Leveraging Large Language Models in Biomedical Research

Minerva Scientific Computing Environment

https://labs.icahn.mssm.edu/minervalab

S M Shamimul Hasan, Ph.D.
The Minerva HPC Team

March 12, 2026

Icahn
School of
Medicine at
Mount
Sinai

Outline

- Introduction to Large Language Models (LLMs)
- Transformer Architecture
- Popular LLM Models
- Applications of LLMs in Healthcare
- Fine-Tuning LLMs
- Hands-On Example

Introduction to Large Language Models (LLMs)

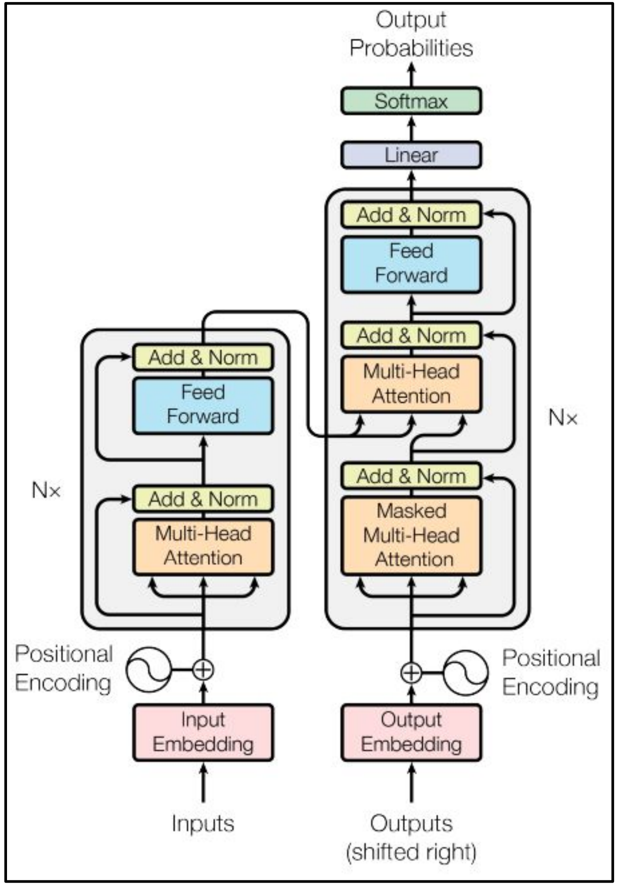
- What are LLMs?
  - LLMs are advanced deep learning models that can understand, generate, and manipulate human language
  - These models are trained on massive datasets containing billions of words, allowing them to learn complex language patterns, grammar, and semantics
- Core Features of LLMs
  - **Massive Scale:** LLMs are characterized by their enormous size, often containing billions or even trillions of parameters
  - **Contextual Understanding:** Unlike earlier models, LLMs use transformers, enabling them to understand context, relationships between words, and the broader meaning of sentences
  - **Transfer Learning:** Pre-trained on diverse datasets and fine-tuned for specific tasks, reducing the need for task-specific training data

Evolution of LLMs

- **Pre-2010s:** Early NLP models relied on rule-based systems and statistical approaches like n-grams and Hidden Markov Models (HMMs). These models lacked understanding of context and were limited in scalability
- **2013:** Introduction of Word2Vec, which used neural networks to generate word embeddings. This was a major shift from purely statistical methods, enabling words to be represented in vector space based on their contextual usage
- **2015-2017:** Rise of Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, which could capture sequence-based data. However, these models struggled with long-range dependencies and were computationally expensive

Evolution of LLMs

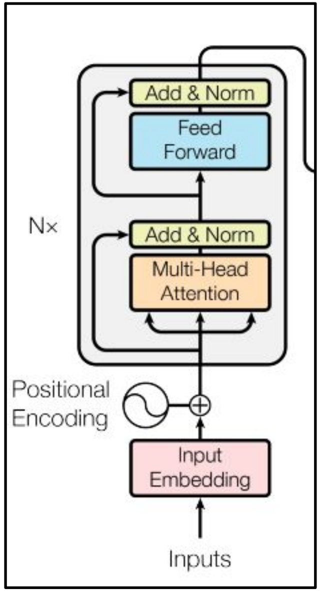
- **The Transformer Breakthrough (2017)**:
  - **Transformers**: Introduced by Vaswani et al. in the paper ***"Attention is All You Need,"*** the transformer model replaced RNNs and LSTMs with the **self-attention mechanism**, revolutionizing the way NLP models processed information
  - **Advantages of Transformers**: Unlike RNNs, transformers process words in parallel, making them faster and more scalable. This parallelization enabled training on much larger datasets
  - **Attention Mechanism**: The self-attention mechanism allows transformers to focus on different parts of the input text, improving their ability to understand context and relationships within text



https://arxiv.org/abs/1706.03762

Encoder Component

- **Input Embedding:**
  - Converts the input text into a numerical form that the model can understand
  - Example: If your sentence is "The cat sat on the mat", each word would be transformed into a different numerical vector, like [0.2, 0.5, ...] for "The", [0.8, 0.1, ...] for "cat", and so on
- **Positional Encoding:**
  - Adds information about the position of each word in the sentence
  - The word "The" might be the first word in the sentence, so it gets a certain positional encoding, and "cat" might be the second, so it gets another. This encoding helps the model understand the word sequence
- **Multi-Head Attention:**
  - This is where the model pays attention to different words in the sentence simultaneously. It helps the model understand relationships between words
  - Example: While processing "sat," the model might look at both "cat" (who sat?) and "mat" (where?). Another head might focus on the relationship between "The" and "cat”



https://arxiv.org/abs/1706.03762

Encoder Component

Add & Norm:

Combines the original input (from the embedding) with the result of the attention mechanism. Then, it normalizes the data, making sure the values are balanced and the training process is smooth

Example: The model combines the information from "sat" and the relationships it found (like how it relates to "cat" and "mat") and then adjusts the values to be on the same scale, which helps the model learn faster

Feed Forward:

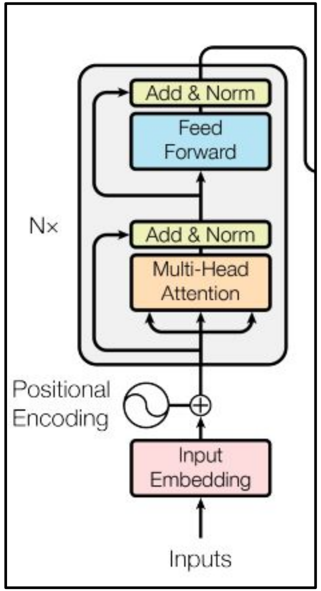
It’s like a standard neural network layer that refines the understanding of the relationships between words

Example: After focusing on the relationships between words like "sat", "cat", and "mat," the model now processes this information more deeply, understanding that the "cat" is performing the action "sat" and the "mat" is where it happened

Add & Norm

Like before, the model combines the original data with the output from the Feed Forward layer and normalizes it again

Example: The model continues refining its understanding of the sentence, ensuring the data stays balanced and ready for the next layers



https://arxiv.org/abs/1706.03762

Decoder Component

Decoder:

The decoder works similarly to the encoder but has a specific task: generating the output, one word (token) at a time. It receives the previous word generated (e.g., the first word of the translation) as input

Example: If the encoder processed "The cat sat on the mat," and the decoder is translating it into French, it might have already produced "Le chat" (The cat). Now, it uses this input to generate the next word in the sentence

Masked Multi-Head Attention

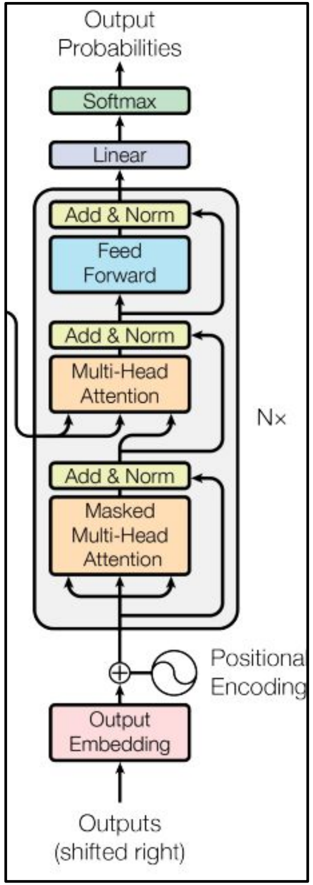
The model can only attend to previous words in the sentence, not future ones (hence the "masked"). This prevents the decoder from "cheating" and seeing future words before generating them

Example: If it has already generated "Le chat", it cannot look ahead to the next words. Instead, it uses the previous context to decide the next word (maybe "s'assit" for "sat").

Add & Norm:

Combines the results from the attention mechanism and normalizes the data again, keeping the learning process smooth

Example: Combines the learned relationships from "Le chat" and balances the information before continuing



https://arxiv.org/abs/1706.03762

Decoder Component

Multi-Head Attention:

The decoder looks at the output of the encoder. This is where the decoder learns from the original input sentence and adjusts its output accordingly

Example: The decoder might look at the encoder’s understanding of "The cat sat on the mat" to decide the next word in the translation

Add & Norm:

The model combines the input with the attention results and normalizes them

Example: The model balances all the learned relationships between words, refining the generation process

Feed Forward:

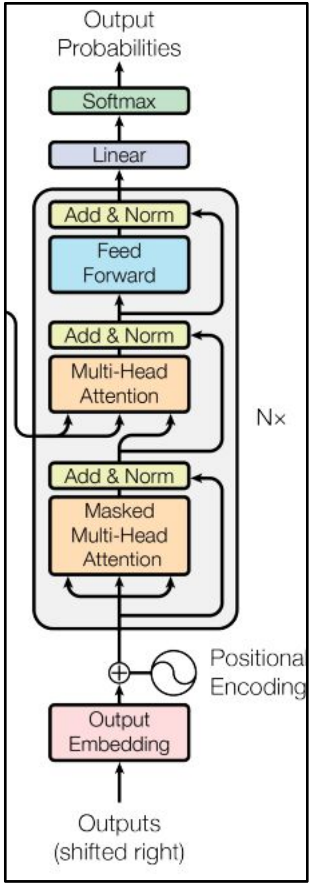
Like the encoder, the decoder has its own Feed Forward layer to process the combined information. This helps in refining the output word by word

Example: Based on what it learned from "Le chat", the decoder is now prepared to generate the next word in the translated sentence (e.g., "s'assit")

Add & Norm:

Another step of combining the learned relationships and balancing the values, preparing the data for the final prediction

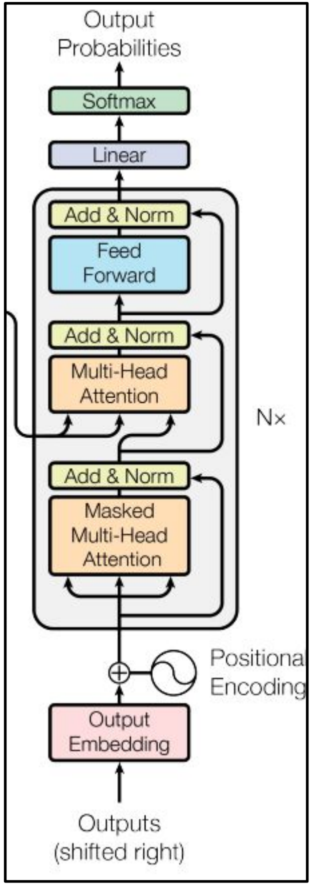
Example: The model further refines the current state of the translation before making the final word prediction



https://arxiv.org/abs/1706.03762

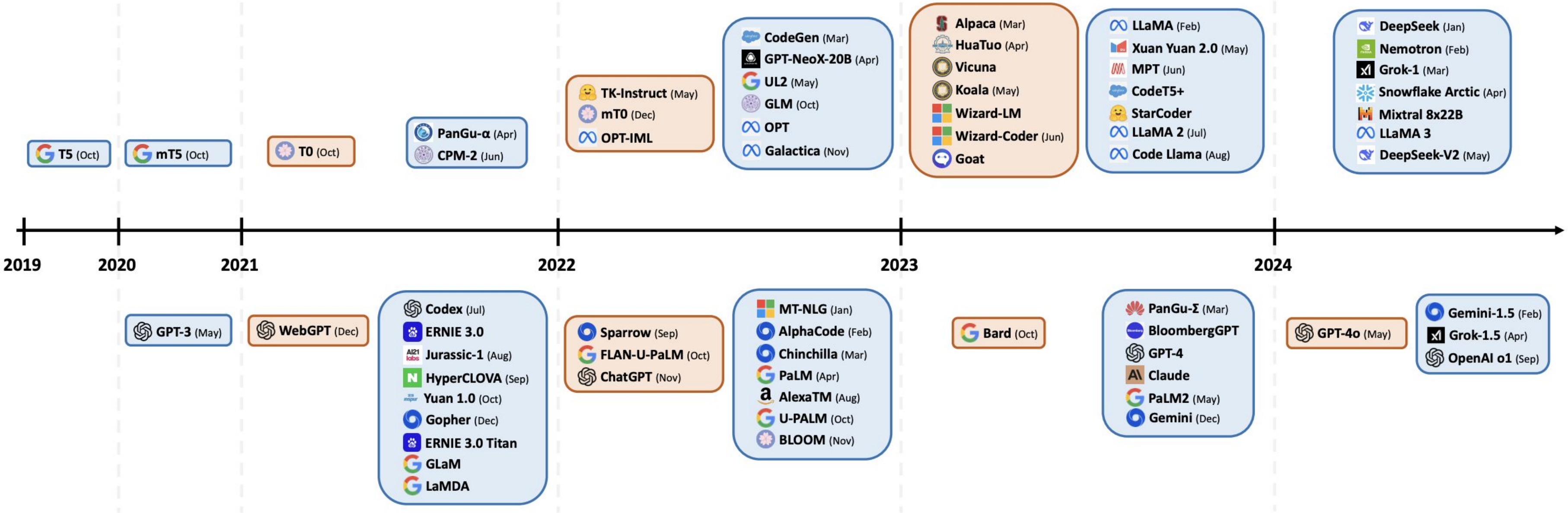
Decoder Component

- **Linear Layer:**
  - Converts the processed data from the decoder into a set of raw scores (called logits) for each possible word in the vocabulary
  - Example: If the model is translating, the raw scores for possible next words like "s'assit" (sat) or "reposait" (rested) are calculated
- **Softmax Layer:**
  - Turns those raw scores into probabilities, indicating which word is the most likely next word in the sentence
  - Example: "s'assit" might get the highest probability (e.g., 0.9), so the model selects that as the next word in the translation
- **Final Output:**
  - The model continues this process of predicting word by word until it completes the sentence
  - Example: After predicting "Le chat s'assit", it continues generating the rest of the translation



https://arxiv.org/abs/1706.03762

LLM Releases



2019 2020 2021 2022 2023 2024

T5 (Oct)

mT5 (Oct)

T0 (Oct)

PanGu-α (Apr)
CPM-2 (Jun)

TK-Instruct (May)
mT0 (Dec)
OPT-IML

CodeGen (Mar)
GPT-NeoX-20B (Apr)
UL2 (May)
GLM (Oct)
OPT
Galactica (Nov)

Alpaca (Mar)
HuaTuo (Apr)
Vicuna
Koala (May)
Wizard-LM
Wizard-Coder (Jun)
Goat

LLaMA (Feb)
Xuan Yuan 2.0 (May)
MPT (Jun)
CodeT5+
StarCoder
LLaMA 2 (Jul)
Code Llama (Aug)

DeepSeek (Jan)
Nemotron (Feb)
Grok-1 (Mar)
Snowflake Arctic (Apr)
Mixtral 8x22B
LLaMA 3
DeepSeek-V2 (May)

GPT-3 (May)

WebGPT (Dec)

Codex (Jul)
ERNIE 3.0
Jurassic-1 (Aug)
HyperCLOVA (Sep)
Yuan 1.0 (Oct)
Gopher (Dec)
ERNIE 3.0 Titan
GLaM
LaMDA

Sparrow (Sep)
FLAN-U-PaLM (Oct)
ChatGPT (Nov)

MT-NLG (Jan)
AlphaCode (Feb)
Chinchilla (Mar)
PaLM (Apr)
AlexaTM (Aug)
U-PALM (Oct)
BLOOM (Nov)

Bard (Oct)

PanGu-Σ (Mar)
BloombergGPT
GPT-4
Claude
PaLM2 (May)
Gemini (Dec)

GPT-4o (May)

Gemini-1.5 (Feb)
Grok-1.5 (Apr)
OpenAI o1 (Sep)

Blue cards represent **pre-trained** models and orange cards correspond to **instruction-tuned** models
Models on the upper half signify open-source availability, whereas those on the bottom are closed-source

https://arxiv.org/pdf/2307.06435

Popular LLM Models

Model	Organization	Parameters
GPT 4	OpenAI	175+ billion
BERT	Google	110 – 340 million
PaLM 2	Google	340 billion
LLaMA 3	Meta	70 billion
T5	Google	220 million – 11 billion
Claude	Anthropic	100+ billion
Falcon	Technology Innovation Institute (TII)	180 billion

Why GPUs are Essential for LLMs?

- **Parallel Processing**:
  - GPUs are designed to handle parallel computations, making them ideal for training large models like LLMs
  - LLMs involve matrix operations that benefit significantly from the parallel nature of GPUs
- **Memory Requirements**:
  - Training LLMs requires handling large datasets and models with billions of parameters, necessitating high GPU memory
  - GPUs like the B200, H100 and A100 offer sufficient memory bandwidth and capacity to handle these models efficiently
- **Real-Time Inference**:
  - Inference is the process of using a trained model to make predictions
  - GPUs accelerate inference by reducing the time taken to process each input, enabling real-time applications like chatbots and virtual assistants

Overview of B200, H100 and A100 GPUs

- **Architecture**:
  - CUDA Cores: Responsible for general-purpose computation, critical for large-scale AI and HPC tasks
  - Tensor Cores: Specialized for deep learning operations, particularly matrix multiplications
    - B200: Next-generation Tensor Cores with FP4, FP8, and FP16 support, optimized for large-scale AI and LLM training
    - H100: Improved Tensor Cores with FP8 support for faster AI training
    - A100: Tensor Cores optimized for mixed precision (FP16) training and inference
- **Memory Capacity**:
  - B200: 192 GB
  - H100: 80 GB
  - A100: 40 GB or 80 GB
- **Performance:**
  - B200: Up to ~20 PFLOPS AI performance
  - H100: Up to ~989 TFLOPS AI performance
  - A100: Up to 312 TFLOPS AI performance

How LLMs Can Help Healthcare?

- **Clinical Trial Optimization**
  - **Patient Matching:** LLMs can analyze patient data, medical histories, and genetic information to match eligible patients to clinical trials quickly and accurately, reducing the time and cost associated with recruitment
- **Improved Patient Care**
  - **Personalized Treatment Plans:** LLMs can recommend treatment plans tailored to patient data, integrating research, guidelines, and real-time monitoring
  - **Drug Interaction Warnings:** LLMs alert providers to potential drug interactions by cross-referencing a patient's medication history with interaction databases
- **Medical Research Advancement**
  - **Literature Review & Data Synthesis:** LLMs assist researchers by rapidly synthesizing new findings from a vast array of biomedical literature, helping identify new drug targets or potential therapies

Training LLMs

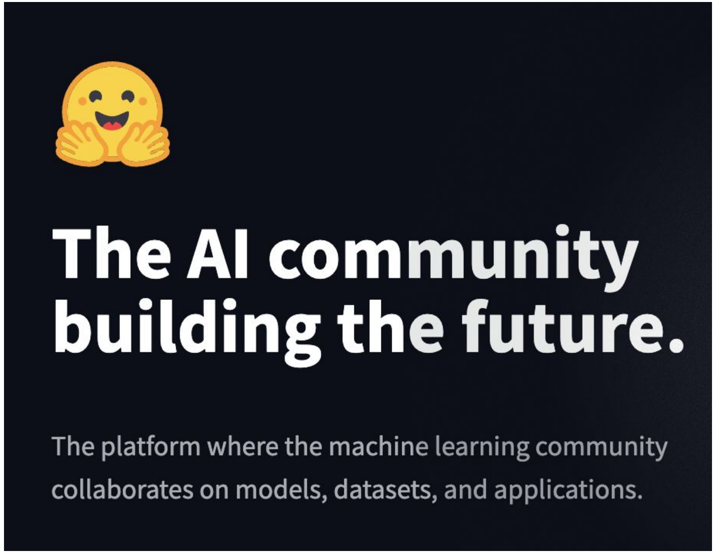
- **Pre-Training vs. Fine-Tuning**:
  - **Pre-Training**: Training on a large corpus of text to learn general language understanding
  - **Fine-Tuning**: Tailoring the pre-trained model to a specific task, such as sentiment analysis
- **Data Requirements**:
  - LLMs require vast and diverse datasets for effective learning
  - **Challenges**: Managing data quality, balancing different types of text (e.g., clinical vs. general language)
  - Example: PubMed for biomedical text, Wikipedia for general knowledge

Zero-Shot, One-Shot, and Few-Shot Learning

- **Zero-Shot Learning**:
  - The model can handle a task or classify data without any prior examples. It relies on general knowledge learned during pre-training to make predictions
- **One-Shot Learning**:
  - The model can perform a task or recognize a class after being trained on just one example. It is especially useful in scenarios with limited data
- **Few-Shot Learning**:
  - The model is trained with only a few examples per class and still achieves reasonable accuracy

Hugging Face

- A leading platform for open-source natural language processing (NLP) models and tools
- Provides easy access to pretrained models and transformer architectures like BERT, GPT, and T5
- Hugging Face Model Hub hosts thousands of models for a wide range of tasks: translation, summarization, question answering, etc.
- Transformers in Healthcare:
  - **BioBERT** and **ClinicalBERT** models, pretrained on biomedical and clinical data, are available via Hugging Face.
  - **MedGPT** and **COVID-Twitter-BERT (CT-BERT)**: Domain-specific transformers for healthcare challenges like COVID-19 tracking and drug discovery



The AI community building the future.

The platform where the machine learning community collaborates on models, datasets, and applications.

https://huggingface.co/

Tokenization

- **Word, Sentence, and Subword Tokenization**:
  - **Word Tokenization**: Splits text into individual words. Example: “Natural Language Processing” → [“Natural”, “Language”, “Processing”]
  - **Sentence Tokenization**: Splits text into sentences. Example: “I love NLP. It’s fascinating.” → [“I love NLP.”, “It’s fascinating.”]
  - **Subword Tokenization**: Breaks down words into smaller units. Useful for handling out-of-vocabulary words. Example: “unhappiness” → [“un”, “happiness”]
- **Tokenization for LLMs**:
  - Tokenization affects model performance by determining how text is represented as input
  - Subword tokenization (e.g., Byte-Pair Encoding) is often used in LLMs to balance vocabulary size and input sequence length
- **Popular Tokenizers**:
  - **BERT Tokenizer**: Uses WordPiece tokenization, suitable for understanding context in text
  - **GPT Tokenizer**: Based on Byte-Pair Encoding, optimized for generating text

Data Preprocessing for LLMs

- **Tokenization with Transformers**: Use pre-trained tokenizers from the Transformers library to match the model’s requirements

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
tokens = tokenizer(text, return_tensors='pt', padding=True, truncation=True,
max_length=128)
```

- **Efficient Tokenization:** Tokenizing large datasets can be a bottleneck; use batching and parallel processing on GPUs

```
def batch_tokenize(texts, tokenizer, batch_size=32):
    tokenized_batches = []
    for i in range(0, len(texts), batch_size):
        batch = texts[i:i + batch_size]
        tokenized_batch = tokenizer(batch, return_tensors='pt', padding=True,
truncation=True, max_length=128).to('cuda')
        tokenized_batches.append(tokenized_batch)
return tokenized_batches
```

Why Fine-Tune Pre-Trained Models in Healthcare?

- **Leverage Medical Knowledge:** Pre-trained models (like BERT, GPT) learn general patterns. Fine-tuning adapts these models to understand specific medical data, like patient records and clinical trial results
- **Enhance Accuracy:** Fine-tuning improves performance in specialized healthcare tasks such as disease diagnosis, medical image analysis, and treatment recommendation
- **Efficient with Limited Data:** Healthcare data can be scarce or sensitive. Fine-tuning allows effective model training with small datasets, maintaining privacy and accuracy
- **Faster Implementation:** By fine-tuning, AI models can be quickly adapted for clinical use, providing real-time decision support in patient care

Fine-Tuning Pre-Trained Model

```
# Import necessary libraries
from transformers import BertForSequenceClassification,
BertTokenizer, AdamW, get_linear_schedule_with_warmup
from torch.utils.data import DataLoader, TensorDataset
import torch

# 1. Load the pre-trained BERT model for sequence
classification
# Setting num_labels=2 for binary classification
model =
BertForSequenceClassification.from_pretrained('bert-base-unca
sed', num_labels=2)

# Move the model to GPU for faster training
model.to('cuda')

# 2. Set up the BERT tokenizer for tokenizing input text
tokenizer =
BertTokenizer.from_pretrained('bert-base-uncased')
```

Fine-Tuning Pre-Trained Model

```
# train_texts and train_labels in raw text form:
train_texts = ["Sample text for training", "Another text"]
train_labels = torch.tensor([1, 0])

# Tokenize the training data (convert text into input IDs and attention masks)
train_encodings = tokenizer(train_texts, truncation=True, padding=True,
max_length=128, return_tensors="pt")

# Create TensorDataset with inputs and labels
train_dataset = TensorDataset(train_encodings['input_ids'],
train_encodings['attention_mask'], train_labels)

# 3. Set up the DataLoader to handle batches of data
# DataLoader will load the data in batches and shuffle it for training
train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
```

Fine-Tuning Pre-Trained Model

```
# 4. Set up the optimizer (AdamW is the recommended optimizer for BERT)
optimizer = AdamW(model.parameters(), lr=1e-5)

# 5. Set up a learning rate scheduler (optional, but often helpful for fine-tuning)
num_epochs = 3
total_steps = len(train_dataloader) * num_epochs

scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=total_steps
)

# 6. Set up the training loop
# Set model to training mode
model.train()
```

Fine-Tuning Pre-Trained Model

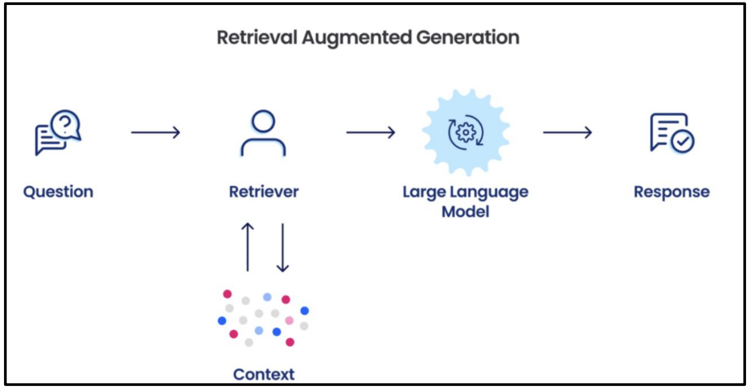
```
# Loop over the dataset for the specified number of epochs
for epoch in range(num_epochs):
    print(f"Epoch {epoch+1}/{num_epochs}")

    # Initialize total loss for this epoch
    total_loss = 0

    # Loop over each batch in the DataLoader
    for batch in train_dataloader:
        # Move input tensors (input_ids and attention_masks) and labels to GPU
        input_ids = batch[0].to('cuda')
        attention_masks = batch[1].to('cuda')
        labels = batch[2].to('cuda')

        # Clear previous gradients
        optimizer.zero_grad()

        # Perform forward pass: compute predictions and loss
        # The model will return loss directly since labels are provided
        outputs = model(input_ids, attention_mask=attention_masks, labels=labels)
```

Fine-Tuning Pre-Trained Model

```
        # Extract the loss
        loss = outputs.loss
        total_loss += loss.item()

        # Backpropagate the gradients
        loss.backward()

        # Update model parameters with optimizer
        optimizer.step()

        # Update learning rate (if using scheduler)
        scheduler.step()

    # Print the average loss for this epoch
    avg_loss = total_loss / len(train_dataloader)
    print(f"Average loss: {avg_loss:.4f}")

# 7. Save the fine-tuned model for later use
model.save_pretrained('./fine_tuned_bert_model')
tokenizer.save_pretrained('./fine_tuned_bert_tokenizer')

# The fine-tuned model can now be used for inference or further evaluation.
```

Retrieval Augmented Generation (RAG)

- RAG combines large language models (LLMs) with an external retrieval mechanism to provide more accurate, context-aware responses
- Benefits of RAG
  - Improved Accuracy: Ensures answers are based on the most relevant external information.
  - Domain-Specific Expertise: Custom knowledge bases make the model domain-adaptive, beneficial for biomedical or specialized fields



https://snorkel.ai/blog/which-is-better-retrieval-augmentation-rag-or-fine-tuning-both/

Retrieval Augmented Generation (RAG)

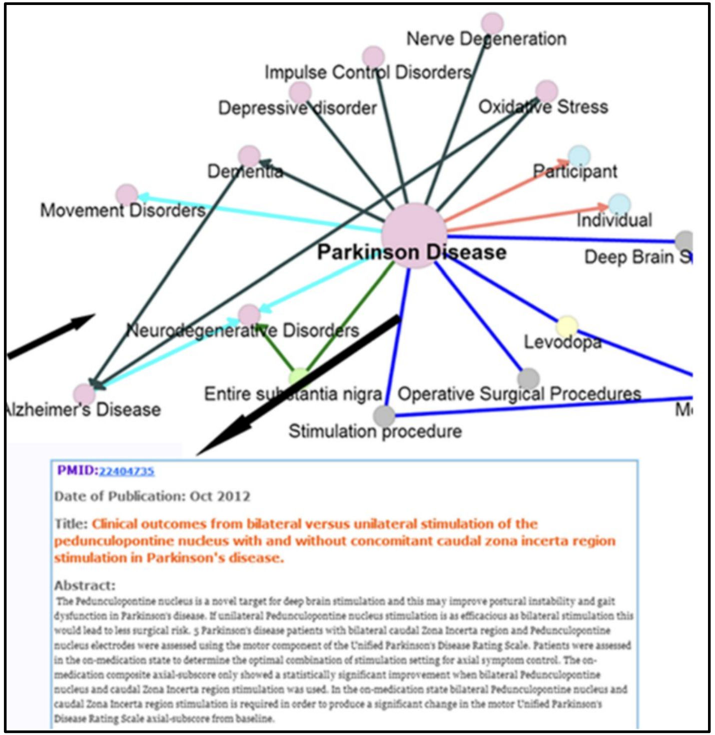
```
# Import the necessary libraries
from ollama import Client

# Step 1: Initialize the Ollama Client
ollama_client = Client(host='http://10.95.46.94:53441', headers={"Authorization": "Bearer
hasans10:..."})

# Step 2: Define a query for healthcare-related question
query = 'What are the main causes of cardiovascular disease?'

# Step 3: Implement a retrieval function to simulate fetching relevant healthcare documents
def retrieve_documents(query):
    # Here, we're simulating the retrieval process with some mock data relevant to the
healthcare domain.
    results = [
        "Document 1: Cardiovascular disease is caused by risk factors like high blood
pressure, high cholesterol, and smoking.",
        "Document 2: Other causes include diabetes, obesity, poor diet, and lack of
physical activity.",
        "Document 3: Family history and age also contribute to the likelihood of
cardiovascular disease."
    ]
    return results
```

Retrieval Augmented Generation (RAG)

```
# Step 4: Retrieve relevant documents based on the query
retrieved_docs = retrieve_documents(query)

# Step 5: Format the retrieved documents and the user's query into the chat
messages for the LLM
messages = [
    {'role': 'user', 'content': query},
    {'role': 'system', 'content': f"Retrieved information: {'
'.join(retrieved_docs)}"}
]

# Step 6: Make the Ollama LLM request with the query and retrieved context
stream = ollama_client.chat(
    model='tinyllama',
    messages=messages,
    stream=True,  # Stream the output for efficient handling
)

# Step 7: Stream and print the response from the LLM
for chunk in stream:
    print(chunk['message']['content'], end='', flush=True)
```

Semantic MEDLINE

- An application offering biomedical document retrieval, summarization, and visualization (38 million citations)
- Utilizes **SemRep**, a natural language processing tool, to identify semantic relationships in biomedical literature
- **Literature Summarization**: Helps researchers quickly understand relationships in medical literature
- RAG: Combines LLM’s language generation with real-time retrieval from the MEDLINE Knowledge Graph
- Knowledge augmentation



PMID:22404735

Date of Publication: Oct 2012

Title: Clinical outcomes from bilateral versus unilateral stimulation of the pedunculopontine nucleus with and without concomitant caudal zona incerta region stimulation in Parkinson's disease.

Abstract:
The Pedunculopontine nucleus is a novel target for deep brain stimulation and this may improve postural instability and gait dysfunction in Parkinson's disease. If unilateral Pedunculopontine nucleus stimulation is as efficacious as bilateral stimulation this would lead to less surgical risk. 5 Parkinson's disease patients with bilateral caudal Zona Incerta region and Pedunculopontine nucleus electrodes were assessed using the motor component of the Unified Parkinson's Disease Rating Scale. Patients were assessed in the on-medication state to determine the optimal combination of stimulation setting for axial symptom control. The on-medication composite axial-subscore only showed a statistically significant improvement when bilateral Pedunculopontine nucleus and caudal Zona Incerta region stimulation was used. In the on-medication state bilateral Pedunculopontine nucleus and caudal Zona Incerta region stimulation is required in order to produce a significant change in the motor Unified Parkinson's Disease Rating Scale axial-subscore from baseline.

https://link.springer.com/content/pdf/10.1186/1471-2105-14-182.pdf

Semantic MEDLINE:

https://lhncbc.nlm.nih.gov/ii/tools/SemRep_SemMedDB_SKR/SemMed.html

Unified Medical Language System (UMLS)

https://www.nlm.nih.gov/research/umls/index.html

Fine-Tuning vs. RAG

- **Fine-Tuning**:
  - Best for **long-term, slow-to-change** tasks like adapting an LLM to a specific domain
  - Focuses on training the model to incorporate domain-specific knowledge permanently
  - Effective in ensuring consistent style and response tone for internal policies
- **Retrieval-Augmented Generation (RAG)**:
  - Best for **dynamic, quick-to-change** tasks like responding to rapidly evolving information (e.g., real-time data, customer records)
  - Retrieves up-to-date context from external data sources at query time

Hyperparameter Tuning

- **Key Hyperparameters**
  - **Learning Rate**: Controls how much to change the model in response to the error at each update
  - **Batch Size**: Number of samples processed before the model is updated
  - **Epochs**: Number of complete passes through the training dataset
- **Impact on Performance**
  - **Learning Rate**: Too high can overshoot the minima; too low can slow down convergence
  - **Batch Size**: Larger sizes lead to more stable gradient estimates but require more memory
- **Example Code**

```
# Adjust learning rate and batch size
optimizer = AdamW(model.parameters(), lr=2e-5)
train_dataloader = DataLoader(train_dataset, batch_size=16)
```

Distributed Training on Multiple GPUs

- **Why Use Multiple GPUs?**
  - Distributes the computational load, reducing training time
  - Enables the training of larger models by splitting data across GPUs
- **Data Parallelism**
  - **Approach**: Split batches of data across multiple GPUs; each GPU computes gradients independently
  - **Code Example**

```
model = torch.nn.DataParallel(model)
model.to('cuda')
```

- **Model Parallelism**
  - **Approach**: Split the model itself across GPUs, useful for extremely large models.

```
model.model.encoder.layer[:6].to('cuda:0')
model.model.encoder.layer[6:].to('cuda:1')
```

Managing Large Datasets

- Use dataset loading strategies that minimize memory footprint, such as loading batches on the fly

```
from torch.utils.data import DataLoader, Dataset
class CustomDataset(Dataset):
    def __getitem__(self, index):
        # Load data on the fly
        return load_sample(index)

dataset = CustomDataset()
dataloader = DataLoader(dataset, batch_size=32, pin_memory=True)
```

Profiling and Debugging GPU Code: PyTorch Profiler

```
import torch
import torch.profiler as profiler
import json
# Example: A simple PyTorch model
class SimpleModel(torch.nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = torch.nn.Linear(10, 5)

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

# Initialize the model and move it to GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SimpleModel().to(device)

# Example input tensor
inputs = torch.randn(1, 10).to(device)

# Profile the model with CPU and CUDA activities
with profiler.profile(activities=[profiler.ProfilerActivity.CPU,
profiler.ProfilerActivity.CUDA]) as prof:
    # Run the model forward pass
    model(inputs)

# Export the profiling results
trace_filename = 'trace.json'
```

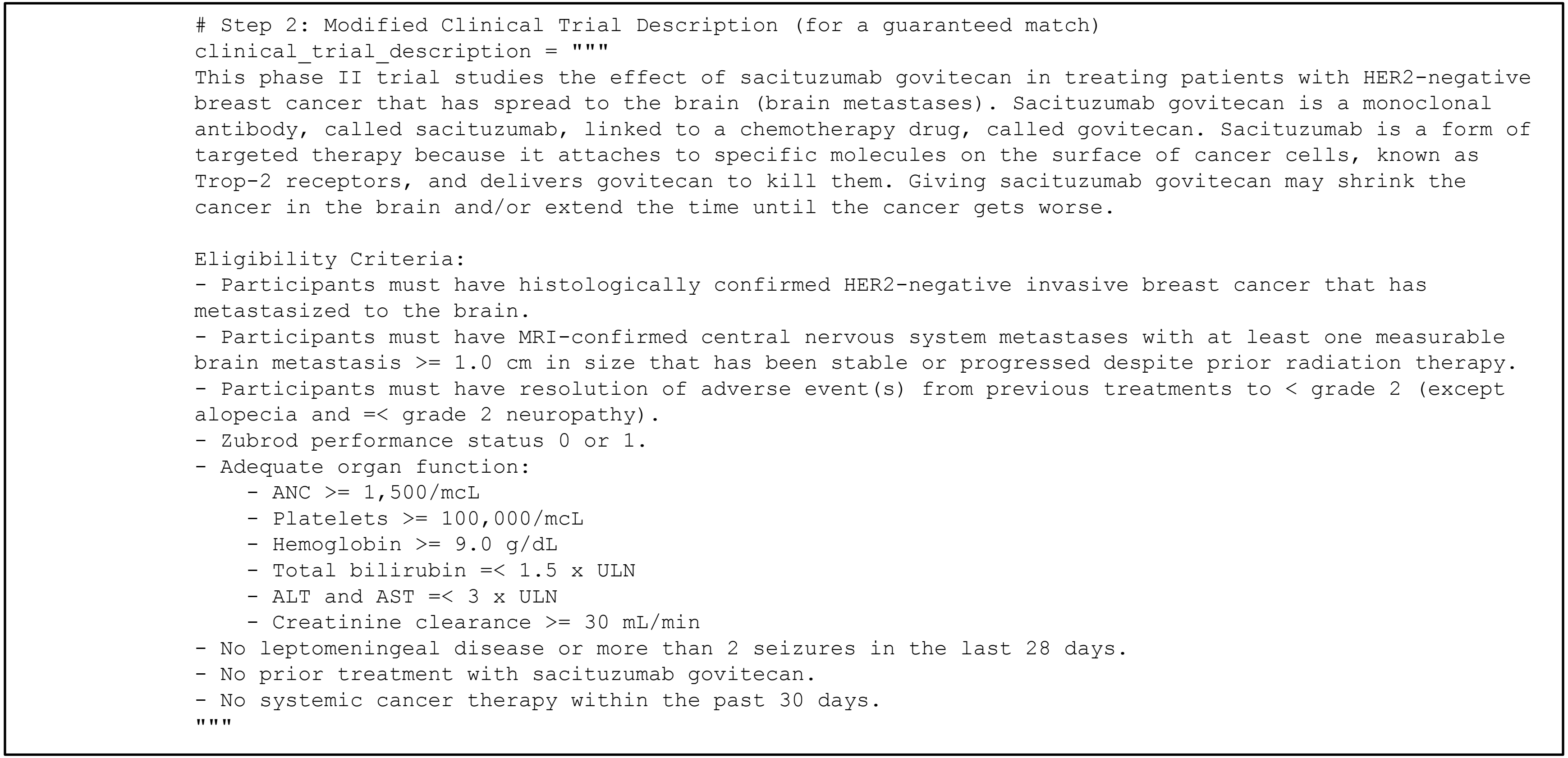
Hands-On Example: Ollama LLM Platform on Minerva

- https://labs.icahn.mssm.edu/minervalab/documentation/ollama/

Hands-On Example: Clinical Trial Matching Example

```
from ollama import Client

# Step 1: Modified Patient Clinical Notes (for a guaranteed match)
patient_notes = """
Patient Jane Smith is a 45-year-old female diagnosed with HER2-negative invasive
breast cancer in 2019.
In 2021, brain metastases were confirmed via MRI, showing a lesion of 1.5 cm in the
right frontal lobe.
The patient has undergone prior radiation therapy for the brain metastasis, and the
lesion has been stable over the last 6 months.
An MRI scan from April 2023 confirmed the lesion remains stable at 1.5 cm. The
patient has no history of seizures and is otherwise healthy.
Her performance status is Zubrod 1, and she has normal blood counts with an ANC of
1,600/mcL and platelets of 150,000/mcL.
Her hemoglobin is 10.0 g/dL.
Her creatinine clearance is 40 mL/min, and she has normal liver function tests
(bilirubin = 0.8 mg/dL, ALT = 20 U/L, and AST = 18 U/L).
The patient experienced mild neuropathy during her prior treatments but has no
significant lingering adverse events.
She has not experienced more than two seizures in the last 28 days.
She is currently being considered for second-line treatment options and has not
received any systemic cancer therapy in the past 30 days.
She has not been treated with sacituzumab govitecan before.
"""
```

Demo: Clinical Trial Matching Example



```
# Step 2: Modified Clinical Trial Description (for a guaranteed match)
clinical_trial_description = """
This phase II trial studies the effect of sacituzumab govitecan in treating patients with HER2-negative
breast cancer that has spread to the brain (brain metastases). Sacituzumab govitecan is a monoclonal
antibody, called sacituzumab, linked to a chemotherapy drug, called govitecan. Sacituzumab is a form of
targeted therapy because it attaches to specific molecules on the surface of cancer cells, known as
Trop-2 receptors, and delivers govitecan to kill them. Giving sacituzumab govitecan may shrink the
cancer in the brain and/or extend the time until the cancer gets worse.

Eligibility Criteria:
- Participants must have histologically confirmed HER2-negative invasive breast cancer that has
metastasized to the brain.
- Participants must have MRI-confirmed central nervous system metastases with at least one measurable
brain metastasis >= 1.0 cm in size that has been stable or progressed despite prior radiation therapy.
- Participants must have resolution of adverse event(s) from previous treatments to < grade 2 (except
alopecia and =< grade 2 neuropathy).
- Zubrod performance status 0 or 1.
- Adequate organ function:
    - ANC >= 1,500/mcL
    - Platelets >= 100,000/mcL
    - Hemoglobin >= 9.0 g/dL
    - Total bilirubin =< 1.5 x ULN
    - ALT and AST =< 3 x ULN
    - Creatinine clearance >= 30 mL/min
- No leptomeningeal disease or more than 2 seizures in the last 28 days.
- No prior treatment with sacituzumab govitecan.
- No systemic cancer therapy within the past 30 days.
"""
```

https://www.cancer.gov/research/participate/clinical-trials-search/v?a=50&id=NCI-2020-07706&loc=0&rl=1&t=C4872

Demo: Clinical Trial Matching Example

```
# Step 3: Initialize the Ollama Client
ollama_client = Client(host='http://10.95.46.94:53441', headers={"Authorization": "Bearer hasans10:..."})
ollama_client.pull('llama3.2')  # Load the model
```

```
# Step 4: Define a function to query the LLM for patient-trial matching
def match_patient_with_trial(patient_notes, trial_description):
    query = f"""
    Based on the following patient clinical notes and clinical trial description, determine if the patient is 
eligible for the trial.
    Respond clearly with either 'The patient is eligible' or 'The patient is not eligible,' followed by an 
explanation.

    Patient Clinical Notes:
    {patient_notes}

    Clinical Trial Description:
    {trial_description}
    """

    # Stream response from Ollama LLM
    stream = ollama_client.chat(
        model='llama3.2',
        messages=[{'role': 'user', 'content': query}],
        stream=True
    )

    # Gather the full response
    response_text = ""
    for chunk in stream:
        response_text += chunk['message']['content']

    return response_text
```

Demo: Clinical Trial Matching Example

```
# Step 5: Perform the matching and check if it's a match
def check_if_match(response_text):
    # Lowercase response for easier comparison
    response_text_lower = response_text.lower()

    # Check for eligibility based on more flexible patterns
    if "the patient is eligible" in response_text_lower:
        return "Match"
    elif "the patient is not eligible" in response_text_lower or "not eligible" in
response_text_lower:
        return "Not a Match"
    else:
        # If neither statement is clear, mark it as uncertain
        return "Unclear - Need More Information"

# Perform the matching and print result
match_response = match_patient_with_trial(patient_notes,
clinical_trial_description)
match_status = check_if_match(match_response)

print(f"LLM Response for Patient Matching:\n{match_response}")
print(f"\nIs the patient a match? {match_status}")
```

Important Reminder

- Need assistance? Feel free to contact us at:

hpchelp@hpc.mssm.edu

Acknowledgements

- Supported by the Clinical and Translational Science Awards (CTSA) grant UL1TR004419 from the National Center for Advancing Translational Sciences, National Institutes of Health.

CTSA Clinical & Translational Science Awards®