Harnessing the Power of Soft Prompts: A Hands-On Guide to Fine-Tuning for Text Summarization

ANSHUL SHIVHARE · Follow

11 min read · May 10, 2024

In our previous blog, we delved into the concept of Soft Prompts and their role in enhancing the performance of Large Language Models (LLMs) (Previous Blog). Building upon that foundation, we now embark on a practical journey towards leveraging Soft Prompts for text summarization tasks.

Setting the Stage: For our text summarization endeavor, we'll be utilizing the CNN/Daily Mail dataset, a renowned corpus in natural language processing (Dataset). Our approach involves fine-tuning a GPT-2 model with a Soft Prompt initialized specifically for the summarization task.

Understanding the Architecture:

To comprehend our methodology better, let's break down the architecture we'll be working with:

- **Soft Prompt Embedding Layer:** We begin by creating a dedicated embedding layer for our soft prompts. This layer, with its own parameters separate from the GPT-2 model, plays a pivotal role in guiding the summarization process.
- **Model with Soft Prompt:** Next, we modify the GPT-2 model to seamlessly integrate the soft prompt embeddings at the beginning of the input sequence, enhancing the model's ability to generate concise summaries.

- **Fine-tuning Architecture and Hyperparameters:** We define the architectural considerations and hyperparameters crucial for the fine-tuning process, ensuring efficient training and optimal performance.

Fine-Tuning on Text Summarization:

- **Dataset:** I'm using the CNN/Daily Mail dataset, which has lots of news articles and their summaries.
- **Training Steps:** During training, I'll add my Soft Prompts to the input and teach the model to use them. I'll keep the original model mostly unchanged to preserve its previous learning.
- **Step-by-Step Code:** I'll guide you through the code, explaining each part as I go.

Now, let's delve into the code implementation, where we'll walk through the process of fine-tuning GPT-2 with soft prompts for text summarization.

```
import numpy as np
import pandas as pd
import csv
import nltk
import random
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
import string
import torch
from torch.nn.utils.rnn import pad_sequence
import torch.nn as nn
from transformers import GPT2LMHeadModel, GPT2Tokenizer,GPT2Model, GPT2Confi
from sklearn.model_selection import train_test_split
from tqdm import tqdm
from nltk.translate.bleu_score import corpus_bleu
from rouge_score import rouge_scorer
from bert_score import BERTScorer
from transformers import BertTokenizer, BertForMaskedLM, BertModel
```

In this section, we import the necessary libraries required for data preprocessing, model training, and evaluation. These libraries include NumPy, pandas, NLTK

(Natural Language Toolkit), PyTorch, Transformers, and various evaluation metrics libraries such as BLEU score, ROUGE score, and BERTScore.

```
# Function for text preprocessing
def preprocess_text(text):
    """
    Preprocesses the input text by converting to lowercase, removing punctua

    Args:
    text (str): Input text to be preprocessed.

    Returns:
    str: Preprocessed text.
    """
    # Convert to lowercase
    text = text.lower()

    # Remove punctuation and digits
    text = text.translate(str.maketrans('', '', string.punctuation + string.

    # Tokenization
    tokens = word_tokenize(text)

    # Remove stop words
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]

    # Join the tokens back into a string
    preprocessed_text = ' '.join(tokens)

    return preprocessed_text
```

This function preprocesses text data by converting it to lowercase, removing punctuation and digits, tokenizing it, and removing stop words using NLTK.

```
# Define a function to tokenize, convert text to indices, and pad sequences
def tokenize_and_pad(data_list, max_article_length=1021, max_highlights_leng
    """
    Tokenizes the input data and pads sequences to specified lengths.

    Args:
    data_list (list): List of tuples containing articles and highlights.
    max_article_length (int): Maximum length of articles.
    max_highlights_length (int): Maximum length of highlights.
```

```
    Returns:
    list: Tokenized and padded data list.
    """
    tokenized_data_list = []
    for article, highlights in data_list:
        # Tokenize and convert to indices
        article_tokens = tokenizer.encode(article, add_special_tokens=True)
        highlights_tokens = tokenizer.encode(highlights, add_special_tokens=

        # Pad sequences to specified lengths
        padded_article_tokens = torch.tensor(article_tokens + [tokenizer.con
        padded_highlights_tokens = torch.tensor(highlights_tokens + [tokeniz

        # Append to the tokenized_data_list only if both token lists are not
        if len(article_tokens) > 0 and len(highlights_tokens) > 0:
            tokenized_data_list.append((padded_article_tokens, padded_highli

    return tokenized_data_list
```

This function tokenizes the input data, converts the tokens to indices using the GPT-2 tokenizer, and pads sequences to specified lengths.

```
def calculate_bleu_score(machine_results, reference_texts):
    bleu_score = corpus_bleu([[ref.split()] for ref in reference_texts], [ge
    return bleu_score
```

This function calculates the BLEU score, a metric commonly used to evaluate the quality of text generation models, by comparing generated text against reference texts.

```
def calculate_rouge_scores(generated_answers, ground_truth):
    scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'], use_st
    total_rouge1, total_rouge2, total_rougeL = 0, 0, 0
    for gen, ref in zip(generated_answers, ground_truth):
        scores = scorer.score(gen, ref)
        total_rouge1 += scores['rouge1'].fmeasure
        total_rouge2 += scores['rouge2'].fmeasure
        total_rougeL += scores['rougeL'].fmeasure
    average_rouge1 = total_rouge1 / len(generated_answers)
    average_rouge2 = total_rouge2 / len(generated_answers)
    average_rougeL = total_rougeL / len(generated_answers)
```

```
    return average_rouge1, average_rouge2, average_rougeL
```

This function calculates the ROUGE scores, which measure the overlap of n-grams between generated text and reference texts, providing another evaluation metric for text generation models.

```
def calculate_bert_score(generated_answers, ground_truth):
    scorer = BERTScorer(model_type='bert-base-uncased')
    P, R, F1 = scorer.score(generated_answers, ground_truth)
    avg_precision = sum(p.mean() for p in P) / len(P)
    avg_recall = sum(r.mean() for r in R) / len(R)
    avg_f1 = sum(f1.mean() for f1 in F1) / len(F1)
    return avg_precision, avg_recall, avg_f1
```

This function calculates the BERTScore, which computes precision, recall, and F1 score based on contextual embeddings provided by BERT, offering another perspective on the quality of generated text compared to reference texts.

```
csv_file_path = 'test.csv'

# List to store tuples of (article, highlights)
test_data_list = []

# Read CSV file and extract relevant columns
with open(csv_file_path, 'r', encoding='utf-8') as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        article = row.get('article', '')
        highlights = row.get('highlights', '')
        test_data_list.append((article, highlights))


csv_file_path = 'train.csv'

# List to store tuples of (article, highlights)
train_data_list = []

# Read CSV file and extract relevant columns
with open(csv_file_path, 'r', encoding='utf-8') as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
```

```
            article = row.get('article', '')
            highlights = row.get('highlights', '')
            train_data_list.append((article, highlights)))

    csv_file_path = 'validation.csv'

    # List to store tuples of (article, highlights)
    val_data_list = []

    # Read CSV file and extract relevant columns
    with open(csv_file_path, 'r', encoding='utf-8') as file:
        csv_reader = csv.DictReader(file)
        for row in csv_reader:
            article = row.get('article', '')
            highlights = row.get('highlights', '')
            val_data_list.append((article, highlights)))
```

These blocks read data from CSV files, extracting relevant columns (‘article’ and ‘highlights’) and storing them as tuples in separate lists for training, testing, and validation data.

```
# Set the seed for reproducibility
random.seed(14)  # You can choose any seed value

# Calculate 1% of the original data size
one_percent_size = int(0.01 * len(test_data_list))

# Randomly sample 1% of the data
onetest_data_list = random.sample(test_data_list, one_percent_size)
# Calculate 1% of the original data size
one_percent_size = int(0.01 * len(train_data_list))

# Randomly sample 1% of the data
onetrain_data_list = random.sample(train_data_list, one_percent_size)
# Calculate 1% of the original data size
one_percent_size = int(0.01 * len(val_data_list))

# Randomly sample 1% of the data
oneval_data_list = random.sample(val_data_list, one_percent_size)
```

These blocks randomly sample 1% of the data from the training, testing, and validation datasets for efficient processing during model development.

```
# Apply preprocessing to your data
ptrain_data_list = [(preprocess_text(article), preprocess_text(highlights))
ptest_data_list = [(preprocess_text(article), preprocess_text(highlights)) f
pval_data_list = [(preprocess_text(article), preprocess_text(highlights)) fo
```

This block applies text preprocessing (lowercasing, punctuation removal, tokenization, and stop word removal) to the sampled data for all three sets: training, testing, and validation.

```
# Load GPT-2 tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

# Define a pad token and add it to the tokenizer
pad_token = tokenizer.eos_token
tokenizer.add_tokens([pad_token])
```

This block loads the GPT-2 tokenizer and defines a special token (‘<pad>’) to be used for padding sequences, then adds it to the tokenizer.

```
# Apply tokenization and padding to your datasets
max_article_length = 1021
max_highlights_length = 1024
tokenized_train_data_list = tokenize_and_pad(ptrain_data_list, max_article_l
tokenized_test_data_list = tokenize_and_pad(ptest_data_list, max_article_len
tokenized_val_data_list = tokenize_and_pad(pval_data_list, max_article_lengt
```

This block tokenizes and pads the preprocessed data to the specified maximum lengths for articles and highlights, creating tokenized and padded datasets for training, testing, and validation.

```
# Initialize lists to store input and target ids
input_ids_train = []
target_ids_train = []
```

```
# Specify maximum lengths
max_article_length = 1021
max_highlights_length = 1024

# Iterate through the tokenized data
for padded_article_tokens, padded_highlights_tokens in tokenized_train_data_
    # Truncate article tokens if greater than max_article_length
    truncated_article_tokens = padded_article_tokens[:max_article_length]

    # Truncate highlights tokens if greater than max_highlights_length
    truncated_highlights_tokens = padded_highlights_tokens[:max_highlights_l

    # Append truncated article tokens to input_ids_train
    input_ids_train.append(truncated_article_tokens)

    # Append truncated highlights tokens to target_ids_train
    target_ids_train.append(truncated_highlights_tokens)

# Convert the lists to PyTorch tensors
input_ids_train = torch.stack(input_ids_train)
target_ids_train = torch.stack(target_ids_train)
```

This block prepares the training data by truncating the article and highlights tokens if they exceed the maximum lengths, then converts them into PyTorch tensors.

```
# Initialize lists to store input and target ids
input_ids_val = []
target_ids_val = []

# Specify maximum lengths
max_article_length = 1021
max_highlights_length = 1024

# Iterate through the tokenized data
for padded_article_tokens, padded_highlights_tokens in tokenized_val_data_li
    # Truncate article tokens if greater than max_article_length
    truncated_article_tokens = padded_article_tokens[:max_article_length]

    # Truncate highlights tokens if greater than max_highlights_length
    truncated_highlights_tokens = padded_highlights_tokens[:max_highlights_l

    # Append truncated article tokens to input_ids_val
    input_ids_val.append(truncated_article_tokens)
```

```
        # Append truncated highlights tokens to target_ids_val
        target_ids_val.append(truncated_highlights_tokens)

    # Convert the lists to PyTorch tensors
    input_ids_val = torch.stack(input_ids_val)
    target_ids_val = torch.stack(target_ids_val)
```

This block prepares the validation data similarly to the training data, truncating and converting it into PyTorch tensors.

```
# Load GPT-2 model and tokenizer
model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
gpt2_model = GPT2LMHeadModel.from_pretrained(model_name)
```

This block loads the GPT-2 model and tokenizer.

```
# Define the number of prompts and embedding size
num_prompts_token = 3  # "summarize the following text"
embedding_size = 768

# Define a specific sentence
sentence = "summarize"

# Tokenize the sentence
input_ids = tokenizer.encode(sentence, return_tensors='pt')

# Get the embeddings for the input_ids from the GPT-2 model
gpt2_embeddings = gpt2_model.transformer.wte(input_ids)

# Create an embedding layer for soft prompts and initialize with the sentenc
soft_prompt_embeddings = nn.Embedding(num_prompts_token, embedding_size)
soft_prompt_embeddings.weight.data.copy_(gpt2_embeddings.squeeze(0))
```

This block defines the number of tokens in prompt and embedding size, tokenizes a specific sentence, retrieves embeddings from the GPT-2 model, and initializes an embedding layer for soft prompts with the sentence embeddings.

```
# Concatenate soft prompt embeddings at the beginning of the input sequence
class GPT2WithPromptTuning(nn.Module):
    def __init__(self, gpt2_model, soft_prompt_embeddings):
        super(GPT2WithPromptTuning, self).__init__()
        self.gpt2_model = gpt2_model
        self.soft_prompt_embeddings = soft_prompt_embeddings

    def forward(self, input_ids, soft_prompt_ids):
        # Get the embeddings for the input_ids from the GPT-2 model
        gpt2_embeddings = self.gpt2_model.transformer.wte(input_ids)
        # Get the embeddings for the soft prompts
        soft_prompt_embeds = self.soft_prompt_embeddings(soft_prompt_ids)
         # Concatenate the embeddings
        # print("Shape of soft prompt embeddings:", soft_prompt_embeds.shape)
        # print("soft prompt embeddings:", soft_prompt_embeds)
        # Shape of Soft prompt will be  3 * 768
        # print("Shape of input embeddings:", gpt2_embeddings.shape)
        # print("soft input prompt embeddings:", gpt2_embeddings)
        # Shape of Soft prompt will be  1021* 768

        # Concatenate the embeddings
        embeddings = torch.cat([soft_prompt_embeds, gpt2_embeddings], dim=0)
        # Shape of concatenated embedding will be  1024* 768
        # print("Shape of concatenated embeddings:", embeddings.shape)
        # print("soft of concatenated embeddings:", embeddings)

        # Pass the concatenated embeddings through the GPT-2 model
        outputs = self.gpt2_model(inputs_embeds=embeddings)

        return outputs
```

This block defines a custom GPT-2 model with prompt tuning, where soft prompt embeddings are concatenated at the beginning of the input sequence.

```
# Initialize the model
model = GPT2WithPromptTuning(gpt2_model, soft_prompt_embeddings)

# Freeze GPT-2 model weights
for param in model.gpt2_model.parameters():
    param.requires_grad = False
```

This block initializes the GPT-2 model with prompt tuning and freezes the weights of the original GPT-2 model to prevent them from being updated during training.

```
# Define hyperparameters
batch_size = 8
epochs = 2
learning_rate = 2e-3
gradient_clip_value = 1.0

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

# Move model to GPU
model.to(device)

# Define optimizer and criterion
optimizer = torch.optim.AdamW(model.soft_prompt_embeddings.parameters(), lr=
criterion = nn.CrossEntropyLoss(ignore_index=-100)

soft_prompt_ids = torch.tensor([0, 1, 2])

# Lists to store scores
train_bleu_scores = []
train_bert_scores = []
train_rouge1_scores = []
train_rouge2_scores = []
train_rougeL_scores = []

val_bleu_scores = []
val_bert_scores = []
val_rouge1_scores = []
val_rouge2_scores = []
val_rougeL_scores = []
```

This block defines hyperparameters such as batch size, number of epochs, learning rate, and gradient clip value, and initializes the AdamW optimizer and cross-entropy loss function for training the model.

```
# Training loop
for epoch in range(epochs):
    # Create a tqdm progress bar for the training data
    data_iterator = tqdm(zip(input_ids_train, target_ids_train), desc=f'Epoc
```

```
    for input_ids, target_ids in data_iterator:
        optimizer.zero_grad()

        # Move input and target tensors to GPU
        input_ids, target_ids = input_ids.to(device), target_ids.to(device)
        # print("input",input_ids.shape,"target",target_ids.shape)
        # print("input",len(input_ids),"target",len(target_ids))

        outputs = model(input_ids, soft_prompt_ids.to(device))
        logits = outputs.logits if hasattr(outputs, "logits") else outputs.l

        loss = criterion(logits, target_ids)
        loss.backward()

        # Gradient clipping to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(model.parameters(), gradient_clip_valu

        optimizer.step()

        # Update the progress bar description with the current loss
        data_iterator.set_postfix(loss=loss.item())

        # Convert tensor predictions and references to lists
        predictions = logits.argmax(dim=-1).squeeze(0).tolist()
        references = target_ids.squeeze(0).tolist()

        # Calculate BLEU Score for training
        bleu_score = calculate_bleu_score([tokenizer.decode(predictions)], [
        train_bleu_scores.append(bleu_score)

        # Calculate BERTScore for training
        bert_precision, bert_recall, bert_f1 = calculate_bert_score([tokeniz
        train_bert_scores.append(bert_f1)

        # Calculate ROUGE Scores for training
        rouge1, rouge2, rougeL = calculate_rouge_scores([tokenizer.decode(pr
        train_rouge1_scores.append(rouge1)
        train_rouge2_scores.append(rouge2)
        train_rougeL_scores.append(rougeL)


#     # Validation loop
    model.eval()
    val_losses = []
    val_bleu_scores_epoch = []
    val_bert_scores_epoch = []
    val_rouge1_scores_epoch = []
    val_rouge2_scores_epoch = []
    val_rougeL_scores_epoch = []
    val_losses = []
    with torch.no_grad():
        for input_ids_val, target_ids_val in zip(input_ids_val, target_ids_v
```

```
            input_ids_val, target_ids_val = input_ids_val.to(device), target
            outputs_val = model(input_ids_val, soft_prompt_ids.to(device))
            logits_val = outputs_val.logits if hasattr(outputs_val, "logits"
            loss_val = criterion(logits_val, target_ids_val)
            val_losses.append(loss_val.item())
            # Convert tensor predictions and references to lists
            predictions_val = logits_val.argmax(dim=-1).squeeze(0).tolist()
            references_val = target_ids_val.squeeze(0).tolist()

            # Calculate BLEU Score for validation
            bleu_score_val = calculate_bleu_score([tokenizer.decode(predicti
            val_bleu_scores_epoch.append(bleu_score_val)

            # Calculate BERTScore for validation
            bert_precision_val, bert_recall_val, bert_f1_val = calculate_ber
            val_bert_scores_epoch.append(bert_f1_val)

            # Calculate ROUGE Scores for validation
            rouge1_val, rouge2_val, rougeL_val = calculate_rouge_scores([tok
            val_rouge1_scores_epoch.append(rouge1_val)
            val_rouge2_scores_epoch.append(rouge2_val)
            val_rougeL_scores_epoch.append(rougeL_val)

    # Calculate average validation loss
    avg_val_loss = sum(val_losses) / len(val_losses)
    print("epoch :", epoch + 1,"train_loss :", loss.item(),"val_loss :", avg

    # Calculate average validation scores
    avg_bleu_score_val = sum(val_bleu_scores_epoch) / len(val_bleu_scores_ep
    avg_bert_score_val = sum(val_bert_scores_epoch) / len(val_bert_scores_ep
    avg_rouge1_score_val = sum(val_rouge1_scores_epoch) / len(val_rouge1_sco
    avg_rouge2_score_val = sum(val_rouge2_scores_epoch) / len(val_rouge2_sco
    avg_rougeL_score_val = sum(val_rougeL_scores_epoch) / len(val_rougeL_sco

    print("Validation BLEU Score:", avg_bleu_score_val)
    print("Validation BERTScore:", avg_bert_score_val)
    print("Validation ROUGE-1 Score:", avg_rouge1_score_val)
    print("Validation ROUGE-2 Score:", avg_rouge2_score_val)
    print("Validation ROUGE-L Score:", avg_rougeL_score_val)

    # Append validation scores
    val_bleu_scores.append(avg_bleu_score_val)
    val_bert_scores.append(avg_bert_score_val)
    val_rouge1_scores.append(avg_rouge1_score_val)
    val_rouge2_scores.append(avg_rouge2_score_val)
    val_rougeL_scores.append(avg_rougeL_score_val)

    # Set the model back to training mode
    model.train()

# Close the tqdm progress bar
data_iterator.close()
```

This code snippet illustrates a training loop for a sequence-to-sequence model. It iterates over epochs, optimizing the model's parameters using gradient descent. During each epoch, it computes the loss function, applies gradient clipping to prevent exploding gradients, and updates the model. The training progress is monitored using a progress bar, displaying the current loss. Additionally, it calculates various evaluation metrics such as BLEU, BERTScore, and ROUGE on both the training and validation sets to assess model performance. Finally, it prints the average loss and evaluation scores for each epoch. The model is switched between training and evaluation modes accordingly.

```
# Calculate average scores for training
avg_train_bleu_score = sum(train_bleu_scores) / len(train_bleu_scores)
avg_train_bert_score = sum(train_bert_scores) / len(train_bert_scores)
avg_train_rouge1_score = sum(train_rouge1_scores) / len(train_rouge1_scores)
avg_train_rouge2_score = sum(train_rouge2_scores) / len(train_rouge2_scores)
avg_train_rougeL_score = sum(train_rougeL_scores) / len(train_rougeL_scores)

print("Average Training BLEU Score:", avg_train_bleu_score)
print("Average Training BERTScore:", avg_train_bert_score)
print("Average Training ROUGE-1 Score:", avg_train_rouge1_score)
print("Average Training ROUGE-2 Score:", avg_train_rouge2_score)
print("Average Training ROUGE-L Score:", avg_train_rougeL_score)

# Calculate average scores for validation
avg_val_bleu_score = sum(val_bleu_scores) / len(val_bleu_scores)
avg_val_bert_score = sum(val_bert_scores) / len(val_bert_scores)
avg_val_rouge1_score = sum(val_rouge1_scores) / len(val_rouge1_scores)
avg_val_rouge2_score = sum(val_rouge2_scores) / len(val_rouge2_scores)
avg_val_rougeL_score = sum(val_rougeL_scores) / len(val_rougeL_scores)

print("Average Validation BLEU Score:", avg_val_bleu_score)
print("Average Validation BERTScore:", avg_val_bert_score)
print("Average Validation ROUGE-1 Score:", avg_val_rouge1_score)
print("Average Validation ROUGE-2 Score:", avg_val_rouge2_score)
print("Average Validation ROUGE-L Score:", avg_val_rougeL_score)


 # Save model weights and tokenizer
torch.save(model.state_dict(), 'Summarize_weights1.pth')
```

Following the training loop, this segment calculates the average evaluation scores

for both the training and validation sets. It computes the average BLEU, BERTScore, and ROUGE scores achieved during training and validation phases. These scores offer insights into the model's performance in generating summaries. Additionally, it prints these average scores to provide a summary of the model's performance across all epochs.

Finally, the trained model's weights and the associated tokenizer are saved for potential future use or deployment, ensuring the trained model's persistence.

In this blog, we delved into the intriguing concept of soft prompts and their application in enhancing the performance of large language models like GPT-2. By creating a new embedding layer of size 3 * *768 for soft prompts and concatenating it with the input embedding layer (1021* * 768), we achieved prompt tuning. This innovative approach allowed us to train the first embedding layer while keeping the rest of the GPT-2 layers frozen, thereby fine-tuning the model to better understand and respond to specific prompts.

We explored how this technique can unlock the full potential of language models, enabling more precise and tailored responses to input prompts. To delve deeper into the concept of soft prompts and their implications, check out my first blog on the topic here.

Your feedback and suggestions are valuable to me. If you found this article helpful or have any further queries, please don't hesitate to leave a comment below. And if you enjoyed the read, a thumbs up is always appreciated!

Connect with me on my other social media platforms for more updates and insights:

- Google Scholar
- Github
- YouTube
- LinkedIn
- Personal Website

Keep exploring, learning, and extracting insights! Until next time.

Soft Prompt | Prompt Tuning | Prompt Engineering | Llm Finetuning | Gpt 2

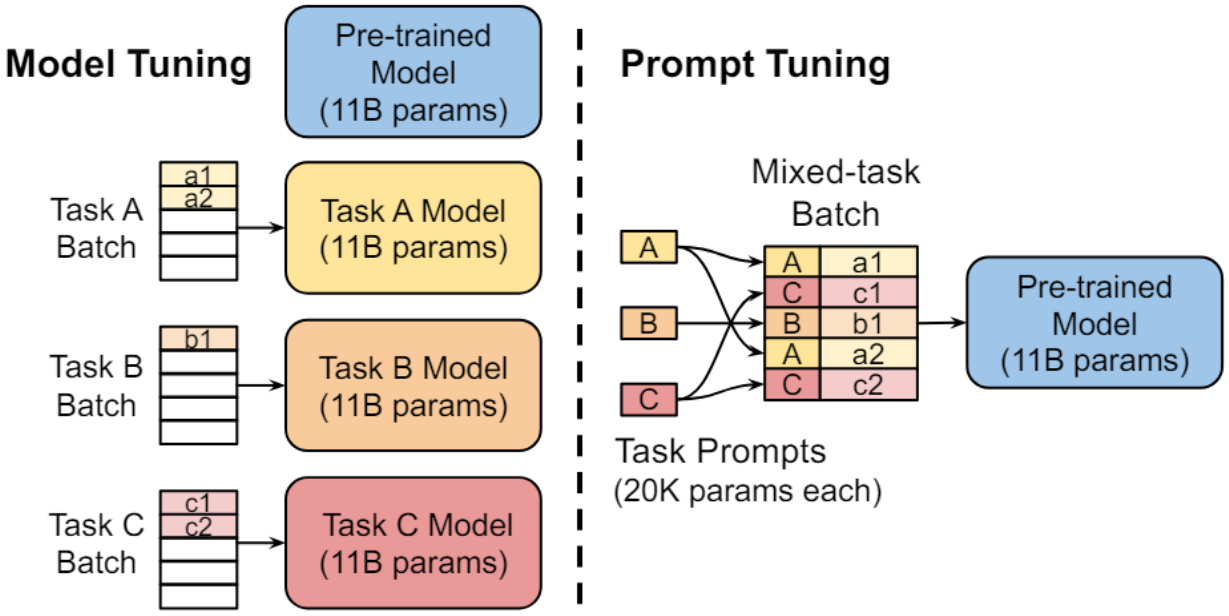
Follow

Written by ANSHUL SHIVHARE

18 Followers

Hey! I'm Anshul—a Data Scientist, Researcher, AI Engineer, Explorer, Blogger, and now diving into writing! Stick around for some simple, fun adventures in words

More from ANSHUL SHIVHARE



ANSHUL SHIVHARE

Unlocking the Power of Large Language Models with Soft Prompts

Welcome to another series in the world of Large Language Models (LLMs), where we explore Soft Prompts. In this series, we'll delve into why

ANSHUL SHIVHARE

Accelerating Inference: Merging Finetuned LLM Weights for VLLM Efficiency

Turbocharging Finetuned LLAMA2: Faster Inferencing with VLLM

Mar 30 3

ANSHUL SHIVHARE

Fine-tuning LLAMA2 on the Public Dataset (Part 1)

Finetuning LLAMA2 and comparing results with non-finetuned versions.

Feb 18 81 3

ANSHUL SHIVHARE

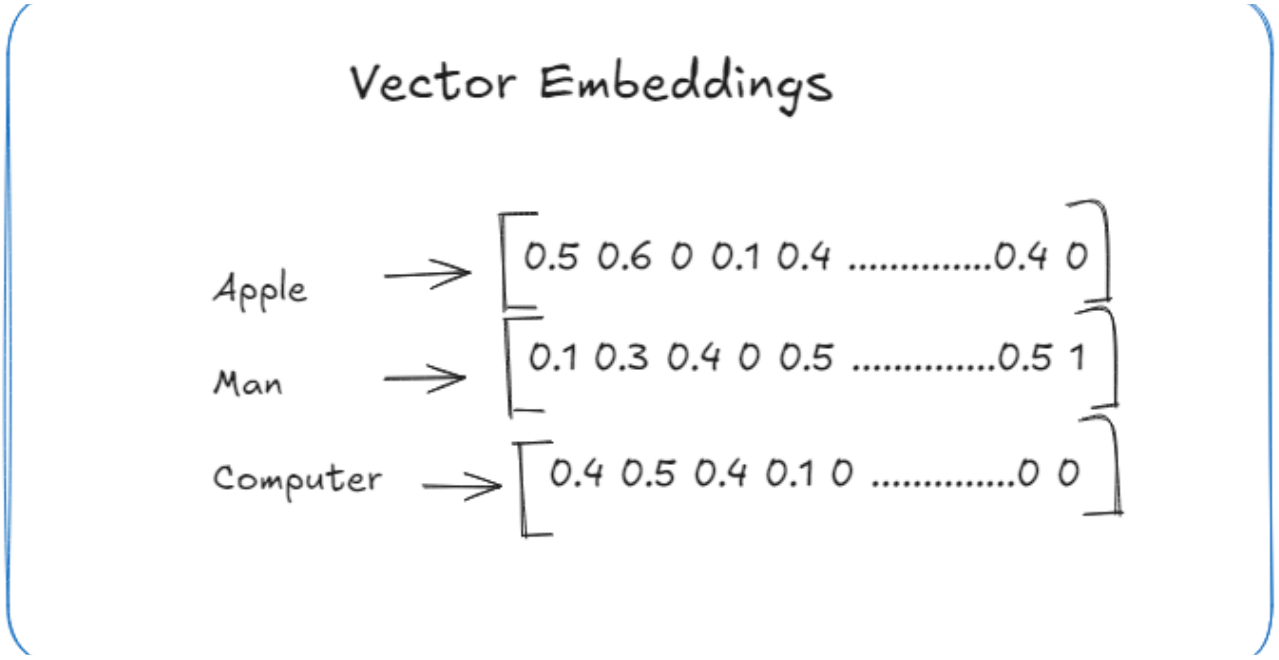
Unveiling the Power of LLM's Using VLLM

Optimize Your Extraction Workflow with LLAMA2 and VLLM

Mar 30 6 1

See all from ANSHUL SHIVHARE

Recommended from Medium



Mdabdullahalhasib in Towards AI

A Complete Guide to Embedding For NLP & Generative AI/LLM

Understand the concept of vector embedding, why it is needed, and implementation with LangChain.

4d ago

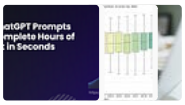
Chuck Russell

Exploring the Impact of Fine-Tuning on Large Language Models

We all would agree that Large Language Models (LLMs) have revolutionized the field of AI, excelling in natural language processing...

Lists

May 23 8



ChatGPT prompts

50 stories · 2125 saves

Natural Language Processing

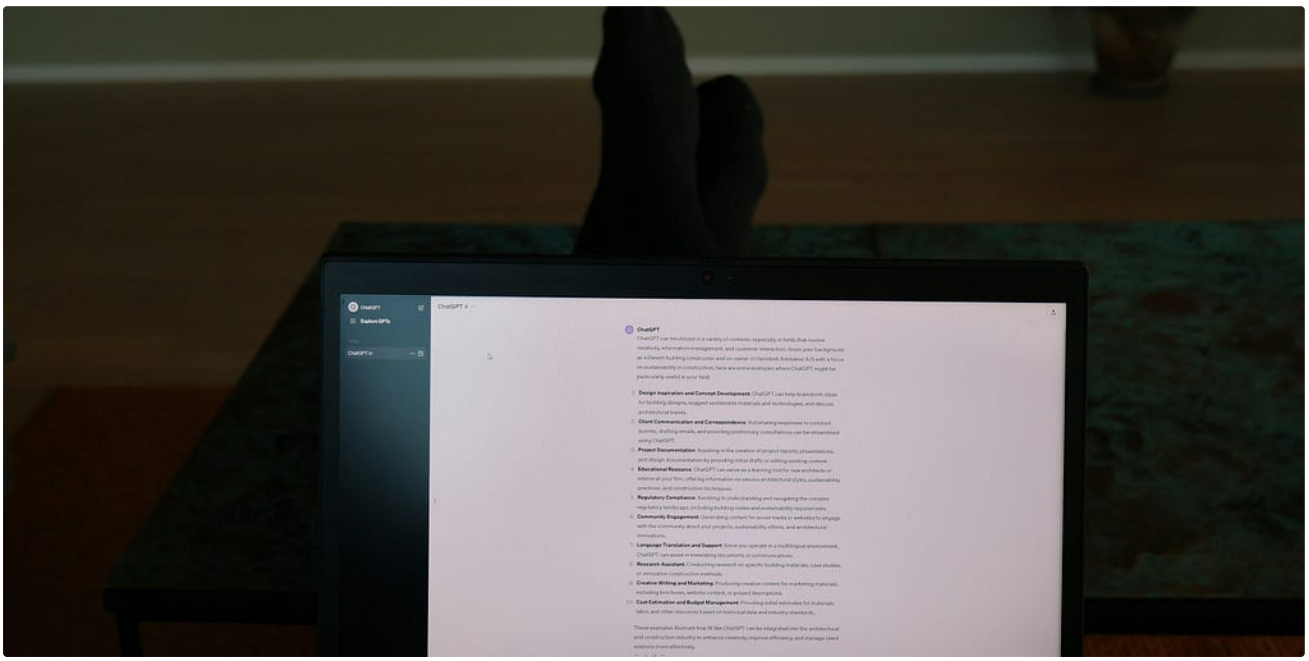
1767 stories · 1368 saves

ChatGPT

21 stories · 846 saves

Generative AI Recommended Reading

52 stories · 1449 saves



Nizamuddin Siddiqui

55 Prompt Output Formats for ChatGPT

You won't find it on any website

Sep 5 156 3

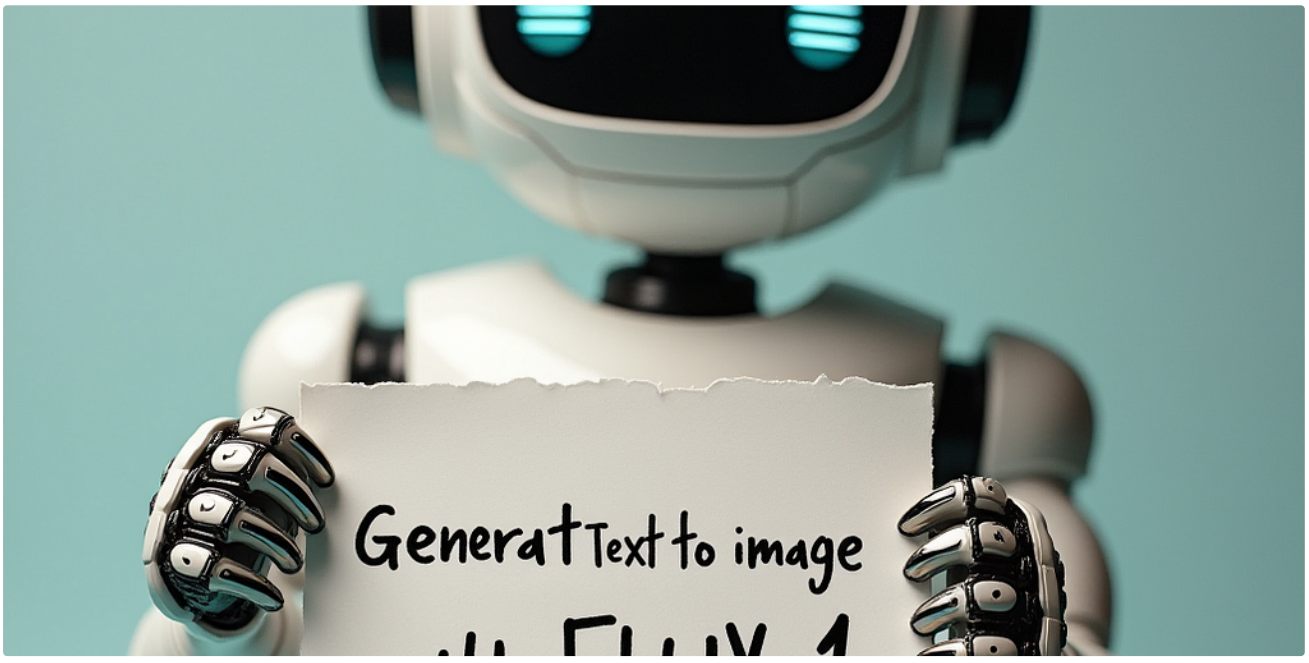
entity_group	score	word	start	end
AGE	0.541046	13 y.o.	11	19
SEX	0.549964	male	19	24
CLINICAL_EVENT	0.598481	seen	30	35
DISEASE_DISORDER	0.802787	PAH deficiency	53	68
SIGN_SYMPTOM	0.798482	vitamin D deficency	80	100
SIGN_SYMPTOM	0.571304	anemia	104	111
MEDICATION	0.789797	viatmin B1, B2	125	140
MEDICATION	0.871156	iron tablets	144	157

Alva Rani James, PhD

Reviewing NER models: DeBERTa versus BioBERT for electronic health record (EHR) data

DeBERTa and BERT-based

May 28 11



Md Monsur ali in Level Up Coding

How to Generate Text to Image Using FLUX.1: A Step-by-Step Colab Guide

Explore the Power of FLUX.1 model for Text-to-Image Generation — Set Up, Model

Hassaan Idrees

Fine-Tuning Transformers: Techniques for Improving Model Performance

Introduction

Jul 27

See more recommendations