SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

Advanced Techniques in Generative AI with OpenAI Models (Lab: Google Generative AI Studio) (PG120502J)

List of Programs

Program 1: Fine-tuning GPT for Text Generation

Title: Fine-tuning GPT for Text Generation

Aim: To understand and implement the process of fine-tuning a pre-trained GPT model on a custom dataset for specific text generation tasks.

Procedure:

1. **Prepare Dataset:** Collect and format a dataset relevant to the desired text generation task. The dataset should be in a JSONL format, where each line contains a "prompt" and a "completion" field.
2. **Upload Dataset:** Upload the prepared dataset to the Google Generative AI Studio or OpenAI platform.
3. **Initiate Fine-tuning Job:** Use the platform's API or UI to initiate a fine-tuning job, specifying the base GPT model and the uploaded dataset.
4. **Monitor Training:** Monitor the fine-tuning process, observing metrics like loss and accuracy.
5. **Deploy Fine-tuned Model:** Once training is complete, deploy the fine-tuned model.
6. **Test Model:** Send inference requests to the fine-tuned model with new prompts and evaluate its generated text.

Source Code (Conceptual Python using `gemini-2.0-flash` for generation, with fine-tuning steps described):

```
# This code is conceptual and demonstrates the typical workflow.
# Actual fine-tuning involves using specific SDKs or API endpoints
# provided by Google Generative AI Studio or OpenAI, which are not
# directly executable here.

import json
import time

# Placeholder for API key - In a real application, this would be securely
managed.
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

def prepare_dataset(data_list, filename="training_data.jsonl"):
    """
    Prepares a dataset in JSONL format for fine-tuning.
    Each item in data_list should be a dictionary with 'prompt' and
'completion' keys.
    """
    with open(filename, 'w') as f:
        for item in data_list:
            f.write(json.dumps(item) + '\n')
    print(f"Dataset saved to {filename}")
    return filename

def initiate_fine_tuning(dataset_path, base_model="gemini-2.0-flash"):
    """
    Conceptual function to initiate a fine-tuning job.
    In a real scenario, this would involve calling the platform's API.
    """
    print(f"Initiating fine-tuning for {base_model} with dataset
{dataset_path}...")
    # Simulate API call and job ID
    job_id = f"ft-job-{int(time.time())}"
    print(f"Fine-tuning job initiated with ID: {job_id}")
    print("Please refer to the Google Generative AI Studio or OpenAI
documentation for actual API calls.")
    return job_id

def get_fine_tuned_model_id(job_id):
    """
    Conceptual function to get the fine-tuned model ID after job completion.
    """
    print(f"Checking status of job {job_id}...")
    # Simulate waiting for job to complete
    time.sleep(10) # In reality, this would be a polling mechanism
    fine_tuned_model_id = f"ft-model-{job_id}"
    print(f"Fine-tuning job {job_id} completed. Model ID:
{fine_tuned_model_id}")
    return fine_tuned_model_id

async def generate_text_with_model(model_id, prompt):
    """
    Generates text using a specified model (conceptual, using gemini-2.0-
flash structure).
    """
    print(f"\nGenerating text with model: {model_id}")
    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/${model_id}:generate
Content?key=${API_KEY}`;

    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            text = result.candidates[0].content.parts[0].text;
            print(f"Generated Text:\n{text}")
            return text
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during generation: {e}"

async def main():
    # 1. Prepare Dataset (Example)
    training_data = [
        {"prompt": "Write a short story about a brave knight.", "completion":
"Sir Reginald, a knight of unwavering courage, set forth to rescue the
princess from the dragon's lair. His sword gleamed under the moonlight."},
        {"prompt": "Write a short story about a magical forest.",
"completion": "The Whispering Woods were alive with ancient magic. Trees
hummed lullabies, and glowing flora illuminated hidden paths. Fairies danced
in the moonbeams."}
    ]
    dataset_file = prepare_dataset(training_data)

    # 2. Initiate Fine-tuning Job (Conceptual)
    # In a real scenario, you'd upload dataset and get a file ID, then use
that for fine-tuning.
    # The actual API call for fine-tuning would look something like:
    # openai.FineTuningJob.create(training_file="file-xxxxxxxx", model="gpt-
3.5-turbo")
    # For Google Generative AI Studio, refer to their specific fine-tuning
API.
    fine_tuning_job_id = initiate_fine_tuning(dataset_file)

    # 3. Get Fine-tuned Model ID (Conceptual)
    # This would involve polling the API until the job status is 'succeeded'.
    fine_tuned_model = get_fine_tuned_model_id(fine_tuning_job_id)

    # 4. Test Model (Conceptual, using gemini-2.0-flash structure for
demonstration)
    # Replace 'fine_tuned_model' with the actual model ID obtained from the
fine-tuning process.
    if fine_tuned_model:
        await generate_text_with_model(fine_tuned_model, "Write a short story
about a talking cat.")
        await generate_text_with_model(fine_tuned_model, "Describe a
futuristic city.")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# Example:
# <button onclick="main()">Run Fine-tuning Example</button>
# main() # This would run it directly if not in a specific event handler
```

Input: A JSONL file containing prompt-completion pairs. Example `training_data.jsonl`:

```
{"prompt": "Write a short story about a brave knight.", "completion": "Sir
Reginald, a knight of unwavering courage, set forth to rescue the princess
from the dragon's lair. His sword gleamed under the moonlight."}
{"prompt": "Write a short story about a magical forest.", "completion": "The
Whispering Woods were alive with ancient magic. Trees hummed lullabies, and
glowing flora illuminated hidden paths. Fairies danced in the moonbeams."}
```

And a prompt for generation: `"Write a short story about a talking cat."`

Expected Output: The fine-tuned model will generate text that aligns with the style and content of the training data. For the input `"Write a short story about a talking cat."`, the output might be:

```
"Whiskers, a tabby with an unusual knack for philosophy, often debated the
meaning of purrs with his human, Sarah. 'It's a complex vocalization,' he'd
explain, 'a symphony of contentment and demand.'"
```

The exact output will vary based on the model and fine-tuning.

Program 2: Implementing Self-supervised Learning with ChatGPT

Title: Implementing Self-supervised Learning with ChatGPT

Aim: To explore the concept of self-supervised learning by demonstrating how models like ChatGPT (which are pre-trained using self-supervision) can be utilized for downstream tasks without explicit labeling, leveraging their learned representations.

Procedure:

1. **Understand Self-supervision:** Review the principles of self-supervised learning, particularly masked language modeling or next-token prediction, which are foundational to models like GPT.
2. **Choose a Downstream Task:** Select a task where pre-trained language models excel due to their self-supervised training (e.g., text summarization, sentiment analysis, question answering).
3. **Prompt Engineering:** Formulate effective prompts to guide ChatGPT to perform the chosen task. This involves providing context and instructions.
4. **Utilize ChatGPT API:** Send the engineered prompts to the ChatGPT API.
5. **Analyze Output:** Evaluate the model's response for accuracy and relevance to the task, demonstrating its ability to perform tasks based on its self-supervised pre-training.

Source Code (Conceptual Python using `gemini-2.0-flash`):

```
# This code demonstrates using a pre-trained model (like ChatGPT, here
simulated with gemini-2.0-flash)
# for a downstream task, leveraging its self-supervised learning.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def perform_sentiment_analysis(text):
    """
    Performs sentiment analysis using a pre-trained model via prompt
engineering.
    This simulates how a self-supervised model can be applied to a task.
    """
    prompt = f"Analyze the sentiment of the following text and classify it as
positive, negative, or neutral. Provide a brief explanation.\n\nText:
'{text}'"

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    print(f"\nAnalyzing sentiment for: '{text}'")
    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
```

```
            sentiment_analysis = result.candidates[0].content.parts[0].text;
            print(f"Sentiment Analysis:\n{sentiment_analysis}")
            return sentiment_analysis
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during sentiment analysis: {e}"

async def main():
    # Example 1: Positive sentiment
    await perform_sentiment_analysis("I absolutely loved the movie! The
acting was superb and the plot was engaging.")

    # Example 2: Negative sentiment
    await perform_sentiment_analysis("The service was terrible and the food
was cold. Very disappointed.")

    # Example 3: Neutral sentiment
    await perform_sentiment_analysis("The meeting is scheduled for 3 PM
tomorrow in Conference Room B.")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: Text snippets for sentiment analysis. Example 1: `"I absolutely loved the movie! The acting was superb and the plot was engaging."` Example 2: `"The service was terrible and the food was cold. Very disappointed."` Example 3: `"The meeting is scheduled for 3 PM tomorrow in Conference Room B."`

Expected Output: For Example 1:

```
Sentiment: Positive
Explanation: The text uses strong positive language like "absolutely loved,"
"superb," and "engaging."
```

For Example 2:

```
Sentiment: Negative
Explanation: The text contains negative terms such as "terrible," "cold," and
"disappointed."
```

For Example 3:

```
Sentiment: Neutral
Explanation: The text describes a factual event without expressing any strong
positive or negative opinions.
```

The exact phrasing may vary, but the sentiment classification should be correct.

Program 3: Implement Image Classification and Retrieval Using Contrastive Objectives with ChatGPT

Title: Implement Image Classification and Retrieval Using Contrastive Objectives with ChatGPT

Aim: To understand and conceptually implement image classification and retrieval by leveraging the power of contrastive learning, potentially using a vision-language model like CLIP (which underpins some multi-modal capabilities of models like GPT-4 or similar) and interacting with it via a text-based interface like ChatGPT.

Procedure:

1. **Understand Contrastive Learning:** Learn about contrastive learning principles, where the model learns to pull similar samples closer together and push dissimilar samples further apart in an embedding space.
2. **Conceptual Model (CLIP-like):** Recognize that models like OpenAI's CLIP are trained on image-text pairs using contrastive objectives, allowing them to understand the relationship between images and text.
3. **Simulate Image Embedding (Conceptual):** For this lab, we will conceptually assume we have a way to get an image embedding or describe an image in text.
4. **Prompt for Classification/Retrieval:** Formulate prompts for ChatGPT to perform classification (e.g., "What is in this image?") or retrieval (e.g., "Find images similar to this description").
5. **Utilize ChatGPT (Text-based Interaction):** Send image descriptions or queries to ChatGPT and interpret its text-based responses. *Note: Direct image input to `gemini-2.0-flash` is used here to simulate image understanding.*

Source Code (Conceptual Python using `gemini-2.0-flash` for image understanding):

```
# This code conceptually demonstrates image classification and retrieval
# using contrastive objectives, by leveraging a multi-modal model like
gemini-2.0-flash
# that has been pre-trained with such objectives.

import base64
import requests # For fetching image data if needed, not directly used in the
final snippet
import io
from PIL import Image

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

def get_image_base64(image_path):
    """
    Converts an image file to a base64 encoded string.
    In a real application, this would load an actual image.
    For this example, we'll use a placeholder.
    """
    # Placeholder for a real image. In a live environment, you'd load an
image file.
    # For demonstration, let's create a dummy image.
    try:
        img = Image.new('RGB', (60, 30), color = 'red')
        buffered = io.BytesIO()
        img.save(buffered, format="PNG")
        return base64.b64encode(buffered.getvalue()).decode('utf-8')
    except Exception as e:
        print(f"Error creating dummy image: {e}")
        return None

async def classify_image(image_description_or_base64_data):
    """
    Classifies an image using a multi-modal model.
    If base64 data is provided, it uses the model's image understanding
capabilities.
    """
    prompt = "What do you see in this image? Classify the main object or
scene."

    chatHistory = []

    # If it's base64 data, use inlineData. Otherwise, treat as text
description.
    if image_description_or_base64_data.startswith("data:image"): # Simple
check for base64
        # Extract base64 part
        base64_data = image_description_or_base64_data.split(",")[1]
        chatHistory.push({
            role: "user",
            parts: [
                { text: prompt },
                {
                    inlineData: {
                        mimeType: "image/png", # Assuming PNG for dummy image
                        data: base64_data
                    }
                }
            ]
        });
    else:
        chatHistory.push({ role: "user", parts: [{ text: f"{prompt}\n\nImage
Description: {image_description_or_base64_data}" }] });

    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    print(f"\nClassifying image...")
    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            classification = result.candidates[0].content.parts[0].text;
            print(f"Image Classification:\n{classification}")
            return classification
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during image classification: {e}"

async def retrieve_similar_images_conceptual(query_description):
    """
    Conceptually demonstrates image retrieval based on a text query.
    In a real system, this would involve searching an image database
    with pre-computed embeddings.
    """
    prompt = f"Given the following query, describe what kind of images would
be similar. Imagine you have a vast collection of images. Query:
'{query_description}'"

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    print(f"\nRetrieving conceptual images for query: '{query_description}'")
    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            retrieval_description =
result.candidates[0].content.parts[0].text;
            print(f"Conceptual Image Retrieval:\n{retrieval_description}")
            return retrieval_description
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during conceptual image retrieval: {e}"

async def main():
    # Example 1: Image Classification (using dummy base64 image data)
    dummy_image_base64 = get_image_base64("dummy.png")
    if dummy_image_base64:
        await classify_image(f"data:image/png;base64,{dummy_image_base64}")

    # Example 2: Image Classification (using a text description as input to
the model)
    await classify_image("A large, fluffy white cat sleeping on a sunny
windowsill.")

    # Example 3: Conceptual Image Retrieval
    await retrieve_similar_images_conceptual("A serene landscape with a river
and mountains at sunset.")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: For image classification:

1. A base64 encoded image (simulated with a dummy red image for this example).
2. A text description of an image: `"A large, fluffy white cat sleeping on a sunny windowsill."`

For image retrieval: A text query: `"A serene landscape with a river and mountains at sunset."`

Expected Output: For image classification (dummy red image):

```
Image Classification:
This image appears to be a simple red rectangle.
```

For image classification (text description):

```
Image Classification:
A cat, specifically a large, white, fluffy one, is sleeping on a windowsill.
The scene suggests a peaceful, sunny environment.
```

For image retrieval:

```
Conceptual Image Retrieval:
Images that would be similar to "A serene landscape with a river and
mountains at sunset" would likely feature:
-   **Natural elements:** Prominent mountains, a flowing river or calm body
of water.
-   **Lighting:** Warm, soft light characteristic of sunset, with hues of
orange, pink, purple, and deep blue in the sky.
-   **Atmosphere:** A tranquil, peaceful, and possibly majestic or awe-
inspiring mood.
-   **Composition:** Wide shots capturing the vastness of the landscape,
perhaps with reflections in the water.
```

The exact output will vary, but it should correctly identify the image content or describe relevant images.

Program 4: Application of Multi-modal GANs

Title: Application of Multi-modal GANs

Aim: To understand the concept and potential applications of Multi-modal Generative Adversarial Networks (GANs), which can generate outputs across different modalities (e.g., generating images from text descriptions, or text from images).

Procedure:

1. **Understand GANs:** Review the basic architecture of GANs (Generator and Discriminator).
2. **Understand Multi-modality:** Grasp how GANs can be extended to handle multiple data types (e.g., text, image, audio).
3. **Conceptual Application:** Choose a multi-modal generation task (e.g., Text-to-Image synthesis).
4. **Simulate Generation:** Describe how such a system would take an input from one modality and generate an output in another. Since direct GAN training and inference are computationally intensive and beyond this environment, we will simulate the output using a multi-modal generative model like `gemini-2.0-flash`.

Source Code (Conceptual Python using `gemini-2.0-flash` for multi-modal generation simulation):

```
# This code conceptually demonstrates the application of multi-modal GANs
# by simulating text-to-image generation using a multi-modal model.
# Actual GAN training and inference are complex and require specialized
frameworks.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def generate_image_from_text_conceptual(text_description):
    """
    Conceptually generates an image from a text description, simulating a
multi-modal GAN.
    This uses imagen-3.0-generate-002 to generate an image.
    """
    print(f"\nGenerating image from text: '{text_description}'")

    payload = { instances: { prompt: text_description }, parameters: {
"sampleCount": 1} };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/imagen-
3.0-generate-002:predict?key=${API_KEY}`;

    try:
        # Display a loading indicator
        print("Generating image... Please wait.")

        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();

        if (result.predictions && result.predictions.length > 0 &&
result.predictions[0].bytesBase64Encoded) {
```

```
            image_url =
`data:image/png;base64,${result.predictions[0].bytesBase64Encoded}`;
            print("Image generated successfully. Displaying image:")
            # In a real UI, you would render this image_url in an <img> tag.
            # For console output, we'll just print a message.
            print(f"Image URL (Base64 encoded): [Image Data Omitted for
brevity]")
            return image_url
        } else {
            print("Error: No image content found in response.")
            return "Error: No image content found."
        }
    except Exception as e:
        print(f"An error occurred during image generation: {e}")
        return f"Error during image generation: {e}"

async def main():
    # Example: Text-to-Image Generation
    await generate_image_from_text_conceptual("A futuristic city at sunset
with flying cars and towering skyscrapers.")
    await generate_image_from_text_conceptual("A watercolor painting of a
serene forest with a hidden waterfall.")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: Text descriptions for image generation. Example 1: `"A futuristic city at sunset with flying cars and towering skyscrapers."` Example 2: `"A watercolor painting of a serene forest with a hidden waterfall."`

Expected Output: For Example 1: A generated image depicting a futuristic city scene with the described elements. For Example 2: A generated image resembling a watercolor painting of a forest with a waterfall. The output will be a base64 encoded image URL, which would be rendered visually in a proper UI.

Program 5: Applications using Autoencoding Variational Bayes

Title: Applications using Autoencoding Variational Bayes (Variational Autoencoders - VAEs)

Aim: To understand the principles of Variational Autoencoders (VAEs) and their applications in generative modeling, particularly for tasks like data generation, anomaly detection, and latent space manipulation.

Procedure:

1. **Understand Autoencoders:** Review the basic concept of autoencoders (encoder-decoder architecture for dimensionality reduction).
2. **Understand VAEs:** Learn how VAEs introduce a probabilistic approach to the latent space, allowing for sampling and generation of new data.
3. **Conceptual Application (Image Generation):** Focus on image generation as a common application.
4. **Simulate Latent Space Sampling:** Describe how a VAE would sample from its learned latent distribution to generate new data. Since direct VAE implementation is complex, we will simulate the *effect* of VAE-like generation using a general generative model.

Source Code (Conceptual Python using `imagen-3.0-generate-002` to simulate VAE-like generation):

```
# This code conceptually demonstrates applications of Variational
Autoencoders (VAEs)
# by simulating image generation, which is a common VAE application.
# Actual VAE implementation involves training a specific neural network
architecture.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def generate_image_with_vae_concept(style_description):
    """
    Conceptually generates an image by sampling from a latent space,
    simulating the generative capability of a VAE.
    This uses imagen-3.0-generate-002 to generate an image based on a style.
    """
    prompt = f"Generate a novel image of a {style_description}."
    print(f"\nSimulating VAE-like generation for: '{style_description}'")

    payload = { instances: { prompt: prompt }, parameters: { "sampleCount":
1} };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/imagen-
3.0-generate-002:predict?key=${API_KEY}`;

    try:
        print("Generating image... Please wait.")
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();

        if (result.predictions && result.predictions.length > 0 &&
result.predictions[0].bytesBase64Encoded) {
            image_url =
`data:image/png;base64,${result.predictions[0].bytesBase64Encoded}`;
```

```
            print("Image generated successfully. Displaying image:")
            print(f"Image URL (Base64 encoded): [Image Data Omitted for
brevity]")
            return image_url
        } else {
            print("Error: No image content found in response.")
            return "Error: No image content found."
        }
    except Exception as e:
        print(f"An error occurred during image generation: {e}")
        return f"Error during image generation: {e}"

async def main():
    # Example 1: Generate a new face (common VAE application)
    await generate_image_with_vae_concept("realistic human face, diverse
features")

    # Example 2: Generate a new piece of abstract art
    await generate_image_with_vae_concept("abstract painting with geometric
shapes and vibrant colors")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: Descriptions of the desired generated output, representing a "sampling" from a conceptual latent space. Example 1: `"realistic human face, diverse features"` Example 2: `"abstract painting with geometric shapes and vibrant colors"`

Expected Output: For Example 1: A generated image of a novel, realistic human face. For Example 2: A generated image of a new abstract painting. The output will be a base64 encoded image URL.

Program 6: Generate an Application Using Conditional Generative Models

Title: Generate an Application Using Conditional Generative Models

Aim: To understand and apply conditional generative models, which can generate data conditioned on specific input attributes or conditions (e.g., generating a specific type of image, or text in a particular style).

Procedure:

1. **Understand Conditional Generation:** Learn how generative models can be controlled by providing additional input conditions (e.g., class labels, text descriptions).
2. **Choose a Conditional Task:** Select a task that benefits from conditional generation (e.g., Text-to-Image generation where the text is the condition, or generating text in a specific tone).
3. **Formulate Conditions:** Define the conditions under which the generation should occur.
4. **Utilize Conditional Model:** Use a generative model that supports conditional inputs. We will use `imagen-3.0-generate-002` for image generation and `gemini-2.0-flash` for text generation, both of which are conditional by nature (conditioned on the prompt).

Source Code (Conceptual Python using `imagen-3.0-generate-002` for image and `gemini-2.0-flash` for text):

```
# This code demonstrates generating an application using conditional
generative models.
# We'll show two examples: conditional image generation and conditional text
generation.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def generate_conditional_image(description, style):
    """
    Generates an image conditioned on both content description and style.
    """
    prompt = f"{description}, in the style of a {style}."
    print(f"\nGenerating conditional image: '{prompt}'")

    payload = { instances: { prompt: prompt }, parameters: { "sampleCount":
1} };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/imagen-
3.0-generate-002:predict?key=${API_KEY}`;

    try:
        print("Generating image... Please wait.")
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();

        if (result.predictions && result.predictions.length > 0 &&
result.predictions[0].bytesBase64Encoded) {
            image_url =
`data:image/png;base64,${result.predictions[0].bytesBase64Encoded}`;
            print("Image generated successfully. Displaying image:")
            print(f"Image URL (Base64 encoded): [Image Data Omitted for
brevity]")
```

```
            return image_url
        } else {
            print("Error: No image content found in response.")
            return "Error: No image content found."
        }
    except Exception as e:
        print(f"An error occurred during image generation: {e}")
        return f"Error during image generation: {e}"

async def generate_conditional_text(topic, tone):
    """
    Generates text conditioned on a topic and a specific tone.
    """
    prompt = f"Write a short paragraph about '{topic}' in a {tone} tone."
    print(f"\nGenerating conditional text: '{prompt}'")

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            generated_text = result.candidates[0].content.parts[0].text;
            print(f"Generated Text:\n{generated_text}")
            return generated_text
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during text generation: {e}"

async def main():
    # Example 1: Conditional Image Generation
    await generate_conditional_image("a majestic lion", "oil painting")
    await generate_conditional_image("a cozy living room", "impressionistic
art")

    # Example 2: Conditional Text Generation
    await generate_conditional_text("the benefits of exercise",
"motivational")
    await generate_conditional_text("the history of AI", "academic")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: For conditional image generation:

1. Description: `"a majestic lion"`, Style: `"oil painting"`
2. Description: `"a cozy living room"`, Style: `"impressionistic art"`

For conditional text generation:

1. Topic: `"the benefits of exercise"`, Tone: `"motivational"`
2. Topic: `"the history of AI"`, Tone: `"academic"`

Expected Output: For conditional image generation:

1. A generated image of a lion in the style of an oil painting.
2. A generated image of a living room in an impressionistic art style.

For conditional text generation:

1. A paragraph about the benefits of exercise written in a motivational tone.
2. A paragraph about the history of AI written in an academic tone.

Program 7: Implement Conditional Generation

Title: Implement Conditional Generation (General Concept)

Aim: To practically implement and demonstrate the core mechanism of conditional generation using a simple example, showcasing how an output can be influenced by a given input condition. This builds upon the previous program by focusing on the implementation aspect.

Procedure:

1. **Define a Simple Conditional Task:** Choose a straightforward task where output depends on a clear condition (e.g., generating a sentence based on a sentiment, or a number sequence based on a starting point).
2. **Design Conditional Logic:** Implement logic that takes the condition as input and guides the generation process.
3. **Utilize a Generative Model:** Use a generative model (like `gemini-2.0-flash`) and craft prompts that explicitly include the condition to guide its output.

Source Code (Conceptual Python using `gemini-2.0-flash` for conditional text generation):

```
# This code implements a general concept of conditional generation using a
text-based model.
# The 'condition' is embedded directly into the prompt.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def generate_story_with_sentiment(genre, sentiment):
    """
    Generates a short story conditioned on a specified genre and sentiment.
    """
    prompt = f"Write a very short {genre} story with a {sentiment}
sentiment."
    print(f"\nGenerating story: '{prompt}'")

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            generated_story = result.candidates[0].content.parts[0].text;
            print(f"Generated Story:\n{generated_story}")
            return generated_story
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during story generation: {e}"

async def main():
    # Example 1: Conditional story generation - positive fantasy
    await generate_story_with_sentiment("fantasy", "positive")

    # Example 2: Conditional story generation - negative sci-fi
    await generate_story_with_sentiment("sci-fi", "negative")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: For story generation:

1. Genre: `"fantasy"`, Sentiment: `"positive"`
2. Genre: `"sci-fi"`, Sentiment: `"negative"`

Expected Output: For Example 1:

```
Generated Story:
Elara, the young sorceress, finally mastered the ancient spell. A shimmering
aura enveloped her, and the withered garden burst into vibrant bloom, filling
the air with the sweet scent of triumph and hope.
```

For Example 2:

```
Generated Story:
The derelict spaceship drifted through the void, its lights flickering
erratically. Commander Vex felt a chill deeper than the vacuum of space as
the last of the crew succumbed to the unknown contagion, leaving him utterly
alone.
```

The output will be a short story matching the specified genre and sentiment.

Program 8: Develop Fine-grained Control in 3D Printing

Title: Develop Fine-grained Control in 3D Printing (Conceptual with AI)

Aim: To conceptually understand how generative AI techniques, particularly those capable of precise conditional generation, could be applied to achieve fine-grained control in 3D printing, from material properties to intricate geometries.

Procedure:

1. **Understand 3D Printing Parameters:** Research key parameters in 3D printing that influence the final product (e.g., infill density, layer height, material composition, support structures).
2. **Identify AI's Role:** Recognize how AI could optimize these parameters or even generate novel designs based on desired properties.
3. **Conceptualize AI-driven Design:** Imagine an AI system that takes high-level user requirements and generates precise 3D models or G-code instructions.
4. **Simulate AI Response:** Use a generative AI model (`gemini-2.0-flash`) to simulate the AI's response to a request for a highly controlled 3D print, detailing the parameters it would "generate."

Source Code (Conceptual Python using `gemini-2.0-flash` to describe 3D printing parameters):

```
# This code conceptually demonstrates how AI could provide fine-grained
control in 3D printing.
# It simulates an AI generating detailed printing parameters based on user
requirements.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def generate_3d_print_parameters(object_description,
desired_properties):
    """
    Simulates an AI generating detailed 3D printing parameters based on user
input.
    """
    prompt = f"Imagine you are an advanced AI for 3D printing. Given the
object '{object_description}' and desired properties '{desired_properties}',
generate a detailed set of 3D printing parameters (e.g., material, infill,
layer height, nozzle temperature, support structure) that would achieve these
properties. Explain your reasoning."
    print(f"\nGenerating 3D printing parameters for: '{object_description}'
with properties '{desired_properties}'")

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            parameters = result.candidates[0].content.parts[0].text;
            print(f"Generated 3D Printing Parameters:\n{parameters}")
            return parameters
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during parameter generation: {e}"

async def main():
    # Example 1: Generate parameters for a strong, lightweight bracket
    await generate_3d_print_parameters(
        "a small structural bracket",
        "high strength, lightweight, smooth finish"
    )

    # Example 2: Generate parameters for a flexible, intricate phone case
    await generate_3d_print_parameters(
        "a custom phone case",
        "flexible, intricate design, durable"
    )

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input:

1. Object: `"a small structural bracket"`, Desired Properties: `"high strength, lightweight, smooth finish"`
2. Object: `"a custom phone case"`, Desired Properties: `"flexible, intricate design, durable"`

Expected Output: For Example 1:

```
Generated 3D Printing Parameters:
To achieve a high-strength, lightweight, and smooth-finish small structural
bracket, the AI would generate the following parameters:

* **Material:** Carbon Fiber Reinforced Nylon (e.g., PAHT-CF) - offers
excellent strength-to-weight ratio.
* **Infill Density:** 80-90% (Gyroid or Cubic infill pattern) - ensures high
strength while optimizing for weight.
* **Layer Height:** 0.12mm - for a smoother surface finish.
* **Nozzle Temperature:** 260-270°C (specific to material)
* **Bed Temperature:** 80-90°C
* **Print Speed:** 40-60 mm/s - a slower speed for better adhesion and
detail.
* **Support Structure:** Tree supports (minimal contact) - to ensure smooth
underside surfaces without excessive material.
* **Shells/Perimeters:** 4-5 - for increased outer wall strength.

Reasoning: The choice of carbon fiber nylon provides inherent strength and
stiffness. A high infill density with a strong pattern maximizes load-bearing
capacity. A low layer height contributes to surface smoothness. Optimized
```

```
temperatures and speeds ensure proper material extrusion and layer bonding.
Tree supports are chosen for their ease of removal and minimal impact on the
surface finish.
```

For Example 2:

```
Generated 3D Printing Parameters:
To achieve a flexible, intricate, and durable custom phone case, the AI would
generate the following parameters:

* **Material:** Thermoplastic Polyurethane (TPU) - known for its flexibility,
durability, and impact resistance.
* **Infill Density:** 15-25% (Grid or Honeycomb infill pattern) - provides
flexibility while maintaining structural integrity.
* **Layer Height:** 0.2mm - a slightly larger layer height is acceptable for
flexible parts and can speed up printing.
* **Nozzle Temperature:** 220-230°C (specific to TPU)
* **Bed Temperature:** 50-60°C (or off, depending on TPU type)
* **Print Speed:** 20-30 mm/s - slower speeds are crucial for flexible
filaments to prevent tangling and ensure good extrusion.
* **Support Structure:** Minimal or none, depending on the design's
overhangs. If needed, tree supports with low density.
* **Retraction Settings:** Optimized for TPU (often lower retraction distance
and speed to prevent clogs).
* **Flow Rate:** Calibrated for flexibility, ensuring proper extrusion
without over-extrusion.

Reasoning: TPU is the ideal material for flexibility. A lower infill density
allows for more deformation. Slower speeds and optimized retraction are
necessary due to TPU's elastic nature. The intricate design will benefit from
the material's ability to handle complex geometries without breaking.
```

The output will be a detailed description of 3D printing parameters and the reasoning behind them.

Program 9: Generate an Application Using Meta Learning

Title: Generate an Application Using Meta Learning

Aim: To understand the concept of meta-learning (learning to learn) and demonstrate how it can be applied to create an application that can quickly adapt to new tasks or environments with minimal new data.

Procedure:

1. **Understand Meta Learning:** Grasp the core idea of meta-learning, where a model learns an "initialization" or "learning strategy" that allows it to learn new tasks efficiently.
2. **Choose a Meta-Learning Scenario:** Select a scenario where rapid adaptation is beneficial (e.g., few-shot learning for classification or regression).
3. **Conceptualize Meta-Learning Application:** Describe how a meta-learning algorithm (like MAML or Reptile) would be used to train a model that can then quickly adapt to new, unseen tasks.
4. **Simulate Meta-Learning Adaptation:** Use a generative AI model (`gemini-2.0-flash`) to simulate the *outcome* of a meta-learning process, showing how it adapts to a new, simple task.

Source Code (Conceptual Python using `gemini-2.0-flash` to simulate meta-learning adaptation):

```
# This code conceptually demonstrates an application using meta-learning.
# It simulates a model adapting to a new, simple classification task
# after having been "meta-trained" to learn quickly.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def meta_learn_new_task_conceptual(new_task_description,
few_shot_examples):
    """
    Simulates a meta-learned model adapting to a new classification task
    with only a few examples.
    """
    prompt = f"Imagine you are a meta-learned AI designed for rapid
adaptation. Your new task is: '{new_task_description}'. Here are a few
examples:\n"
    for example_input, example_output in few_shot_examples:
        prompt += f"- Input: '{example_input}', Output: '{example_output}'\n"
    prompt += "Based on these examples, classify the following new input:\n"
    prompt += "Input: 'Is the sky blue?'\nOutput:"

    print(f"\nSimulating meta-learning adaptation for new task:
'{new_task_description}'")

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            adaptation_result = result.candidates[0].content.parts[0].text;
            print(f"Meta-learning Adaptation Result:\n{adaptation_result}")
            return adaptation_result
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during adaptation simulation: {e}"

async def main():
    # Example: Meta-learning for a simple "Yes/No Question" classification
task
    new_task = "Classify if a given sentence is a question that can be
answered with a simple 'Yes' or 'No'."
    few_shot_data = [
        ("Is the sun hot?", "Yes"),
        ("Do birds fly?", "Yes"),
        ("What is your name?", "No"),
        ("Are you hungry?", "Yes")
    ]

    await meta_learn_new_task_conceptual(new_task, few_shot_data)

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: New task description: `"Classify if a given sentence is a question that can be answered with a simple 'Yes' or 'No'."` Few-shot examples:

- Input: `"Is the sun hot?"`, Output: `"Yes"`
- Input: `"Do birds fly?"`, Output: `"Yes"`
- Input: `"What is your name?"`, Output: `"No"`
- Input: `"Are you hungry?"`, Output: `"Yes"` New input to classify: `"Is the sky blue?"`

Expected Output:

```
Meta-learning Adaptation Result:
Yes
```

The output should correctly classify the new input based on the few-shot examples provided.

Program 10: Adapt a Generative Model from MNIST to SVHN Using Meta Learning

Title: Adapt a Generative Model from MNIST to SVHN Using Meta Learning

Aim: To conceptually demonstrate how a generative model (e.g., a VAE or GAN) initially trained on the MNIST dataset can be rapidly adapted to generate images from a different domain, like SVHN (Street View House Numbers), using meta-learning techniques.

Procedure:

1. **Understand Domain Adaptation:** Recognize the challenge of transferring models between different data distributions.
2. **Understand Meta-Learning for Adaptation:** Learn how meta-learning can enable a model to quickly learn the characteristics of a new domain with limited data.
3. **Conceptual Model (Meta-Generative Model):** Imagine a generative model that has been meta-trained on various digit datasets (including MNIST) such that it can quickly adapt its generation capabilities to a new digit dataset (SVHN).
4. **Simulate Adaptation and Generation:** Use a generative AI model (`imagen-3.0-generate-002`) to simulate the *result* of such an adaptation, generating SVHN-like digits after being "shown" a few examples.

Source Code (Conceptual Python using `imagen-3.0-generate-002` for image generation):

```
# This code conceptually demonstrates adapting a generative model from MNIST
to SVHN
# using meta-learning. It simulates the generation of SVHN-like digits.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def adapt_and_generate_svhn_conceptual(svhn_style_description):
    """
    Simulates a generative model adapted to SVHN, generating new SVHN-like
digits.
    """
    prompt = f"Generate a new image of a digit, specifically a house number,
in the style of the Street View House Numbers (SVHN) dataset. The image
should be clear and realistic for a house number. Example style:
{svhn_style_description}"
    print(f"\nSimulating adaptation to SVHN and generating:
'{svhn_style_description}'")

    payload = { instances: { prompt: prompt }, parameters: { "sampleCount":
1} };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/imagen-
3.0-generate-002:predict?key=${API_KEY}`;

    try:
        print("Generating image... Please wait.")
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();

        if (result.predictions && result.predictions.length > 0 &&
result.predictions[0].bytesBase64Encoded) {
```

```
            image_url =
`data:image/png;base64,${result.predictions[0].bytesBase64Encoded}`;
            print("Image generated successfully. Displaying image:")
            print(f"Image URL (Base64 encoded): [Image Data Omitted for
brevity]")
            return image_url
        } else {
            print("Error: No image content found in response.")
            return "Error: No image content found."
        }
    except Exception as e:
        print(f"An error occurred during image generation: {e}")
        return f"Error during image generation: {e}"

async def main():
    # Example: Generate an SVHN-style digit
    await adapt_and_generate_svhn_conceptual("a slightly tilted, brightly lit
'7' on a textured background")
    await adapt_and_generate_svhn_conceptual("a clear, bold '2' with some
surrounding street view elements")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: Descriptions indicating the desired SVHN style. Example 1: `"a slightly tilted, brightly lit '7' on a textured background"` Example 2: `"a clear, bold '2' with some surrounding street view elements"`

Expected Output: For Example 1: A generated image of a digit '7' resembling the SVHN dataset style. For Example 2: A generated image of a digit '2' with SVHN-like characteristics. The output will be a base64 encoded image URL.

Program 11: Develop Applications Using RL Algorithm

Title: Develop Applications Using Reinforcement Learning (RL) Algorithm

Aim: To understand the fundamentals of Reinforcement Learning (RL) and conceptually develop an application where an agent learns to achieve a goal through trial and error, interacting with an environment.

Procedure:

1. **Understand RL Basics:** Learn about agents, environments, states, actions, rewards, and policies.
2. **Define a Simple Problem:** Choose a simple problem that can be framed as an RL task (e.g., a simple navigation task, or a game).
3. **Conceptualize RL Agent:** Describe how an RL agent would learn to solve this problem (e.g., Q-learning, Policy Gradients).
4. **Simulate RL Agent's Behavior:** Use a generative AI model (`gemini-2.0-flash`) to simulate the *outcome* of an RL agent's learning process, describing its learned policy or actions.

Source Code (Conceptual Python using `gemini-2.0-flash` to simulate RL agent's learned behavior):

```
# This code conceptually demonstrates developing an application using an RL
algorithm.
# It simulates an RL agent learning to navigate a simple grid world.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def simulate_rl_navigation(environment_description, goal_description):
    """
    Simulates an RL agent learning to navigate an environment to reach a
goal.
    """
    prompt = f"Imagine an RL agent in a simple grid-world environment
described as: '{environment_description}'. The agent's goal is to
'{goal_description}'. Describe the optimal sequence of actions the agent
would learn to take from the starting point to reach the goal, and explain
why these actions are optimal based on rewards and penalties."
    print(f"\nSimulating RL navigation for: '{environment_description}' to
'{goal_description}'")

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            rl_behavior = result.candidates[0].content.parts[0].text;
            print(f"Simulated RL Agent Behavior:\n{rl_behavior}")
            return rl_behavior
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during RL simulation: {e}"

async def main():
    # Example: RL agent learning to navigate a maze
    await simulate_rl_navigation(
        "A 5x5 grid maze with walls at (1,2), (2,2), (3,2). Start at (0,0).",
        "reach the treasure at (4,4)"
    )

    # Example: RL agent learning to balance a pole (simplified description)
    await simulate_rl_navigation(
        "A pole balancing on a cart. Actions: move cart left/right.",
        "keep the pole upright for as long as possible"
    )

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input:

1. Environment: `"A 5x5 grid maze with walls at (1,2), (2,2), (3,2). Start at (0,0)."`, Goal: `"reach the treasure at (4,4)"`
2. Environment: `"A pole balancing on a cart. Actions: move cart left/right."`, Goal: `"keep the pole upright for as long as possible"`

Expected Output: For Example 1:

```
Simulated RL Agent Behavior:
Given a 5x5 grid maze with walls at (1,2), (2,2), (3,2), and starting at
(0,0) with the goal of reaching (4,4), an optimal RL agent would learn the
following sequence of actions:

1.  **Move Right** (from (0,0) to (0,1))
2.  **Move Right** (from (0,1) to (0,2))
3.  **Move Right** (from (0,2) to (0,3))
4.  **Move Right** (from (0,3) to (0,4))
5.  **Move Down** (from (0,4) to (1,4))
6.  **Move Down** (from (1,4) to (2,4))
7.  **Move Down** (from (2,4) to (3,4))
8.  **Move Down** (from (3,4) to (4,4)) - Goal Reached!

Reasoning: The agent would learn to avoid the walls by moving along the top
edge of the maze, then descending to the goal. Each "move" action would
likely have a small negative reward (cost of movement), while reaching the
goal would have a large positive reward. Colliding with a wall would incur a
large negative penalty. Through exploration and exploitation, the agent would
converge on this shortest path that maximizes cumulative reward.
```

For Example 2:

```
Simulated RL Agent Behavior:
For a pole balancing on a cart, where the goal is to keep the pole upright as
long as possible, an RL agent would learn a policy that involves:

* **Observing Pole Angle and Angular Velocity:** The agent would continuously
monitor the pole's deviation from vertical and how fast it's falling.
* **Applying Corrective Forces:** When the pole starts to lean right, the
agent would apply a force to move the cart right, effectively "catching" the
pole and bringing it back towards the center. Conversely, if the pole leans
left, the cart moves left.
* **Anticipatory Movements:** A sophisticated agent would learn to anticipate
the pole's fall and initiate movements before the deviation becomes too
large, demonstrating proactive control.
* **Balancing Act:** The optimal policy would involve a series of small,
precise left and right movements of the cart, constantly adjusting to
maintain the pole's equilibrium.

Reasoning: The agent receives a positive reward for every timestep the pole
remains upright and a large negative reward (penalty) if the pole falls.
Through repeated trials, it learns which actions (moving left or right) in
which states (pole angle, angular velocity) lead to higher cumulative
rewards, effectively discovering the physics of balancing the pole.
```

The output will describe the learned optimal actions or policy for the given RL problem.

Program 12: Fine-tune a Pre-trained Transformer Model on a Few-shot Text Classification Problem Using a Meta Learning Approach.

Title: Fine-tune a Pre-trained Transformer Model on a Few-shot Text Classification Problem Using a Meta Learning Approach

Aim: To understand and conceptually implement how meta-learning can enable a pre-trained transformer model (like BERT or GPT) to quickly adapt and perform well on new text classification tasks with very limited labeled data (few-shot learning).

Procedure:

1. **Understand Transformer Models:** Review the architecture and pre-training objectives of transformer models.
2. **Understand Few-shot Learning:** Grasp the challenge of training models with very few examples per class.
3. **Understand Meta-Learning for Few-shot:** Learn how meta-learning (e.g., MAML, ProtoNet) can train a model to learn a good initialization or a metric space for rapid adaptation.
4. **Conceptual Application:** Describe how a meta-learned transformer would be fine-tuned on a new, few-shot text classification task.
5. **Simulate Few-shot Adaptation:** Use a generative AI model (`gemini-2.0-flash`) to simulate the *outcome* of such an adaptation, performing classification on new examples after being "trained" on a few.

Source Code (Conceptual Python using `gemini-2.0-flash` to simulate few-shot classification):

```
# This code conceptually demonstrates fine-tuning a transformer model on a
few-shot
# text classification problem using a meta-learning approach.
# It simulates the classification of new text based on a few examples
provided.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-2.0-flash or
imagen-3.0-generate-002, provide an API key here. Otherwise, leave this as-
is.

async def few_shot_text_classify_conceptual(task_name, few_shot_examples,
text_to_classify):
    """
    Simulates a meta-learned transformer performing few-shot text
classification.
    """
    prompt = f"Imagine you are a transformer model that has been meta-learned
for few-shot text classification. Your task is to classify text for
'{task_name}'. Here are a few examples:\n"
    for example_text, example_label in few_shot_examples:
        prompt += f"- Text: '{example_text}', Label: '{example_label}'\n"
    prompt += f"Based on these examples, classify the following text:\nText:
'{text_to_classify}'\nLabel:"

    print(f"\nPerforming few-shot text classification for '{task_name}' on
text: '{text_to_classify}'")

    chatHistory = []
    chatHistory.push({ role: "user", parts: [{ text: prompt }] });
    payload = { contents: chatHistory };
    apiUrl = `https://generativelanguage.googleapis.com/v1beta/models/gemini-
2.0-flash:generateContent?key=${API_KEY}`;

    try:
        response = await fetch(apiUrl, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(payload)
        });
        result = await response.json();
        if (result.candidates && result.candidates.length > 0 &&
            result.candidates[0].content &&
result.candidates[0].content.parts &&
            result.candidates[0].content.parts.length > 0) {
            classification_result =
result.candidates[0].content.parts[0].text;
            print(f"Classification Result:\n{classification_result}")
            return classification_result
        } else {
            print("Error: No content found in response.")
            return "Error: No content found."
        }
    except Exception as e:
        print(f"An error occurred: {e}")
        return f"Error during classification simulation: {e}"

async def main():
    # Example: Few-shot sentiment classification
    sentiment_task = "Sentiment Analysis (Positive/Negative)"
    sentiment_examples = [
        ("This product is amazing!", "Positive"),
        ("I hate this movie.", "Negative"),
        ("The weather is great today.", "Positive")
    ]
    await few_shot_text_classify_conceptual(sentiment_task,
sentiment_examples, "I am so happy with my new phone!")
    await few_shot_text_classify_conceptual(sentiment_task,
sentiment_examples, "This is the worst experience ever.")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: Task: `"Sentiment Analysis (Positive/Negative)"` Few-shot examples:

- Text: `"This product is amazing!"`, Label: `"Positive"`
- Text: `"I hate this movie."`, Label: `"Negative"`
- Text: `"The weather is great today."`, Label: `"Positive"` Text to classify:

1. `"I am so happy with my new phone!"`
2. `"This is the worst experience ever."`

Expected Output: For Text 1:

```
Classification Result:
Positive
```

For Text 2:

```
Classification Result:
```

```
Negative
```

The output should correctly classify the sentiment based on the few-shot examples.

Program 13: Implement RL Algorithm

Title: Implement Reinforcement Learning (RL) Algorithm (Specific Example)

Aim: To implement a basic Reinforcement Learning algorithm (e.g., Q-learning or SARSA) for a simple, discrete environment, demonstrating the agent's learning process and policy convergence.

Procedure:

1. **Define Environment:** Create a simple grid-world environment with states, actions, and rewards.
2. **Initialize Q-table:** Create a Q-table (state-action value function) and initialize it.
3. **Implement Exploration-Exploitation:** Use an epsilon-greedy strategy for action selection.
4. **Implement Q-learning Update Rule:** Apply the Q-learning update rule to iteratively improve Q-values.
5. **Train Agent:** Run multiple episodes to train the agent.
6. **Evaluate Policy:** After training, evaluate the learned policy by letting the agent navigate the environment.

Source Code (Conceptual Python - Q-Learning Pseudocode/Description):

```
# This code provides a conceptual implementation outline for a Q-learning
algorithm.
# Actual execution requires a full RL framework or a custom environment
simulation.

# Placeholder for API key (not used in this conceptual RL code, but kept for
consistency)
API_KEY = ""

def initialize_q_table(num_states, num_actions):
    """Initializes the Q-table with zeros."""
    return [[0.0 for _ in range(num_actions)] for _ in range(num_states)]

def choose_action(state, q_table, epsilon, num_actions):
    """Epsilon-greedy action selection."""
    if random.uniform(0, 1) < epsilon:
        return random.randint(0, num_actions - 1)  # Explore
    else:
        return q_table[state].index(max(q_table[state])) # Exploit

def update_q_table(q_table, state, action, reward, next_state, alpha, gamma):
    """Q-learning update rule."""
    old_value = q_table[state][action]
    next_max = max(q_table[next_state])

    new_value = old_value + alpha * (reward + gamma * next_max - old_value)
    q_table[state][action] = new_value

def train_q_learning(env, num_episodes, alpha, gamma, epsilon_start,
epsilon_end, epsilon_decay):
    """
    Conceptual training loop for Q-learning.
    'env' would be an environment object with methods like reset() and
step().
    """
    num_states = env.observation_space.n
    num_actions = env.action_space.n
    q_table = initialize_q_table(num_states, num_actions)
    epsilon = epsilon_start

    print("\nStarting Q-learning training...")
    for episode in range(num_episodes):
        state = env.reset() # Get initial state
        done = False
        total_reward = 0

        while not done:
            action = choose_action(state, q_table, epsilon, num_actions)
            next_state, reward, done, _ = env.step(action) # Take action in
env

            update_q_table(q_table, state, action, reward, next_state, alpha,
gamma)

            state = next_state
            total_reward += reward

        epsilon = max(epsilon_end, epsilon * epsilon_decay) # Decay epsilon

        if (episode + 1) % 100 == 0:
            print(f"Episode {episode + 1}/{num_episodes}, Total Reward:
{total_reward}, Epsilon: {epsilon:.4f}")

    print("Q-learning training complete.")
    return q_table

def evaluate_policy(env, q_table):
    """
    Evaluates the learned policy by letting the agent navigate the
environment.
    """
    state = env.reset()
    done = False
    total_reward = 0
    path = [state]

    print("\nEvaluating learned policy:")
    while not done:
        action = q_table[state].index(max(q_table[state])) # Always exploit
        next_state, reward, done, _ = env.step(action)
        state = next_state
        total_reward += reward
        path.append(state)

    print(f"Final Path: {path}")
    print(f"Total Reward: {total_reward}")
    return path, total_reward

# To run this, you would need a concrete 'env' implementation (e.g., OpenAI
Gym's FrozenLake)
# and then call functions like:
# import gym
# env = gym.make("FrozenLake-v1")
# q_table = train_q_learning(env, num_episodes=1000, alpha=0.1, gamma=0.99,
epsilon_start=1.0, epsilon_end=0.01, epsilon_decay=0.995)
# evaluate_policy(env, q_table)
```

Input: A simple environment definition (e.g., a 4x4 FrozenLake-like grid):

- States: 0-15 (representing grid cells)
- Actions: 0 (Left), 1 (Down), 2 (Right), 3 (Up)

- Rewards: +1 for reaching goal, 0 for normal moves, -1 for falling into a hole.
- Hyperparameters: `num_episodes`, `alpha` (learning rate), `gamma` (discount factor), `epsilon` (exploration rate).

Expected Output: During training, print statements showing episode number, total reward, and epsilon decay. After training, the `q_table` containing learned state-action values. During evaluation, the path taken by the agent and the total reward achieved, demonstrating its ability to reach the goal. Example (for a successful run on FrozenLake):

```
Starting Q-learning training...
...
Episode 900/1000, Total Reward: 1.0, Epsilon: 0.0150
Episode 1000/1000, Total Reward: 1.0, Epsilon: 0.0100
Q-learning training complete.

Evaluating learned policy:
Final Path: [0, 4, 8, 9, 10, 14, 15]
Total Reward: 1.0
```

Program 14: Implement Adversarial Training Methods

Title: Implement Adversarial Training Methods (Conceptual with GANs)

Aim: To understand and conceptually implement adversarial training, particularly in the context of Generative Adversarial Networks (GANs), where two neural networks (Generator and Discriminator) compete to improve each other.

Procedure:

1. **Understand GAN Architecture:** Review the Generator (creates fake data) and Discriminator (distinguishes real from fake) components.
2. **Understand Adversarial Loss:** Learn how the loss functions are designed to create a minimax game between the two networks.
3. **Conceptual Training Loop:** Describe the alternating training steps for the Discriminator and the Generator.
4. **Simulate Training Outcome:** Use a generative AI model (`imagen-3.0-generate-002`) to simulate the *result* of successful adversarial training, which is the generation of realistic data.

Source Code (Conceptual Python - GAN Training Loop Description):

```
# This code conceptually outlines the implementation of adversarial training
methods,
# specifically for a Generative Adversarial Network (GAN).
# Actual GAN implementation involves complex neural network architectures and
training.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-3.0-generate-002,
provide an API key here. Otherwise, leave this as-is.

async def simulate_gan_training_outcome(data_type_to_generate):
    """
    Simulates the outcome of successful GAN adversarial training:
    the generation of realistic data.
    """
    prompt = f"Imagine a Generative Adversarial Network (GAN) has been
successfully trained to generate realistic '{data_type_to_generate}'.
Describe the characteristics of the generated data, indicating its realism
and diversity."
    print(f"\nSimulating GAN training outcome for generating:
'{data_type_to_generate}'")

    # Use imagen-3.0-generate-002 to generate an image as a proxy for the
GAN's output
    if "image" in data_type_to_generate.lower():
        generation_prompt = f"Generate a realistic image of
{data_type_to_generate}."
        payload = { instances: { prompt: generation_prompt }, parameters: {
"sampleCount": 1} };
        apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/imagen-3.0-generate-
002:predict?key=${API_KEY}`;

        try:
            print("Generating image (simulating GAN output)... Please wait.")
            response = await fetch(apiUrl, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(payload)
            });
            result = await response.json();

            if (result.predictions && result.predictions.length > 0 &&
result.predictions[0].bytesBase64Encoded) {
                image_url =
`data:image/png;base64,${result.predictions[0].bytesBase64Encoded}`;
                print("Simulated GAN Output (Image generated successfully):")
                print(f"Image URL (Base64 encoded): [Image Data Omitted for
brevity]")
                return image_url
            } else {
                print("Error: No image content found in response.")
                return "Error: No image content found."
            }
        except Exception as e:
            print(f"An error occurred during image generation: {e}")
            return f"Error during image generation: {e}"
    else:
        # For non-image data, use gemini-2.0-flash to describe the outcome
        chatHistory = []
        chatHistory.push({ role: "user", parts: [{ text: prompt }] });
        payload = { contents: chatHistory };
        apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${API_KEY}`;

        try:
            response = await fetch(apiUrl, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(payload)
            });
            result = await response.json();
            if (result.candidates && result.candidates.length > 0 &&
                result.candidates[0].content &&
result.candidates[0].content.parts &&
                result.candidates[0].content.parts.length > 0) {
                description = result.candidates[0].content.parts[0].text;
                print(f"Simulated GAN Training Outcome:\n{description}")
                return description
            } else {
                print("Error: No content found in response.")
                return "Error: No content found."
            }
        except Exception as e:
            print(f"An error occurred: {e}")
            return f"Error during simulation: {e}"

async def main():
    # Example 1: Simulate GAN generating human faces
    await simulate_gan_training_outcome("human faces")

    # Example 2: Simulate GAN generating realistic text
    await simulate_gan_training_outcome("short news articles")

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input: Type of data to be generated by the GAN. Example 1: `"human faces"` Example 2: `"short news articles"`

Expected Output: For Example 1: A generated image of a realistic human face. For Example 2: A description of realistic and diverse short news articles that a GAN would generate. The output will either be a base64 encoded image URL or a text description.

Program 15: Develop RL-based Generative Models Using Benchmark Dataset

Title: Develop RL-based Generative Models Using Benchmark Dataset

Aim: To understand how Reinforcement Learning (RL) can be integrated with generative models to improve their generation process, often by using an RL agent to optimize a generative model's output based on a learned reward function, and to apply this concept to a benchmark dataset.

Procedure:

1. **Understand RL and Generative Models:** Review both concepts and identify areas of synergy (e.g., RL for optimizing discrete generation like text or molecules, or for improving image quality).
2. **Choose a Benchmark Dataset:** Select a dataset commonly used for generative tasks (e.g., MNIST for images, or a text corpus).
3. **Conceptual RL-Generative Architecture:** Imagine an architecture where a generative model proposes outputs, and an RL agent (or a learned reward function) provides feedback to guide the generator towards better outputs.
4. **Simulate Outcome:** Use a generative AI model (`gemini-2.0-flash` or `imagen-3.0-generate-002`) to simulate the *improved* output that such an RL-based generative model would produce on a benchmark dataset.

Source Code (Conceptual Python using `imagen-3.0-generate-002` to simulate improved generation):

```
# This code conceptually demonstrates developing RL-based generative models
# using a benchmark dataset. It simulates improved image generation,
# implying an RL component has optimized the generative process.

# Placeholder for API key
API_KEY = "" # If you want to use models other than gemini-3.0-generate-002,
provide an API key here. Otherwise, leave this as-is.

async def simulate_rl_generative_improvement(dataset_name,
improvement_description):
    """
    Simulates the improved generation from an RL-based generative model
    on a benchmark dataset.
    """
    prompt = f"Imagine an RL-based generative model has been trained on the
'{dataset_name}' benchmark dataset, resulting in '{improvement_description}'.
Generate an example of an output from this improved model."
    print(f"\nSimulating RL-based generative improvement for
'{dataset_name}': '{improvement_description}'")

    # For image datasets like MNIST, use imagen-3.0-generate-002
    if "mnist" in dataset_name.lower() or "image" in dataset_name.lower():
        generation_prompt = f"Generate a clear, high-quality image of a
digit, similar to an improved MNIST generation. The digit should be well-
formed and distinct."
        payload = { instances: { prompt: generation_prompt }, parameters: {
"sampleCount": 1} };
        apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/imagen-3.0-generate-
002:predict?key=${API_KEY}`;

        try:
            print("Generating image (simulating improved generation)...
Please wait.")
```

```
            response = await fetch(apiUrl, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(payload)
            });
            result = await response.json();

            if (result.predictions && result.predictions.length > 0 &&
result.predictions[0].bytesBase64Encoded) {
                image_url =
`data:image/png;base64,${result.predictions[0].bytesBase64Encoded}`;
                print("Simulated Improved Generative Output (Image generated
successfully):")
                print(f"Image URL (Base64 encoded): [Image Data Omitted for
brevity]")
                return image_url
            } else {
                print("Error: No image content found in response.")
                return "Error: No content found."
            }
        except Exception as e:
            print(f"An error occurred during image generation: {e}")
            return f"Error during image generation: {e}"
    else:
        # For other data types, use gemini-2.0-flash to describe the improved
output
        chatHistory = []
        chatHistory.push({ role: "user", parts: [{ text: prompt }] });
        payload = { contents: chatHistory };
        apiUrl =
`https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-
flash:generateContent?key=${API_KEY}`;

        try:
            response = await fetch(apiUrl, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(payload)
            });
            result = await response.json();
            if (result.candidates && result.candidates.length > 0 &&
                result.candidates[0].content &&
result.candidates[0].content.parts &&
                result.candidates[0].content.parts.length > 0) {
                description = result.candidates[0].content.parts[0].text;
                print(f"Simulated Improved Generative
Output:\n{description}")
                return description
            } else {
                print("Error: No content found in response.")
                return "Error: No content found."
            }
        except Exception as e:
            print(f"An error occurred: {e}")
            return f"Error during simulation: {e}"

async def main():
    # Example 1: Simulate improved MNIST digit generation
    await simulate_rl_generative_improvement(
        "MNIST",
        "significantly higher quality and diversity of generated digits, with
fewer artifacts"
    )

    # Example 2: Simulate improved text generation (e.g., on a story corpus)
    await simulate_rl_generative_improvement(
        "a story corpus",
        "more coherent, engaging, and grammatically correct narratives"
    )

# To run this in a browser environment (like Canvas), you'd typically call
main()
# within an async context or a button click handler.
# main()
```

Input:

1. Dataset: `"MNIST"`, Improvement: `"significantly higher quality and diversity of generated digits, with fewer artifacts"`
2. Dataset: `"a story corpus"`, Improvement: `"more coherent, engaging, and grammatically correct narratives"`

Expected Output: For Example 1: A generated image of a high-quality, clear digit (e.g., '3' or '8') resembling an improved MNIST sample. For Example 2: A description of a well-structured, engaging, and grammatically correct short story that an RL-based generative model would produce. The output will either be a base64 encoded image URL or a text description.