, height='400')
```

Implementing Basic Safety Measures

```
#Filter
def is_harmful(text):
    #safety lift
    harmful_keywords = [
        "hate speech", "violence", "illegal", "offensive",
        "self-harm", "exploit", "harassment", "weapon"
    ]

    content = text.lower()
    for keyword in harmful_keywords:
        if keyword in content:
            return True
    return False

# 2.Text generation with Safety Guardrails
def generate_text_safe(model, tokenizer, prompt, max_new_tokens=100):
    #Input Sanitization
    if is_harmful(prompt):
        return "Your request contains potentially harmful content and

    model.eval()
    inputs = tokenizer(prompt, return_tensors="pt").to(model.device)

    with torch.no_grad():
        output_ids = model.generate(
            inputs["input_ids"],
            max_new_tokens=max_new_tokens,
            do_sample=True,
            temperature=0.7,
            pad_token_id=tokenizer.pad_token_id,
            eos_token_id=tokenizer.eos_token_id
        )

    generated_tokens = output_ids[0, inputs["input_ids"].shape[-1]:]
    generated_text = tokenizer.decode(generated_tokens, skip_special_
```

```
    generated_text = tokenizer.decode(generated_tokens, skip_special_

    #Output Guardrail
    if is_harmful(generated_text):
        return "I cannot generate that content as it may violate ethi

    return generated_text
```

```
from flask import Flask, request, jsonify, render_template_string
from google.colab import output
import threading

app = Flask(__name__)

# --- 1. SAFETY LOGIC ---
def is_harmful(text):
    harmful_keywords = ["hate", "violence", "illegal", "offensive", "we
    content = text.lower()
    return any(keyword in content for keyword in harmful_keywords)

# --- 2. HTML INTERFACE ---
html_code = """
<!DOCTYPE html>
<html>
<head>
    <style>
        body { font-family: sans-serif; padding: 20px; background: #f2f
        .container { background: white; padding: 20px; border-radius: 1
        textarea { width: 100%; height: 100px; margin-bottom: 15px; bor
        button { background: #007bff; color: white; border: none; paddi
        button:hover { background: #0056b3; }
        #output { margin-top: 20px; white-space: pre-wrap; background:
        .warning { color: #d9534f; font-weight: bold; border-left: 4px
    </style>
</head>
<body>
    <div class="container">
        <h3>LLM Interface</h3>
        <textarea id="prompt" placeholder="Type something... (try 'viol
        <button id="genBtn" onclick="generate()">Generate Safe Response
        <div id="output">Results will appear here...</div>
    </div>

    <script>
    async function generate() {
        const prompt = document.getElementById('prompt').value;
        const outDiv = document.getElementById('output');
        const btn = document.getElementById('genBtn');

        outDiv.innerText = "Processing...";
```

```
        outDiv.innerText = "Processing ...";
        btn.disabled = true;

        try {
            const response = await fetch('/generate', {
                method: 'POST',
                headers: {'Content-Type': 'application/json'},
                body: JSON.stringify({prompt: prompt})
            });
            const data = await response.json();

            if (data && data.generated_text) {
                // Style based on the "Safety Warning" text instead of
                if (data.generated_text.includes("Safety Warning") || c
                    outDiv.innerHTML = `<div class="warning">${data.ger
                } else {
                    outDiv.innerText = data.generated_text;
                }
            } else if (data && data.error) {
                outDiv.innerHTML = `<div class="warning">Server Error:
            } else {
                outDiv.innerText = "Error: Unexpected response format."
            }
        } catch (e) {
            outDiv.innerText = "Network Error: " + e.message;
        } finally {
            btn.disabled = false;
        }
    }
    </script>
</body>
</html>
"""

@app.route("/")
def index():
    return render_template_string(html_code)

@app.route("/generate", methods=["POST"])
def generate():
    data = request.json
    prompt = data.get("prompt", "")

    #Safety check before generation
    if is_harmful(prompt):
        return jsonify({"generated_text": "Safety Warning: Your prompt

    try:
        #Generate Text
        res = generate_text(model, tokenizer, prompt, max_new_tokens=50
```

```
        #After generation safety check
        if is_harmful(res):
            return jsonify({"generated_text": "The response violated sa

        return jsonify({"generated_text": res})

    except Exception as e:
        return jsonify({"error": str(e)}), 500

#Launch interface
threading.Thread(target=app.run, kwargs={"host": "0.0.0.0", "port": 500
output.serve_kernel_port_as_iframe(5001, height='450')
```