How to Train a Large Language Model (LLM)

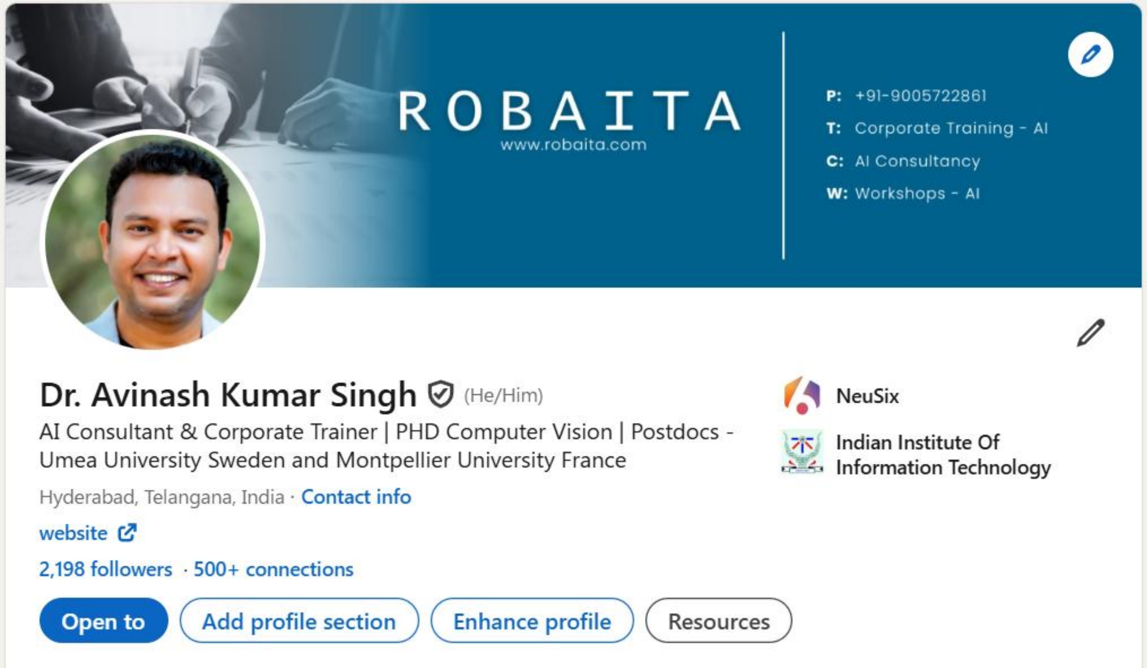
From Data Collection to Fine-Tuning

Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita

Dr. Avinash Kumar Singh

- **Possess** 15+ years of **hands-on expertise** in Machine Learning, Computer Vision, NLP, IoT, Robotics, and Generative AI.
- **Founded** Robaita—an initiative **empowering** individuals and organizations to **build, educate, and implement** AI solutions.
- **Earned** a Ph.D. in Human-Robot Interaction from IIIT Allahabad in 2016.
- **Received** postdoctoral fellowships at Umeå University, Sweden (2020) and Montpellier University, France (2021).
- **Authored** 30+ research papers in **high-impact** SCI journals and international conferences.
- Unlearning, learning, making mistakes …



https://www.linkedin.com/in/dr-avinash-kumar-singh-2a570a31/

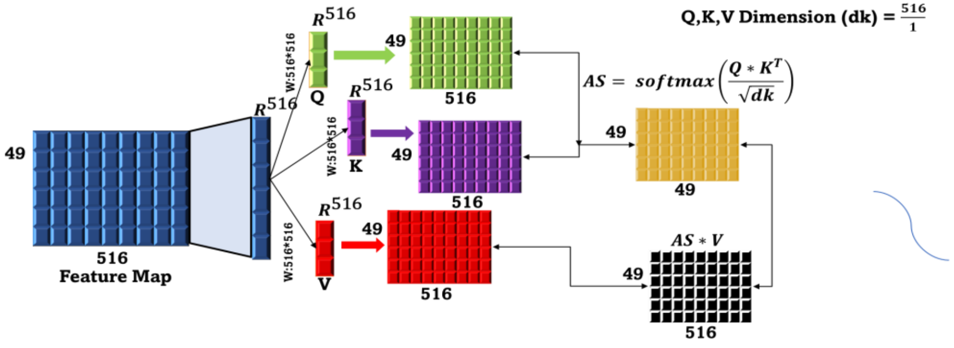
Discussion Points

- Data Collection and Preprocessing: Source Selection, Data Cleaning, Metadata Annotation, Format Standardization, Dataset Sharding and Storage
- Tokenization and Vocabulary Creation: Tokenizer Types, Vocabulary Design, Training the Tokenizer, Tokenization Pipeline, Efficiency and Compression
- Model Architecture and Configuration: Transformer Backbone, Configuration Parameters, Positional Encodings, Initialization and Optimization, Distributed Training Setup
- Pretraining Objectives: Causal Language Modeling (CLM) , Masked Language Modeling (MLM), Next Sentence Prediction.
- Fine-Tuning and Alignment Techniques: Supervised Fine-Tuning, Reinforcement Learning from Human Feedback (RLHF) , Constitutional AI / Rule-based Alignment, Parameter-Efficient Fine-Tuning (PEFT) , Evaluation and Red-Teaming

Attention Network Summary

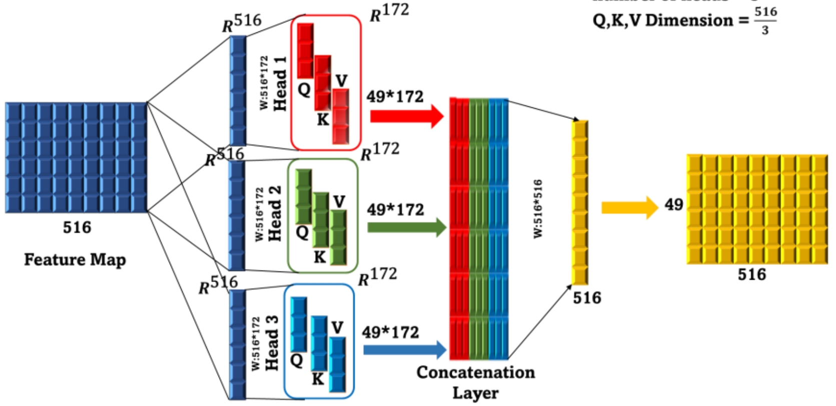
Self Attention

Feature dimension = 516
number of heads = 1
Q,K,V Dimension (dk) = $\frac{516}{1}$



Mult Head Attention

Feature dimension = 516
number of heads = 3
Q,K,V Dimension = $\frac{516}{3}$



Transformer

Cross Multi-Head Attention

Let decoder attend to encoder outputs (i.e., from "the cat sat on the mat")

Let:

- Encoder output = E ∈ (6, 4)

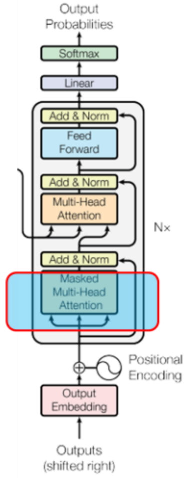
Here:

- **Q = from decoder (shape = (3, 4))**
- **K, V = from encoder output E (shape = (6, 4))**

→ Attention shape:

- Each head: (3, 2)
- Concatenated: (3, 4)

Output shape: (3, 4)



Transformers

Masked Multi Head Attention

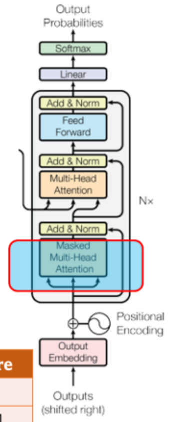
Example: *"Le chat dort" ("The cat sleeps")*

Scaled Dot Product (Q&K)

$$\text{score}_{i,j} = \frac{Q_i \cdot K_j^T}{\sqrt{d_k}}$$

	Token 1 ("Le")	Token 2 ("chat")	Token 3 ("dort")
Token 1	(0.1×0.1 + ...) = **0.63** → ÷2 = 0.315	**0.54**	**0.45**
Token 2	**0.54**	**0.61**	**0.56**
Token 3	**0.45**	**0.56**	**0.54**

Attention Score Matrix	Look-Ahead Mask	Masked Score	Softmax	Attention Score
[0.315, 0.270, 0.225]		[0.315, -∞, -∞]		[1.0, 0.0, 0.0]
[0.270, 0.305, 0.280]		[0.270, 0.305, -∞]		[0.491, 0.509, 0.0]
[0.225, 0.280, 0.270]		[0.225, 0.280, 0.270]		[0.326, 0.347, 0.327]



Data Collection and Preprocessing

Objective: Gathering text data and preparing it for training

Purpose: To provide high-quality, diverse input for the model to learn language patterns

Sources: Public datasets like Wikipedia, Common Crawl, BookCorpus

```python
from datasets import load_dataset
dataset = load_dataset('wikitext', 'wikitext-103-raw-v1')
```

Cleaning: Remove HTML, scripts, or broken sentences

- Example: Eliminate <script> tags or gibberish

Annotation: Add fields like source: Wikipedia, language: English

- Helps trace model behavior to data origins

Standardization: Convert data to JSONL format

- Easier for pipelines to handle consistent formats

Storage: Use tools like WebDataset for sharded data loading

- Example: Break large datasets into small files for distributed training

Dataset Information

Wikipedia

Languages Supported:

- ~300+ languages (e.g., English, German, French, Hindi, Japanese, etc.)
- English Wikipedia is the largest and most frequently used in NLP tasks.

Dataset Size (English):

- ~6+ million articles
- ~20 GB in raw text (compressed XML ~16 GB)

Use Cases:

- Pretraining large language models (e.g., BERT, GPT)
- Knowledge extraction, QA systems
- Summarization, entity recognition, translation
- Creating knowledge graphs and fact-checking tools

https://huggingface.co/datasets/legacy-datasets/wikipedia

Dataset Information

Book Corpus

Languages Supported:

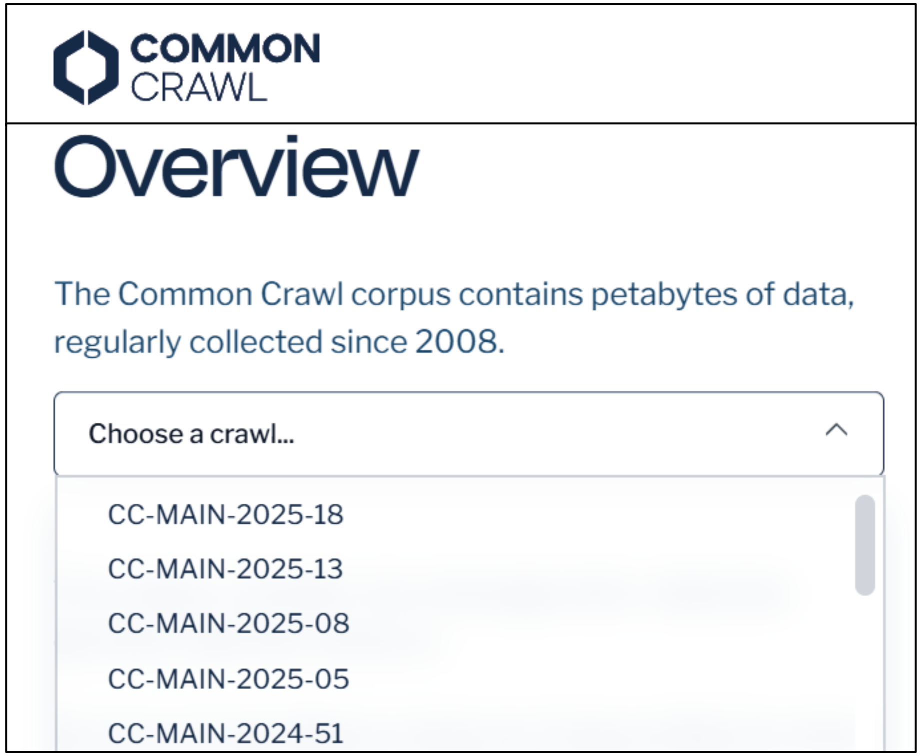
- Primarily English, but includes many other languages from web pages (multi-lingual)

Dataset Size (English):

- Petabyte-scale web archive
- Monthly crawls (100–300 TB per snapshot)
- Processed subset: CCNet (~100 GB–1 TB depending on filtering)

Use Cases:

- Pretraining massive LLMs (GPT-3, LLaMA, BLOOM)
- Training web-scale language understanding
- Building search engines or domain-specific corpora
- Training models for open-ended generation



https://commoncrawl.org/overview

Dataset Information

BookCorpus

Languages Supported:

- English only (original dataset)

Dataset Size (English):

- ~11,000 books
- ~1 GB plain text
- Some cleaned and reformatted versions available (e.g., bookcorpusopen on Hugging Face)

Use Cases:

- Pretraining for narrative and long-form text models (e.g., GPT, BERT, RoBERTa)
- Language modeling with coherent, story-like structure
- Fine-tuning models for fiction, dialog, or structured prose

https://github.com/jackbandy/bookcorpus-datasheet

Dataset Facts

Dataset BookCorpus
Instances Per Dataset 7,185 unique books, 11,038 total

Motivation	
Original Authors	Zhu and Kiros et al. (2015) [39]
Original Use Case	Sentence embedding
Funding	Google, Samsung, NSERC, CIFAR, ONR

Composition	
Sample or Complete	Sample, ≈2% of smashwords.com in 2014
Missing Data	98 empty files, ≤655 truncated files
Sensitive Information	Author email addresses

Collection	
Sampling Strategy	Free books with ≥20,000 words
Ethical Review	None stated
Author Consent	None

Cleaning and Labeling	
Cleaning Done	None stated, some implicit
Labeling Done	None stated, genres by smashwords.com

Uses and Distribution	
Notable Uses	Language models (e.g. GPT [29], BERT [9])
Other Uses	List available on HuggingFace [12]
Original Distribution	Author website (now defunct) [39]
Replicate Distribution	BookCorpusOpen [13]

Maintenance and Evolution	
Corrections or Erratum	None
Methods to Extend	"Homemade BookCorpus" [21]
Replicate Maintainers	Shawn Presser [12]

Genres	% of BookCorpus*
Romance 2,881 books	26.1%
Fantasy 1,502 books	13.6%
Vampires 600 books	5.4%

Horror 4.1% • Teen 3.9%
Adventure 3.5% • Literature 3.0%
Historical Fiction 1.6%

Not a significant source of nonfiction.

* Percentages based on directories in books_txt_full. Some books cross-listed.

Dataset Summary

Dataset	Language(s)	Size	Use Cases
Wikipedia	~300+	~20 GB (en)	QA, summarization, NER, fine-tuning LLMs, knowledge graphs
Common Crawl	Multi-lingual	100 TB+	Pretraining LLMs, search engines, web knowledge extraction
BookCorpus	English	~1 GB	Narrative pretraining, fiction modeling, long-form dialog

Tokenization and Vocabulary Creation

Objective: Converting text to numerical tokens the model can process

Purpose: Converts human-readable input into model-understandable format

Tokenizer Types: BPE, WordPiece, Unigram

- **BPE (Byte Pair Encoding):** BPE is a compression-inspired tokenization technique. It starts with characters and iteratively merges the most frequent adjacent pairs into new "subword tokens".
  - Example: BPE splits "playing" into "play" + "ing"

Initialize Vocabulary:
["low", "lower", "new", "newest", "widest"]
We first split each word into characters and add a special end-of-word token </w>
"l o w </w>"
"l o w e r </w>"
"n e w </w>"
"n e w e s t </w>"
"w i d e s t </w>“
Treat each word as a list of tokens (starting from characters) and look for adjacent token pairs.

Symbol Pair	Count
(l, o)	2
(o, w)	2
(w,)	1
(o, w)	1
(w, e)	1
(e, r)	1
(n, e)	2
(e, w)	2
(w,)	1
(e, s)	2
(s, t)	2
(t,)	2
(w, i)	1
(i, d)	1
(d, e)	1

Merge the Most Frequent Pair
In the table above, suppose the most frequent pair is (e, s) with a frequency of 2.
We then merge e and s into es in all instances:
"n e w e s t </w>" becomes "n e w es t </w>“
Then repeat:
- Recompute symbol pairs
- Find the new most frequent pair
- Merge it
- Continue until reaching desired vocabulary size or number of merges

Tokenization and Vocabulary Creation

WordPiece

WordPiece is similar to BPE but with a key difference: it selects the merge that gives the highest likelihood increase (not just frequency). It was introduced in Google's BERT.

How it works:

- Start with a base vocabulary (e.g., characters).
- Greedily add new subwords that improve the model likelihood (based on a language model).
- Use "##" to denote subword continuation.

For example "unaffordable" tokenized as ["un", "##aff", "##ord", "##able"]

Step 1: Initialize Vocabulary

- A vocabulary of known tokens, typically includes common words and subwords.
- Subwords are usually marked with ## when they are not at the start of a word.

["un", "afford", "##able", "##a", "##ff", "##or", "##d", "##able"]

Step 2: Greedy Matching – Left to Right

We try to greedily match the longest token from the vocabulary that matches a part of the word.

- Check from the start: "un" → found in vocab

Remaining: "affordable"

- Check next longest: "afford" → Found

Remaining: "able"

- "able" → not in vocab, but "##able" is

["un", "afford", "##able"]

Tokenization and Vocabulary Creation

- **Vocabulary Size**: 10k–50k tokens
  - Larger vocab = less sequence length but more memory use
- **Pipeline**: Convert "Hello world" → [7592, 6213]
  - Each token maps to an index in the vocabulary
- **Trade-off**: Larger vocab → better coverage, but slower inference

```
from tokenizers import ByteLevelBPETokenizer
tokenizer = ByteLevelBPETokenizer()
tokenizer.train(files=["data.txt"], vocab_size=10000)
```

Model Architecture and Configuration

Objective: Structure of the neural network that processes tokens

Purpose: Enables learning from sequences of words

Backbone: Use decoder-only for GPT (e.g., text generation)

- **Example:** GPT-2 uses only decoder blocks for left-to-right learning

```
from transformers import GPT2Config
config = GPT2Config(n_layer=2, n_head=4, n_embd=128)
```

Defines a small transformer model with 2 layers, 4 attention heads, and 128-dim embeddings

Positional Encoding: Injects information about word position

- Transformers are position-agnostic by default

Distributed Training: Train on multiple GPUs with PyTorch Lightning or DeepSpeed

- Speeds up training and handles large models

Implementation Details

Step-1: Load dataset

We use the "ag_news" dataset which consists of news article titles and descriptions.

```
# Download CSVs (for Colab use)
!wget -q https://raw.githubusercontent.com/mhjabreel/CharCnn_Keras/master/data/ag_news_csv/train.csv
!wget -q https://raw.githubusercontent.com/mhjabreel/CharCnn_Keras/master/data/ag_news_csv/test.csv

# Load CSVs into DataFrames
train_df = pd.read_csv("train.csv", header=None, names=["Class Index", "Title", "Description"])
test_df = pd.read_csv("test.csv", header=None, names=["Class Index", "Title", "Description"]) # Load the
test data

# Combine title and description for training text
train_df["text"] = train_df["Title"] + ". " + train_df["Description"]
train_dataset = Dataset.from_pandas(train_df[["text"]].head(100))  # limit to 100 samples for demo

# Combine title and description for test text and create eval dataset
test_df["text"] = test_df["Title"] + ". " + test_df["Description"]
eval_dataset = Dataset.from_pandas(test_df[["text"]].head(50)) # limit test data for evaluation
```

Implementation Details

Step-2: Tokenization (convert the words into tokens)

We tokenize the text using a pre-trained tokenizer.

- In this example, we use the GPT-2.
- Tokenization involves splitting text into tokens and padding/truncating them to a fixed length for batch processing.

```
# Step 2: Tokenization
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token

def tokenize_function(example):
    result = tokenizer(example["text"], truncation=True, padding="max_length", max_length=64)
    result["labels"] = result["input_ids"].copy()
    return result

tokenized_train_dataset = train_dataset.map(tokenize_function, batched=True)
tokenized_eval_dataset = eval_dataset.map(tokenize_function, batched=True)
```

Implementation Details

Step-3: Configure LLM Parameters

We define the model architecture and training arguments.

- We use the distilGPT2 model for demonstration as it is small and fast to train.
- TrainingArguments specify hyperparameters such as batch size, number of epochs, logging settings, and save strategy.

```
from transformers import GPT2Config, GPT2LMHeadModel

config = GPT2Config(
    vocab_size=tokenizer.vocab_size,  # Use vocabulary size from the tokenizer
    n_positions=64,                   # Maximum sequence length the model can handle
    n_ctx=64,                         # Context size (same as n_positions)
    n_embd=128,                       # Size of token embeddings and hidden states
    n_layer=4,                        # Number of transformer blocks (depth of the model)
    n_head=4,                         # Number of attention heads
    pad_token_id=tokenizer.pad_token_id  # Define padding token to avoid mismatch
)
```

Implementation Details

Step-4: Train the LLM

We now initialize the Trainer object with the model, tokenizer, training arguments, and tokenized dataset.

- Trainer handles the training loop internally.
- We then call the .train() method to start training.

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir="./gpt2_scratch",       # Directory to save model checkpoints and final model
    evaluation_strategy="epoch",       # Evaluate the model after each epoch
    per_device_train_batch_size=4,     # Batch size per device (GPU/CPU)
    num_train_epochs=10,               # Number of training epochs
    logging_dir="./logs",              # Directory to store training logs
    logging_steps=10,                  # Log metrics every 10 steps
    save_steps=20,                     # Save model checkpoint every 20 steps
    save_total_limit=1,                # Retain only the most recent checkpoint
    fp16=False                         # Use mixed precision (set True if supported by GPU)
)
trainer = Trainer(
    model=model,                                 # GPT-2 model initialized from scratch
    args=training_args,                          # TrainingArguments that specify epochs, logging, batch size, etc.
    train_dataset=tokenized_train_dataset,       # Tokenized training data
    eval_dataset=tokenized_eval_dataset,         # Tokenized evaluation data (optional but useful for monitoring)
    tokenizer=tokenizer                          # Tokenizer used for encoding/decoding text
)
trainer.train()
print("✅ Training complete.")
```

Implementation Details

Step-5: Inference

Now we use the trained model to generate text. We provide a prompt and let the model predict the continuation of the text.

```
def generate_text(prompt):
    inputs = tokenizer(prompt, return_tensors="pt")
    input_ids = inputs["input_ids"]
    attention_mask = inputs["attention_mask"]
    # Get the device of the model
    device = tiny_model.device
    # Move input tensors to the model's device
    input_ids = input_ids.to(device)
    attention_mask = attention_mask.to(device)
    outputs = tiny_model.generate(
        input_ids=input_ids,
        attention_mask=attention_mask,
        max_new_tokens=50,
        num_return_sequences=1,
        do_sample=True,
        top_k=50,
        top_p=0.95,
        temperature=0.7
    )
    return tokenizer.decode(outputs[0], skip_special_tokens=True)
prompt = "Breaking news:"
generated_text = generate_text(prompt)
print("\nGenerated Text:\n", generated_text)
```

```
Generated Text: Breaking news: are with the all-
(Reuters). Reuters - (Reuters).
```

Pretraining Objectives

Objective: Tasks the model learns to solve during pretraining

Purpose: Teaches the model to understand and generate language

CLM (Causal Language Modeling): Predict the next word in sequence

- Example: Input: "The cat sat on the" → Output: "mat“

MLM (Masked Language Modeling): Predict missing words

- Example: Input: "The [MASK] sat on the mat" → Output: "cat“

```
from transformers import GPT2LMHeadModel
model = GPT2LMHeadModel(config)
```

Loads a GPT-style model with language modeling head

Multi-task Learning: Train on QA, summarization, etc.

  - Helps models generalize to many tasks

Metric: Perplexity = how uncertain the model is. Lower is better.

Implementation Details [MLM]

Step-1: Load dataset

We use the "ag_news" dataset which consists of news article titles and descriptions.

```
!wget -q https://raw.githubusercontent.com/mhjabreel/CharCnn_Keras/master/data/ag_news_csv/train.csv
import pandas as pd

df = pd.read_csv("train.csv", header=None, names=["Class Index", "Title", "Description"])
df["text"] = df["Title"] + " " + df["Description"]
texts = df["text"].tolist()

texts= texts [0:5000] # taking only 2000 samples
# Display sample
print("Sample Text:\n", texts[0])
```

Implementation Details

Step-2: Tokenization (convert the words into tokens)

We tokenize the text using a pre-trained tokenizer.

- We'll use the BERT tokenizer and apply random masking (MLM-style) using Hugging Face's built-in DataCollatorForLanguageModeling.

```
from transformers import AutoTokenizer
from datasets import Dataset

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

dataset = Dataset.from_dict({"text": texts})
def tokenize_function(examples):
    return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=64)

tokenized_dataset = dataset.map(tokenize_function, batched=True, remove_columns=["text"])
```

Implementation Details

Step-2: Tokenization (convert the words into tokens)

We tokenize the text using a pre-trained tokenizer.

- We'll use the BERT tokenizer and apply random masking (MLM-style) using Hugging Face's built-in DataCollatorForLanguageModeling.

```
from transformers import DataCollatorForLanguageModeling

data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=True,
    mlm_probability=0.15  # 15% of tokens will be replaced with [MASK]
)
```

```
Example 2: Original: carlyle looks toward commercial aerospace ( reuters ) reuters - private investment firm carlyle group,
\ which has a reputation for making well - timed and occasionally \ controversial plays in the defense industry, has
quietly placed \ its bets on another part of the market. Masked : [MASK]le [MASK] toward commercial aerospace ( reuters )
[MASK] - private investment firm carlyle group, [MASK] which [MASK] a reputation [MASK] makingiol - timed [MASK]
occasionally \ controversial plays in the defense industry, has quietly placed \ its bets on another part [MASK] the
market.

Example 3: Original: oil and economy cloud stocks ' outlook ( reuters ) reuters - soaring crude prices plus worries \ abou
the economy and the outlook for earnings are expected to \ hang over the stock market next week during the depth of the \
summer doldrums. Masked : oil and economy cloud stocks ' outlook ( reuters ) reuters [MASK] soaring sarcastic prices plus
worries \ about the [MASK] [MASK] the outlook for earnings [MASK] expected to \ hang over the stock market next week durin
the depth of [MASK] eats summer doldrums.
```

Implementation Details

Step-3: Configure LLM Parameters

We'll fine-tune bert-base-uncased using the prepared dataset and collator.

```
from transformers import AutoModelForMaskedLM, TrainingArguments, Trainer

model = AutoModelForMaskedLM.from_pretrained("bert-base-uncased")

training_args = TrainingArguments(
    output_dir="./bert-mlm",
    # evaluation_strategy="no", # Removed this argument as it caused a TypeError
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    num_train_epochs=1,
    weight_decay=0.01,
    logging_steps=100,
    save_steps=500,
)
```

Implementation Details

Step-4: Train the LLM

We now initialize the Trainer object with the model, tokenizer, training arguments, and tokenized dataset.

- Trainer handles the training loop internally.
- We then call the .train() method to start training.

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset,
    tokenizer=tokenizer,
    data_collator=data_collator,
)

trainer.train()
```

Implementation Details

Step-5: Inference

Now we use the trained model to generate text. We provide a prompt and let the model predict the continuation of the text.

```python
from transformers import pipeline

fill_mask = pipeline("fill-mask", model=model, tokenizer=tokenizer)

# Example: Predict the masked word
sentence = "The [MASK] sat on the mat."
results = fill_mask(sentence)

print("Predictions for [MASK]:")
for r in results:
    print(f"{r['token_str']:>10s} | score: {r['score']:.4f}")
```

```
Predictions for [MASK]:
▪ man | score: 0.0851
▪ girl | score: 0.0399
▪ boy | score: 0.0367
▪ dog | score: 0.0341
▪ woman | score: 0.0209
```

Utilize LLMs for the Downstream Tasks

Step-1: Load dataset

Movie Review dataset having two classes “Positive” and “Negative”

```
from datasets import load_dataset

# Load IMDB dataset normally (non-streaming)
dataset = load_dataset("imdb", split="train")
dataset = dataset.train_test_split(test_size=0.1)
```

Utilize LLMs for the Downstream Tasks

Step-2: Tokenization (convert the words into tokens)

```
from transformers import AutoTokenizer
import torch

tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token

# Define the format_labels function
def format_labels(example):
    # Tokenize the full input including the prompt and the label
    full_sequence = f"Review: {example['text']}\\nSentiment: {example['label']}"
    tokenized_input = tokenizer(
        full_sequence,
        padding="max_length",
        truncation=True,
        max_length=128,
        return_attention_mask=True,
        return_token_type_ids=False # GPT-2 doesn't use token_type_ids
    )

    # Tokenize just the prompt part to find its length
    prompt_sequence = f"Review: {example['text']}\\nSentiment:"
    # Tokenize the prompt without padding/truncation to get the accurate prompt length in tokens
    tokenized_prompt = tokenizer(
        prompt_sequence,
        add_special_tokens=False # Exclude special tokens for accurate length of the prompt text
    )
```

Utilize LLMs for the Downstream Tasks

Step-3: Configure LLM Parameters

```
from transformers import AutoModelForCausalLM, TrainingArguments, Trainer

model = AutoModelForCausalLM.from_pretrained("gpt2")
model.resize_token_embeddings(len(tokenizer))

training_args = TrainingArguments(
    output_dir="./results",
    # evaluation_strategy="epoch",
    learning_rate=2e-5,
    weight_decay=0.01,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    num_train_epochs=2,
    logging_dir="./logs",
    push_to_hub=False
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["test"],
    tokenizer=tokenizer
)
```

Utilize LLMs for the Downstream Tasks

Step-4: Train the LLM

We now initialize the Trainer object with the model, tokenizer, training arguments, and tokenized dataset.

- Trainer handles the training loop internally.
- We then call the .train() method to start training.

```
trainer.train()
```

Utilize LLMs for the Downstream Tasks

Step-5: Inference

Now we use the trained model to generate text. We provide a prompt and let the model predict the label of the text.

```
input_text = "Review: The movie was dull and boring.\nSentiment:"
inputs = tokenizer(input_text, return_tensors="pt").to(model.device)
outputs = model.generate(**inputs, max_length=50)
print(tokenizer.decode(outputs[0]))
```

```
Review: The movie was great and a good
watch. Sentiment: **Positive**
```

Thanks for your time