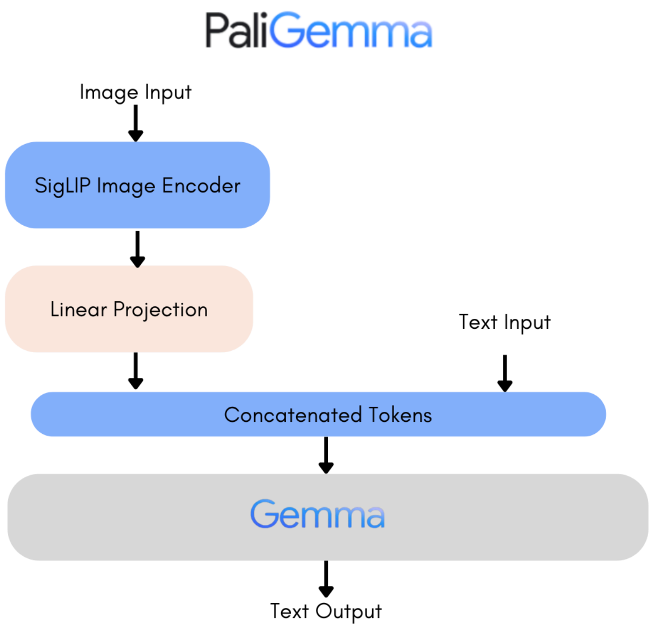
Paligemma :

Paligemma is a sophisticated, integrated AI system that combines vision and language models to provide comprehensive multimodal understanding and generation capabilities. The name "Paligemma" suggests a combination of "Pali," potentially hinting at a foundational or structural aspect, and "gemma," which can imply something valuable or precious, indicating the integration of crucial AI components.



PaliGemma is a vision-language model (VLM) developed by Google. It is a multimodal model that combines the capabilities of a vision model and a language model. The model is composed of a Siglip-400m vision encoder and a Gemma-2B decoder linked by a multimodal linear projection. PaliGemma is designed to process both images and text and generate text as output, supporting multiple languages

Key Features and Capabilities

1. Multimodal Comprehension: PaliGemma can simultaneously understand both images and text, making it suitable for tasks such as image captioning, visual question answering, and text reading from images.
2. Fine-Tuning: PaliGemma is designed to be fine-tuned on specific tasks, which allows it to adapt to different use cases and achieve better performance. This fine-tuning process involves adjusting the model's weights based on the specific task and dataset.
3. Pre-Training: PaliGemma is pre-trained on a variety of datasets, including WebLI, CC3M-35L, VQ^{2}A-CC3M-35L/VQG-CC3M-35L, OpenImages, and WIT. This pre-training helps the model learn general representations of images and text that can be leveraged for downstream tasks.

4. Resolutions and Precisions: PaliGemma models come in three resolutions (224x224, 448x448, and 896x896) and three precisions (bfloat16, float16, and float32). The higher resolutions are more memory-intensive but can be beneficial for fine-grained tasks like optical character recognition (OCR).
5. Integration with Transformers: PaliGemma models are integrated with the transformers library, making it easy to use and fine-tune the models for specific tasks

Use Cases and Benchmarks

PaliGemma is suitable for a variety of tasks, including:

1. Image Captioning: PaliGemma can generate captions for images based on the input text and image.
2. Visual Question Answering: The model can answer questions about images, providing detailed and contextual responses.
3. Text Reading from Images: PaliGemma can read text embedded within images, such as captions or signs.
4. Object Detection and Segmentation: The model can be fine-tuned for tasks like object detection and segmentation, which involve identifying and localizing objects within images.

Limitations and Future Directions

1. Niche Datasets: PaliGemma may struggle with niche datasets or environments that were not present during pretraining, which is expected given the limited scope of its pretraining.
2. Fine-Tuning: While PaliGemma is designed to be fine-tuned, the model's performance can be improved significantly by fine-tuning it on specific tasks and datasets.
3. Comparison to Other Models: PaliGemma can be compared to other VLMs and LMMs, such as ChatGPT-4o, which have larger architectures but may not be as efficient or fine-tunable

Architecture of Paligemma

The architecture of Paligemma can be divided into several key components:

1. **Input Processing Module**:
   - **Vision Processing**: This module processes visual inputs using advanced vision models such as SigLIP.
   - **Language Processing**: This module handles textual inputs using the Gemma language model.
2. **Multimodal Fusion Layer**:

    - This layer integrates outputs from both the vision and language processing modules to create a unified representation. Techniques like cross-modal attention mechanisms are often used here.
3. **Core Understanding Engine**:
    - **Contextual Understanding**: Integrates multimodal information to understand the context and nuances of the input data.
    - **Knowledge Integration**: Utilizes external knowledge bases to enhance understanding and provide more accurate responses.
4. **Output Generation Module**:
    - **Response Generation**: Uses the integrated representation to generate appropriate responses or actions.
    - **Adaptive Learning**: Continuously learns from interactions to improve future responses.
5. **Feedback Loop**:
    - **Performance Monitoring**: Tracks the performance of the system and identifies areas for improvement.
    - **Iterative Learning**: Updates the model based on feedback to refine its capabilities.

PaliGemma is a vision-language model (VLM) developed by Google that combines a vision encoder and a language decoder. Its architecture consists of:

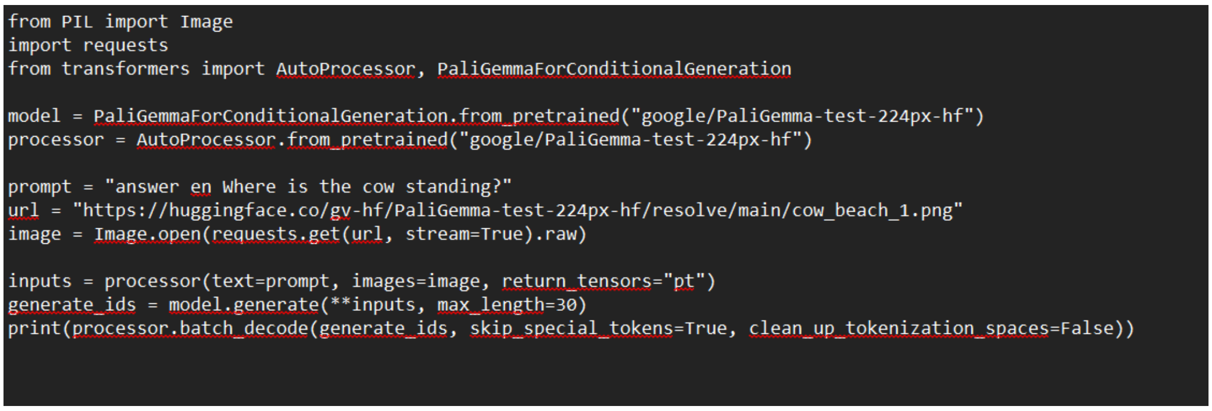
- SigLIP-400m as the vision encoder: SigLIP is a robust contrastively trained visual encoder similar to OpenAI's CLIP, but using a simpler sigmoid loss function.
- Gemma-2B as the text decoder: Gemma is a relatively compact decoder-only language model from Google. It tokenizes the input text and processes all tokens using its 256,000 token vocabulary.
- Gemma's transformer-based decoder: The decoder is largely similar to the original transformer decoder by Vaswani et al. (2017), with modifications like multi-head attention, rotary positional embeddings, GeGLU activation, and RMSNorm.
- Additional tokens: PaliGemma extends Gemma's token vocabulary with 1024 location tokens (<loc0000> to <loc1023>) representing normalized image coordinates, and 128 segmentation tokens (<seg000> to <seg127>) from a vector quantized visual auto-encoder.

The vision encoder and language decoder are linked using a multimodal linear projection. PaliGemma is designed to take both image and text as input and generate text as output, supporting multiple languages.

The model has a total of 3 billion parameters and is pre-trained on a mixture of datasets like WebLI, CC3M-35L, VQ^{2}A-CC3M-35L/VQG-CC3M-35L, OpenImages, and WIT. It is designed to be fine-tuned on specific vision-language tasks for better performance

How Paligemma Works

1. **Input Reception**: The system receives visual and textual inputs.
2. **Processing**: The inputs are processed through their respective modules—visual data through SigLIP and textual data through Gemma.
3. **Integration**: The multimodal fusion layer combines the processed data into a coherent representation.
4. **Understanding**: The core understanding engine interprets the integrated data, using context and external knowledge.
5. **Response Generation**: An appropriate response is generated based on the interpretation.
6. **Learning and Adaptation**: The system learns from interactions and feedback to improve its future performance.



```
from PIL import Image
import requests
from transformers import AutoProcessor, PaliGemmaForConditionalGeneration

model = PaliGemmaForConditionalGeneration.from_pretrained("google/PaliGemma-test-224px-hf")
processor = AutoProcessor.from_pretrained("google/PaliGemma-test-224px-hf")

prompt = "answer en Where is the cow standing?"
url = "https://huggingface.co/gv-hf/PaliGemma-test-224px-hf/resolve/main/cow_beach_1.png"
image = Image.open(requests.get(url, stream=True).raw)

inputs = processor(text=prompt, images=image, return_tensors="pt")
generate_ids = model.generate(**inputs, max_length=30)
print(processor.batch_decode(generate_ids, skip_special_tokens=True, clean_up_tokenization_spaces=False))
```

SigLIP Vision Model

Definition

SigLIP (Signal Language-Image Pretraining) is a vision model designed to understand and interpret visual data. It employs a pretraining technique that integrates both visual and textual information to enhance its understanding capabilities.

SigLIP is based on the Vision Transformer (ViT) architecture, which uses self-attention mechanisms to process input images. The model consists of a series of transformer blocks, each of which includes a multi-head self-attention mechanism and a feed-forward network (FFN). The output of each block is a set of feature maps that capture different aspects of the input image

Architecture of SigLIP

1. **Image Encoder**:

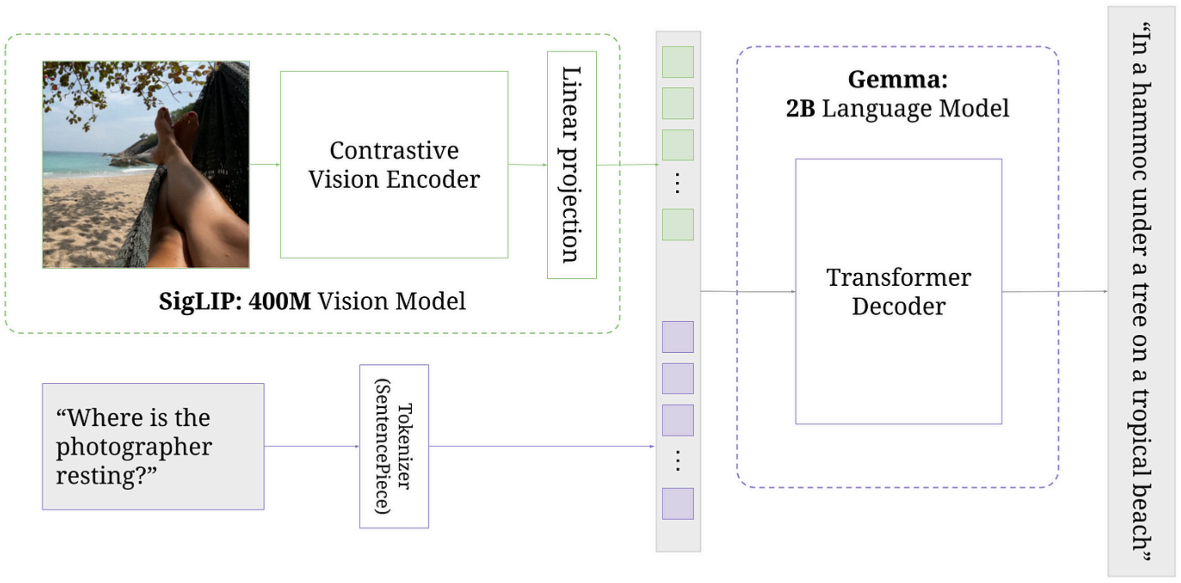
    - Utilizes Convolutional Neural Networks (CNNs) or Vision Transformers (ViTs) to extract features from images.
    - Multi-layered structure with attention mechanisms to focus on important aspects of the visual data.
2. **Text Encoder**:
    - Incorporates a language model (like BERT or GPT) to process textual descriptions associated with images.
    - Embedding layers to convert text into vector representations.
3. **Cross-Modal Attention Mechanism**:
    - Connects the image and text encoders, allowing the model to learn correspondences between visual features and textual descriptions.
    - Uses attention layers to highlight relevant parts of the image based on the text and vice versa.
4. **Fusion Layer**:
    - Combines the outputs of the image and text encoders into a unified representation.
    - Dense layers and normalization techniques to ensure cohesive integration.
5. **Output Layer**:
    - Produces predictions or classifications based on the fused representation.
    - Can be fine-tuned for specific tasks such as image captioning, visual question answering, or object recognition.

Functionality

- **Pretraining**: The model is pretrained on large datasets containing paired image-text data to learn the relationships between visual and textual information.
- **Fine-Tuning**: After pretraining, the model can be fine-tuned on specific datasets to adapt to various vision-related tasks.

Training

SigLIP is trained using a contrastive loss function, which aims to maximize the similarity between positive pairs of images and minimize the similarity between negative pairs. This approach helps the model learn robust and generalizable representations of images. The model is trained on a large dataset of images and their corresponding text descriptions, which are used to generate positive and negative pairs.



Contrastive Vision Encoder
Linear projection
SigLIP: 400M Vision Model
Gemma: 2B Language Model
Transformer Decoder
"In a hammoc under a tree on a tropical beach"
"Where is the photographer resting?"
Tokenizer (SentencePiece)

Key Features

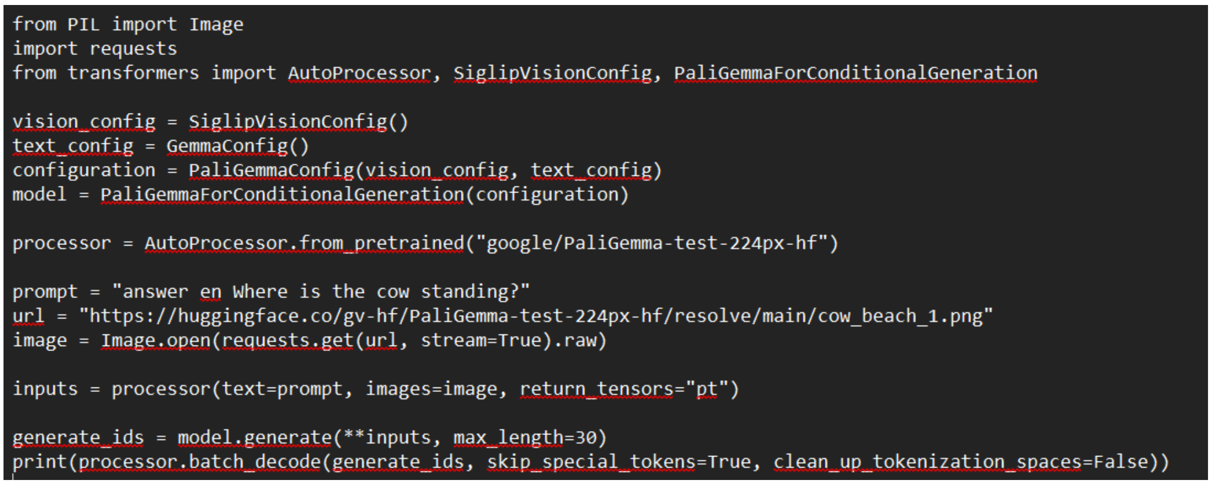
1. *Robustness:* SigLIP is designed to be robust to various types of image corruptions and transformations, such as noise, blur, and rotation. This makes it suitable for real-world applications where images may be degraded or distorted.
2. *Efficiency:* SigLIP is optimized for efficiency and can be used on a wide range of devices, from mobile phones to high-performance servers. This makes it a practical choice for many applications.
3. *Multimodal Capabilities:* SigLIP can be used in conjunction with other models, such as language models, to perform multimodal tasks like image captioning and visual question answering.

Comparison to Other Models

SigLIP is comparable to other robust visual encoders like **CLIP**, which is also developed by OpenAI. While both models are designed to be robust and efficient, SigLIP is simpler and more lightweight, making it easier to integrate into various applications.

Implementation

SigLIP can be implemented using the transformers library in Python. The following code snippet demonstrates how to use SigLIP for image captioning:



```
from PIL import Image
import requests
from transformers import AutoProcessor, SiglipVisionConfig, PaliGemmaForConditionalGeneration

vision_config = SiglipVisionConfig()
text_config = GemmaConfig()
configuration = PaliGemmaConfig(vision_config, text_config)
model = PaliGemmaForConditionalGeneration(configuration)

processor = AutoProcessor.from_pretrained("google/PaliGemma-test-224px-hf")

prompt = "answer en Where is the cow standing?"
url = "https://huggingface.co/gv-hf/PaliGemma-test-224px-hf/resolve/main/cow_beach_1.png"
image = Image.open(requests.get(url, stream=True).raw)

inputs = processor(text=prompt, images=image, return_tensors="pt")

generate_ids = model.generate(**inputs, max_length=30)
print(processor.batch_decode(generate_ids, skip_special_tokens=True, clean_up_tokenization_spaces=False))
```

Gemma Language Model

Definition

The Gemma language model is an advanced neural network model designed to understand and generate human language. It leverages extensive pretraining on vast amounts of text data to develop a deep understanding of language nuances.

Gemma is a family of lightweight, state-of-the-art open models developed by Google. It is designed to be a robust and efficient model that can be used for various applications such as text generation and multimodal tasks. Here are the key aspects of Gemma:

Architecture and Training

Gemma is based on the transformer architecture and is trained using a combination of masked language modeling and next sentence prediction tasks. The model is trained on a large dataset of text and is designed to be robust and efficient.

Model Sizes and Capabilities

Gemma models are available in two sizes: 2B and 7B. The 2B model is designed for lower resource requirements and can run on mobile devices and laptops, while the 7B model is more powerful and can run on desktop computers and small servers.

Tuning and Customization

Gemma models can be tuned and customized for specific tasks using techniques such as LoRA (Low-Rank Adaptation) and model parallelism. This allows developers to adapt the model to their specific needs and improve its performance on targeted tasks.

Responsible AI Development

Gemma is designed with responsible AI development in mind. The model is trained on curated data and is tuned for safety using techniques such as automated filtering of personal information and extensive fine-tuning with human feedback. The model is also evaluated using robust methods such as manual red-teaming and automated adversarial testing to ensure it does not exhibit dangerous behaviors.

Deployment and Integration

Gemma models can be deployed on various platforms, including Google Cloud, and can be integrated with popular frameworks such as JAX, PyTorch, and TensorFlow. The model can also be fine-tuned on specific data sets and tasks using tools such as LoRA and model parallelism.

Performance and Benchmarks

Gemma models have achieved state-of-the-art performance for their size compared to other open models. The model has been tested on various benchmarks and has shown exceptional performance in tasks such as text generation and multimodal tasks.

Availability and Community

Gemma models are available for download from Kaggle and can be used for various applications. The model has a growing community of developers and researchers who are working on fine-tuning and customizing the model for specific tasks.

Key Features

1. Lightweight and Efficient: Gemma models are designed to be lightweight and efficient, making them suitable for deployment on a wide range of devices and platforms.
2. State-of-the-Art Performance: Gemma models have achieved state-of-the-art performance for their size compared to other open models.
3. Responsible AI Development: Gemma is designed with responsible AI development in mind, incorporating techniques such as automated filtering of personal information and extensive fine-tuning with human feedback.
4. Customization and Tuning: Gemma models can be tuned and customized for specific tasks using techniques such as LoRA and model parallelism.
5. Integration with Popular Frameworks: Gemma models can be integrated with popular frameworks such as JAX, PyTorch, and TensorFlow, making it easy to use and deploy

Architecture of Gemma

1. **Embedding Layer**:
    - Converts input text into dense vector representations.
    - Utilizes word embeddings or contextual embeddings like those from BERT or GPT.
2. **Transformer Layers**:
    - Multiple transformer blocks, each containing self-attention mechanisms and feed-forward neural networks.
    - Layer normalization and residual connections to maintain stable training.
3. **Contextual Understanding Module**:
    - Encodes the context of the input text to generate coherent and contextually appropriate responses.
    - Uses attention mechanisms to focus on relevant parts of the input text.
4. **Output Layer**:
    - Generates the final output text based on the processed and contextually understood input.
    - Can produce various forms of output such as summaries, translations, or conversational responses.

Functionality

- **Pretraining**: Trained on diverse text corpora to learn language patterns, grammar, and context.
- **Fine-Tuning**: Adapted to specific tasks like question answering, text generation, or sentiment analysis through fine-tuning on relevant datasets.

Integration in Paligemma

In the Paligemma system, SigLIP and Gemma work in tandem to process and understand multimodal inputs. SigLIP handles visual data, converting it into a format that can be integrated with the textual data processed by Gemma. The fusion layer then combines these representations, allowing the core understanding engine to interpret the integrated data and generate appropriate responses. This integration enables Paligemma to excel in tasks that require a deep understanding of both visual and textual information.

Model Data

Data for Paligemma

Paligemma, being a multimodal system, requires diverse datasets encompassing both visual and textual data:

1. **Image-Text Pairs**: Datasets like COCO (Common Objects in Context) and Visual Genome provide images with corresponding textual descriptions or annotations.
2. **Textual Data**: Large corpora of text, such as Wikipedia articles, books, and web pages, are used to pretrain the language model (Gemma).
3. **Annotated Multimodal Data**: Datasets specifically designed for tasks like visual question answering (VQA), image captioning, and scene understanding, which combine both image and text annotations.

Data for SigLIP

SigLIP's training data includes:

1. **Image Datasets**: High-quality and diverse image datasets like ImageNet, COCO, and Open Images.
2. **Paired Image-Text Data**: Data where each image is paired with descriptive text, aiding the model in learning the relationships between visual content and language.

Data for Gemma

Gemma's data requirements are primarily textual:

1. **Large-Scale Text Corpora**: Datasets such as the Common Crawl, Wikipedia, and BookCorpus provide extensive text data for pretraining.
2. **Specialized Text Datasets**: Fine-tuning datasets tailored to specific tasks like sentiment analysis, question answering (SQuAD), and natural language inference.

Model Building

Building Paligemma

1. **Data Collection and Preprocessing**:
   - Collect and clean large volumes of multimodal data.
   - Preprocess images (resizing, normalization) and text (tokenization, normalization).
2. **Pretraining**:
   - Train SigLIP on image-text pairs to learn cross-modal representations.
   - Train Gemma on large-scale text corpora to develop a deep understanding of language.
3. **Multimodal Fusion**:
   - Develop and train the fusion layer to integrate visual and textual features.
   - Utilize techniques like cross-attention to effectively combine modalities.
4. **Fine-Tuning**:
   - Fine-tune the integrated model on task-specific multimodal datasets.
   - Use transfer learning to adapt pre-trained models to new tasks with limited data.

Building SigLIP

1. **Image Encoder Training**:
    - Use convolutional neural networks (CNNs) or Vision Transformers (ViTs) to extract features from images.
    - Train on large image datasets to develop robust visual representations.
2. **Text Encoder Training**:
    - Employ transformers to process textual descriptions associated with images.
    - Train on paired image-text data to learn the correlation between visual and textual information.
3. **Cross-Modal Pretraining**:
    - Implement cross-modal attention mechanisms to align image features with textual features.
    - Pretrain on large-scale datasets to learn rich, shared representations.

Building Gemma

1. **Embedding Layer Setup**:
    - Initialize word embeddings using pre-trained vectors or train embeddings from scratch on a large text corpus.
2. **Transformer Training**:
    - Use transformer architecture with multiple layers of self-attention and feed-forward networks.
    - Pretrain on extensive text data to capture language patterns and contextual relationships.
3. **Contextual Understanding**:
    - Integrate advanced attention mechanisms to focus on relevant text segments.
    - Train on datasets like books, articles, and dialogues to understand various contexts and nuances.

Model Architecture

Architecture of Paligemma

1. **Input Processing Module**:
    - **SigLIP Vision Processing**: Image encoder + text encoder + cross-modal attention.
    - **Gemma Language Processing**: Transformer-based language model.
2. **Multimodal Fusion Layer**:
    - Cross-attention mechanisms to combine image and text features.
    - Dense layers and normalization to create a unified representation.
3. **Core Understanding Engine**:
    - Contextual understanding through integrated representations.

    - External knowledge integration via APIs or databases to enhance comprehension.
4. **Output Generation Module**:
    - Decoders for generating textual responses or actions.
    - Task-specific layers for applications like VQA or image captioning.
5. **Feedback Loop**:
    - Performance monitoring to identify strengths and weaknesses.
    - Iterative learning to adapt and improve based on feedback.

Architecture of SigLIP

1. **Image Encoder**:
    - Convolutional layers or Vision Transformer layers to process images.
    - Attention mechanisms to focus on important visual features.
2. **Text Encoder**:
    - Transformer layers to process associated textual descriptions.
    - Embedding layers to convert text into meaningful vectors.
3. **Cross-Modal Attention Mechanism**:
    - Attention layers connecting image and text encoders.
    - Mechanisms to highlight relevant image regions based on text and vice versa.
4. **Fusion Layer**:
    - Combines image and text embeddings into a cohesive representation.
    - Dense layers for integration and normalization.
5. **Output Layer**:
    - Task-specific layers for classification, captioning, or other vision tasks.
    - Fine-tuning mechanisms to adapt to different applications.

Architecture of Gemma

1. **Embedding Layer**:
    - Converts text into dense vector representations.
    - Utilizes pre-trained word embeddings or contextual embeddings.
2. **Transformer Layers**:
    - Multiple layers of self-attention and feed-forward networks.
    - Layer normalization and residual connections for stable training.
3. **Contextual Understanding Module**:
    - Attention mechanisms to focus on relevant parts of the text.
    - Mechanisms to maintain context over long text sequences.
4. **Output Layer**:
    - Generates final text outputs, such as responses, summaries, or translations.
    - Task-specific adaptations for different language applications.

By integrating these sophisticated components, Paligemma achieves a powerful synergy between vision and language, enabling it to perform complex tasks that require deep understanding and generation capabilities across both modalities.

Fine-tuning Paligemma involves adjusting the pre-trained models (SigLIP for vision and Gemma for language) on a specific task using a tailored dataset. Here's a detailed guide on how to fine-tune Paligemma, along with code examples.

Prerequisites

1. **Data Preparation**: You need a dataset relevant to your task, which contains both visual and textual information. For example, a Visual Question Answering (VQA) dataset.
2. **Environment Setup**: Ensure you have a suitable machine learning environment with necessary libraries installed (e.g., PyTorch, Transformers, Vision libraries).

Step-by-Step Fine-Tuning Process

PaliGemma is a vision-language model (VLM) developed by Google that can be fine-tuned for various tasks. Fine-tuning involves adjusting the model's weights based on specific task requirements and datasets to improve its performance. Here's a step-by-step guide on how to fine-tune PaliGemma:

Step 1: Download the Model and Dependencies

1. Download PaliGemma Model Checkpoint: Download the pre-trained PaliGemma model checkpoint and tokenizer from Kaggle or other sources.
2. Install Dependencies: Install the required dependencies, including JAX, TensorFlow, NumPy, and other libraries.

Step 2: Prepare the Model

1. Load the Model: Load the pre-trained PaliGemma model onto GPU devices.
2. Prepare Inputs: Prepare the model's inputs for training and inference by processing the data into the required format.

Step 3: Fine-tune the Model

1. Freeze Parameters: Freeze the majority of the model's parameters, except for the attention layers, to prevent overfitting.
2. Train the Model: Train the fine-tuned model using the specific task's dataset and hyperparameters. This step can be done using JAX and other libraries.

Step 4: Test and Save the Model

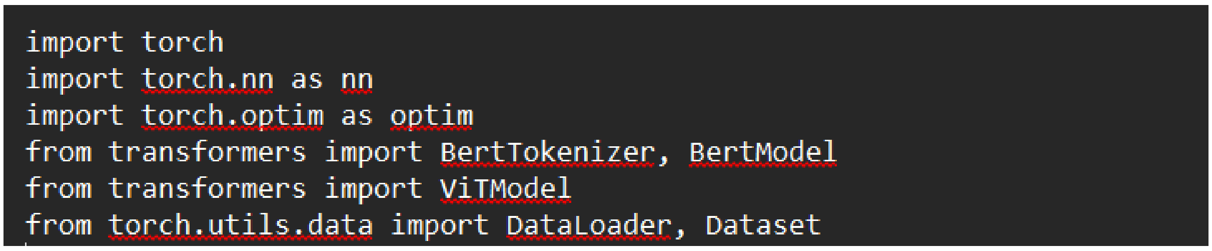
1. Test the Model: Test the fine-tuned model on a validation dataset to evaluate its performance.

2. Save the Model: Save the fine-tuned model for later use by saving its weights and other parameters.

Additional Tips

1. Use a Small Version: For fine-tuning in Google Colab, use the smallest version of PaliGemma (paligemma-3b-pt-224) to limit GPU memory consumption.
2. Use Roboflow Universe: Use Roboflow Universe for accessing and managing datasets for fine-tuning PaliGemma.
3. Fine-tune for Specific Tasks: Fine-tune PaliGemma for specific tasks such as object detection, segmentation, or text reading from images

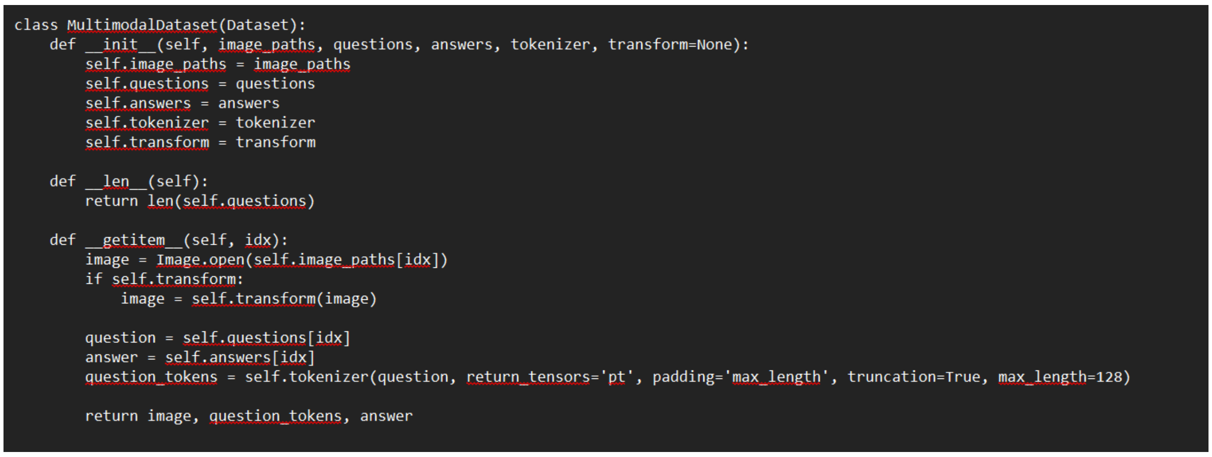
1. Import Necessary Libraries



```
import torch
import torch.nn as nn
import torch.optim as optim
from transformers import BertTokenizer, BertModel
from transformers import ViTModel
from torch.utils.data import DataLoader, Dataset
```

2. Prepare the Dataset

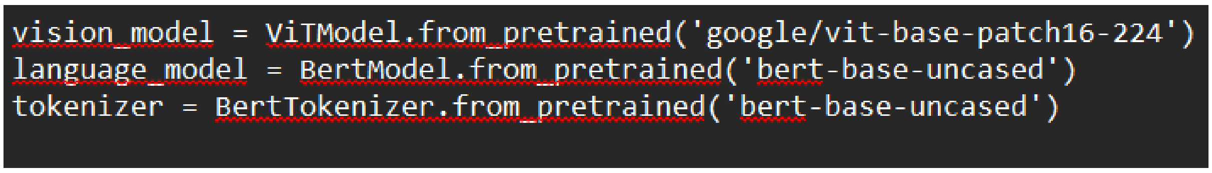
Define a custom dataset class to handle your data.



```
class MultimodalDataset(Dataset):
    def __init__(self, image_paths, questions, answers, tokenizer, transform=None):
        self.image_paths = image_paths
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.transform = transform

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        image = Image.open(self.image_paths[idx])
        if self.transform:
            image = self.transform(image)

        question = self.questions[idx]
        answer = self.answers[idx]
        question_tokens = self.tokenizer(question, return_tensors='pt', padding='max_length', truncation=True, max_length=128)

        return image, question_tokens, answer
```

3. Initialize Pre-trained Models

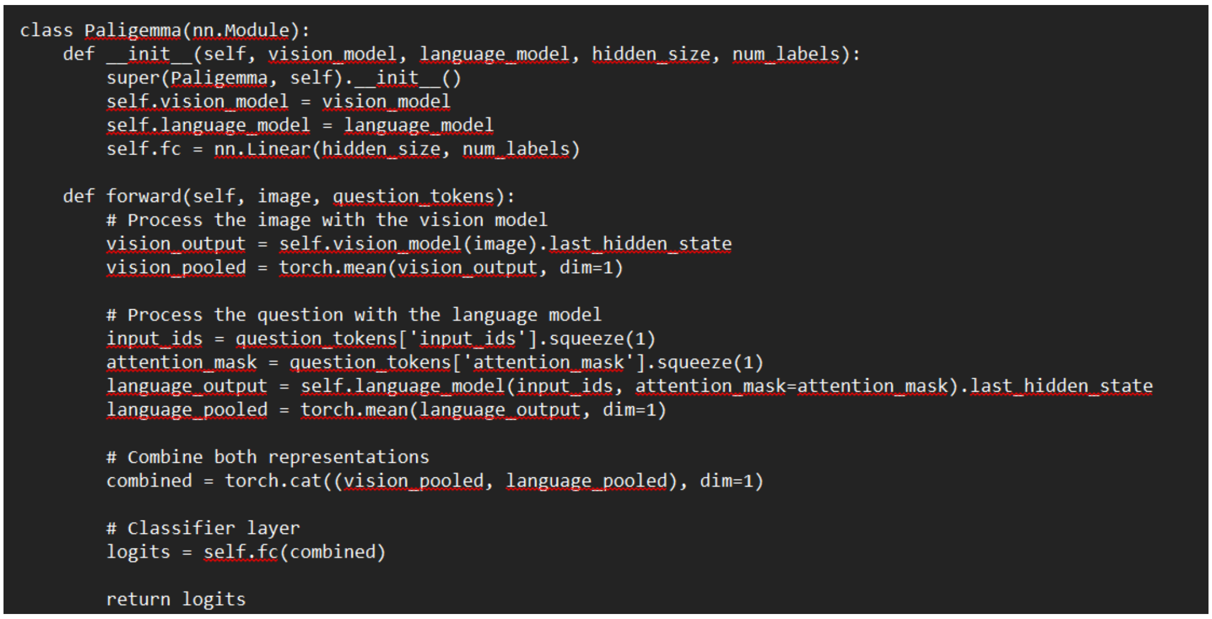
Load pre-trained models for vision and language.



```
vision_model = ViTModel.from_pretrained('google/vit-base-patch16-224')
language_model = BertModel.from_pretrained('bert-base-uncased')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

4. Define the Multimodal Model

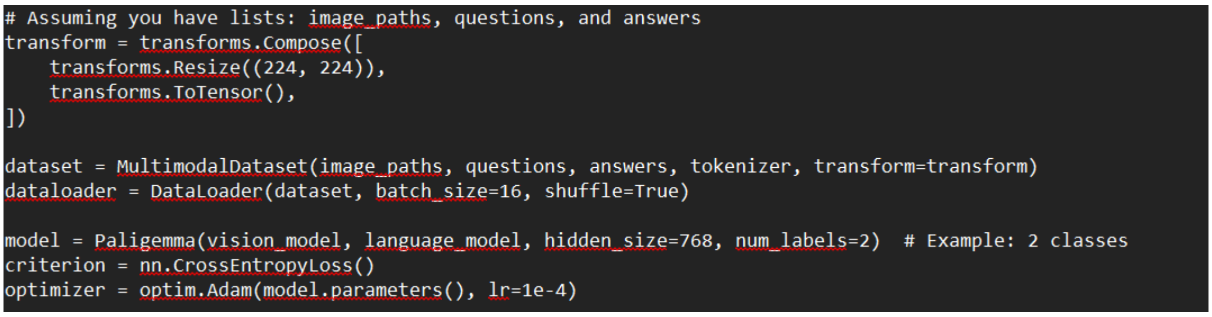
Combine vision and language models into a single architecture.



```
class Paligemma(nn.Module):
    def __init__(self, vision_model, language_model, hidden_size, num_labels):
        super(Paligemma, self).__init__()
        self.vision_model = vision_model
        self.language_model = language_model
        self.fc = nn.Linear(hidden_size, num_labels)

    def forward(self, image, question_tokens):
        # Process the image with the vision model
        vision_output = self.vision_model(image).last_hidden_state
        vision_pooled = torch.mean(vision_output, dim=1)

        # Process the question with the language model
        input_ids = question_tokens['input_ids'].squeeze(1)
        attention_mask = question_tokens['attention_mask'].squeeze(1)
        language_output = self.language_model(input_ids, attention_mask=attention_mask).last_hidden_state
        language_pooled = torch.mean(language_output, dim=1)

        # Combine both representations
        combined = torch.cat((vision_pooled, language_pooled), dim=1)

        # Classifier layer
        logits = self.fc(combined)

        return logits
```

5. Prepare for Training

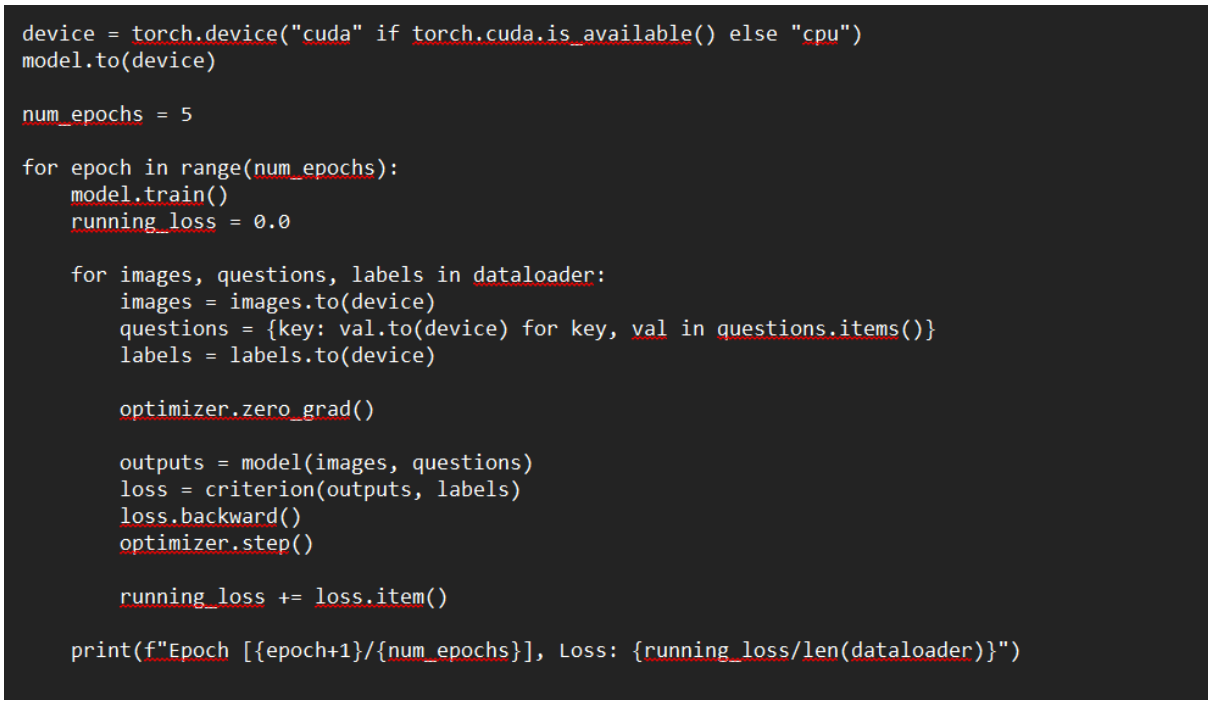
Set up the data loaders, loss function, and optimizer.



```
# Assuming you have lists: image_paths, questions, and answers
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

dataset = MultimodalDataset(image_paths, questions, answers, tokenizer, transform=transform)
dataloader = DataLoader(dataset, batch_size=16, shuffle=True)

model = Paligemma(vision_model, language_model, hidden_size=768, num_labels=2)  # Example: 2 classes
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

6. Fine-Tuning the Model

Define the training loop to fine-tune the model.



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

num_epochs = 5

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    for images, questions, labels in dataloader:
        images = images.to(device)
        questions = {key: val.to(device) for key, val in questions.items()}
        labels = labels.to(device)

        optimizer.zero_grad()

        outputs = model(images, questions)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(dataloader)}")
```