Chapter 9 - Multimodal Large Language Models

When you think about large language models (LLMs), multimodality might not be the first thing that comes to mind.

> **Multimodality in LLMs**
>
> Multimodality in large language models (LLMs) refers to the ability of these models to process and understand multiple types of data, such as text, images, and audio.

As the name suggests, LLMs are ***language*** models.

But we can quickly see that models can be much more useful if they're able to handle types of data other than text.

A model that is able to handle text and images (each of which is called a ***modality***) is said to be ***multimodal***.

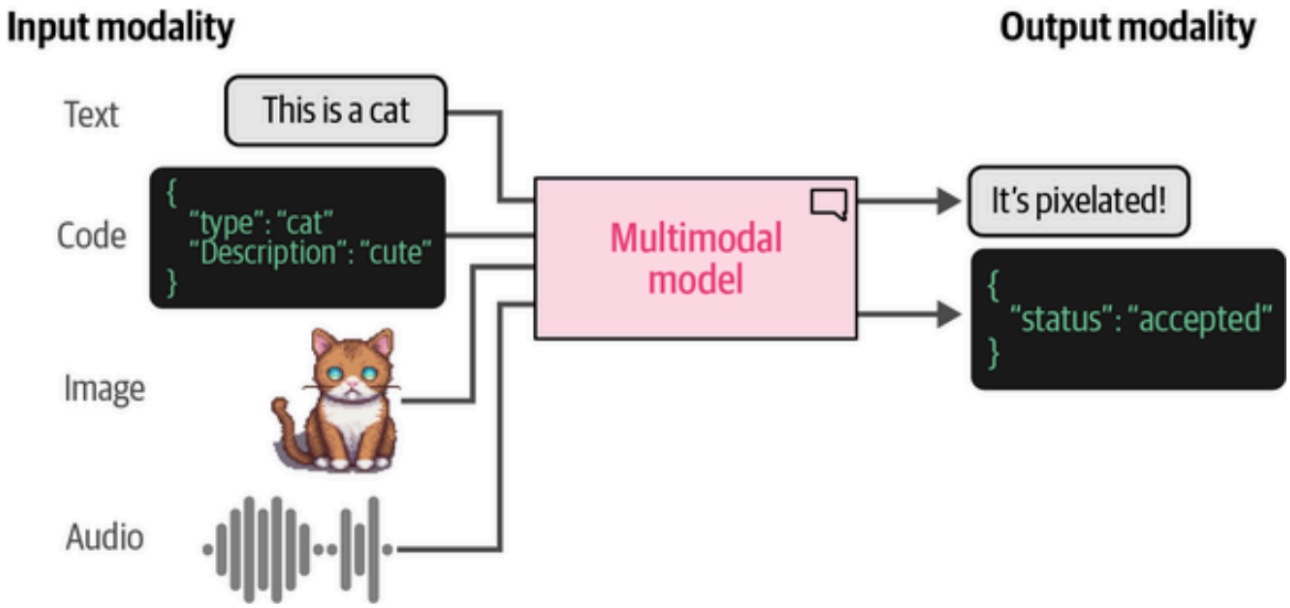


Figure 9-1. Models that are able to deal with different types (or modalities) of data, such as images, audio, video, or sensors, are said to be multimodal. It's possible for a model to accept a modality as input yet not be able to generate in that modality.

In the previous chapters, we have seen what LLMs can do, from generalization and reasoning to arithmetic and language processing.

As models grow larger and smarter, so do their skill sets.

If models can handle multimodal input (audio, visual data, etc.), they can do more things.

Just like in real life, communication includes body language and facial expressions, not just words.

This applies to LLMs. As you can imagine, if an LLM can process images, audio, and text, it would be better at giving an output for a query.

Imagine you not only read, but witness a story, you would be able to make a better response to a question about it.

In this chapter, we'll start by looking at how images are converted to numerical representations.

As you will soon see, it uses an adaptation of the original Transformer technique.

Transformers for Vision

From previous chapters, we have seen and demonstrated what transformers are capable of, including classification, clustering, search, and generative modeling.

Because of the increasing usefulness of transformers, scientists have looked into its potential in the field of computer vision.

They came up with a method called the **Vision Transformer (ViT)**.

> **ViT (Vision Transformer)**
>
> A Vision Transformer (ViT) is a type of neural network architecture that applies the transformer model, originally designed for natural language processing, to computer vision tasks. It processes images by dividing them into patches, treating these patches as sequences, and using self-attention to capture relationships across the entire image.

Like the original Transformer, **ViT** is used to transform unstructured data, an image, into representations (embeddings).

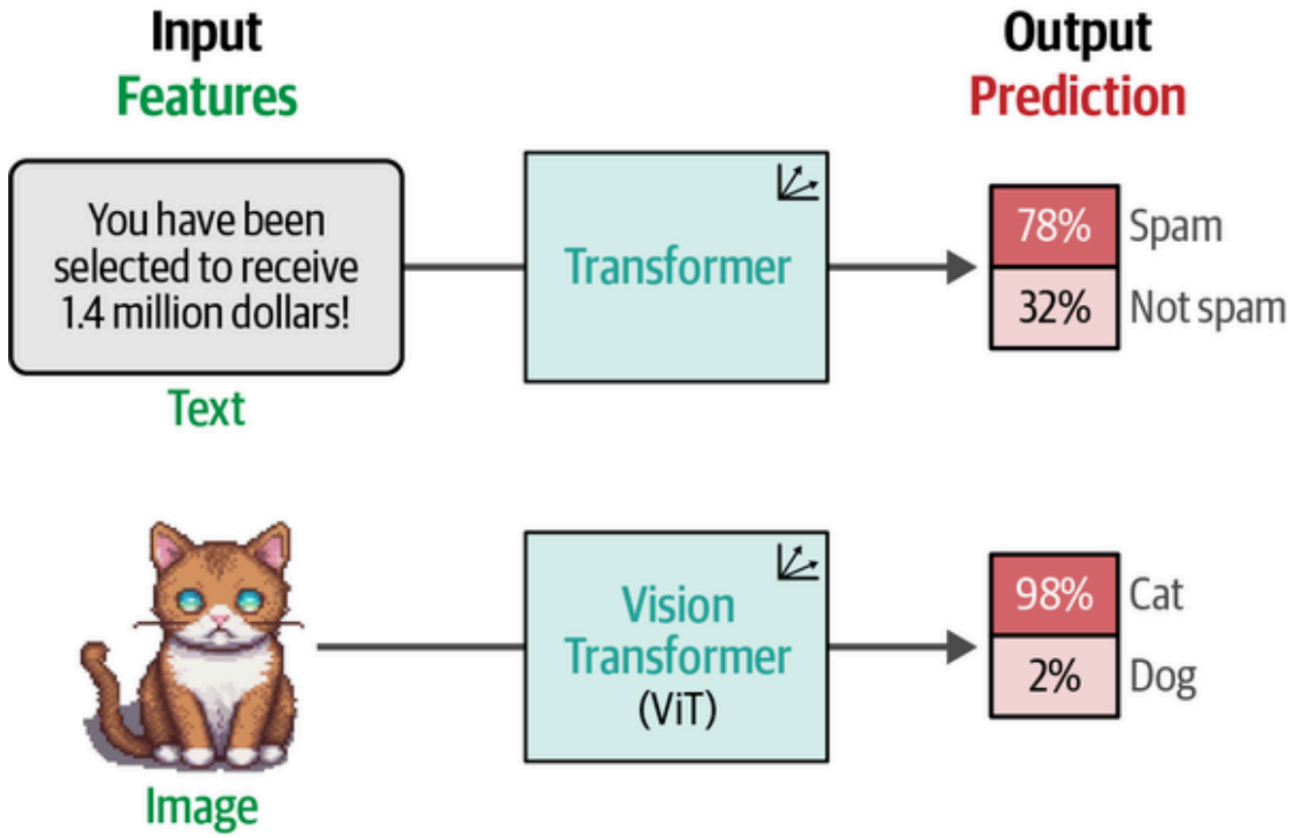


Figure 9-2. Both the original Transformer as well as the Vision Transformer take unstructured data, convert it to numerical representations, and finally use that for tasks like classification.

ViT relies on the **encoder** component of the Transformer architecture. As we saw in Chapter 1, the encoder converts textual input into embeddings before being passed to the **decoder**.

However, before this can be done, the textual input needs to be tokenized first.

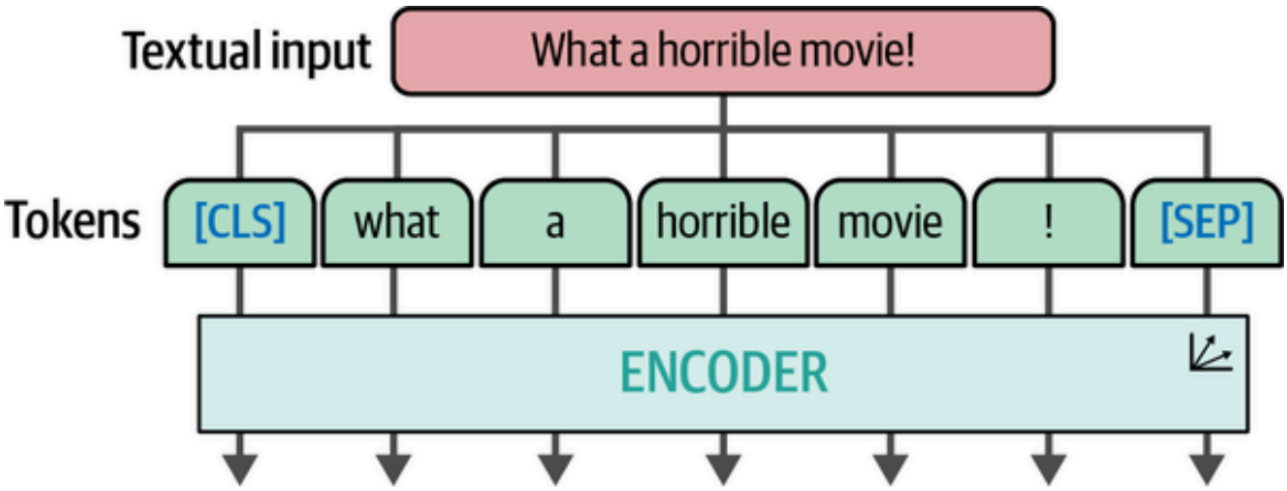


Figure 9-3. Text is passed to one or multiple encoders by first tokenizing it using a tokenizer.

Unlike text, an image does not consist of words, so the tokenization process is different for images.

The authors of ViT came up with a method for tokenizing images into "words."

This lets them use the original encoder structure.

Consider an image of a cat. As we know, images are represented by pixels, let's say 512 × 512 pixels.

Unlike tokens which use a vocabulary to find their individual separate meaning, an individual pixel doesn't tell you much.

Only when combined do pixels give an idea what an image or a part of an image is.

The creators of **ViT** knew this and took this into account. So instead of processing separate pixels, ViT uses ***patches*** of an image. This is illustrated in the diagram below.

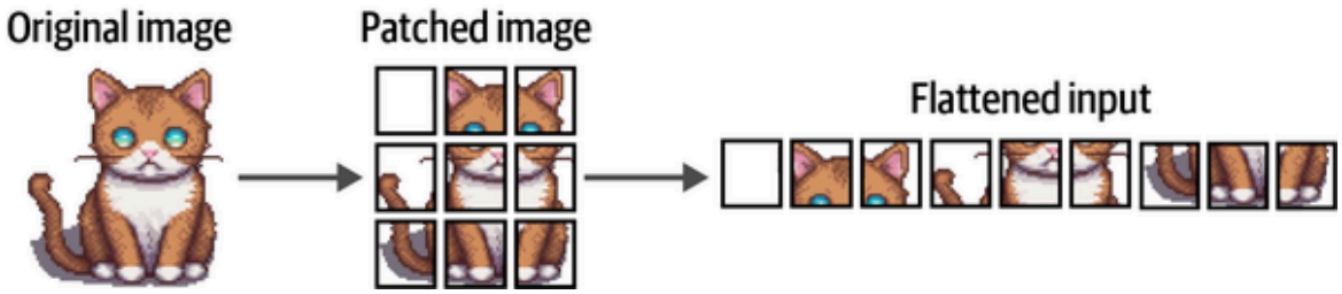


Figure 9-4. The "tokenization" process for image input. It converts an image into patches of subimages.

In the original ***ViT*** paper, a patch is of size **16 x 16 x 3** (RGB channels).

After obtaining these patches, the RGB values for each patch are normalized to a 0-1 range. Then, each patch (16 x 16 x 3) is flattened into a single vector of 768 values.

Each flattened patch vector is then passed through a **linear projection layer** (a simple neural network) to generate an embedding for that specific patch.

The vector for each patch now works like tokens, but without a vocabulary. Using a vocabulary ID for images doesn't make much sense, images and patches of images have more complex characteristics and RGB similarity does not really ensure image similarity.

That is why they need a linear embedding model. These embeddings can then be used as input to a Transformer model.

What is a Linear Embedding Layer?

It's a **single fully-connected layer** (also called a dense layer or linear layer).

Mathematical operation:

```
Output = (Input × Weight_Matrix) + Bias
```

For ViT:

- Input: Flattened patch vector (768 values)
- Weight Matrix: 768 × D (where D = desired embedding dimension, often 768)
- Output: Embedding of dimension D

How is it Different from CNNs?

Linear Projection	CNN
Single matrix multiplication	Multiple convolutional layers
No spatial awareness	Learns spatial patterns (edges, shapes)
Simple & fast	Complex hierarchy of features
Treats input as flat vector	Preserves 2D structure

Why not use CNN? ViT is "let the Transformer handle everything." The linear layer just transforms the raw patch data into the right shape. The Transformer's attention mechanism does the heavy lifting for understanding relationships.

How is it Trained?

It's not pretrained separately. It's trained **end-to-end** with the entire ViT model.

Training process:

1. Image->patches->flatten->linear projection->Transformer encoder
2. Final output used for task (classification, etc.)
3. Loss calculated
4. Gradients backpropagate through entire network
5. Linear projection weights updated along with everything else

Key point: The linear projection learns to transform raw pixel values into representations that the Transformer can work with effectively. It learns *what features to extract* from patches based on the final task.

The full process can be seen below.

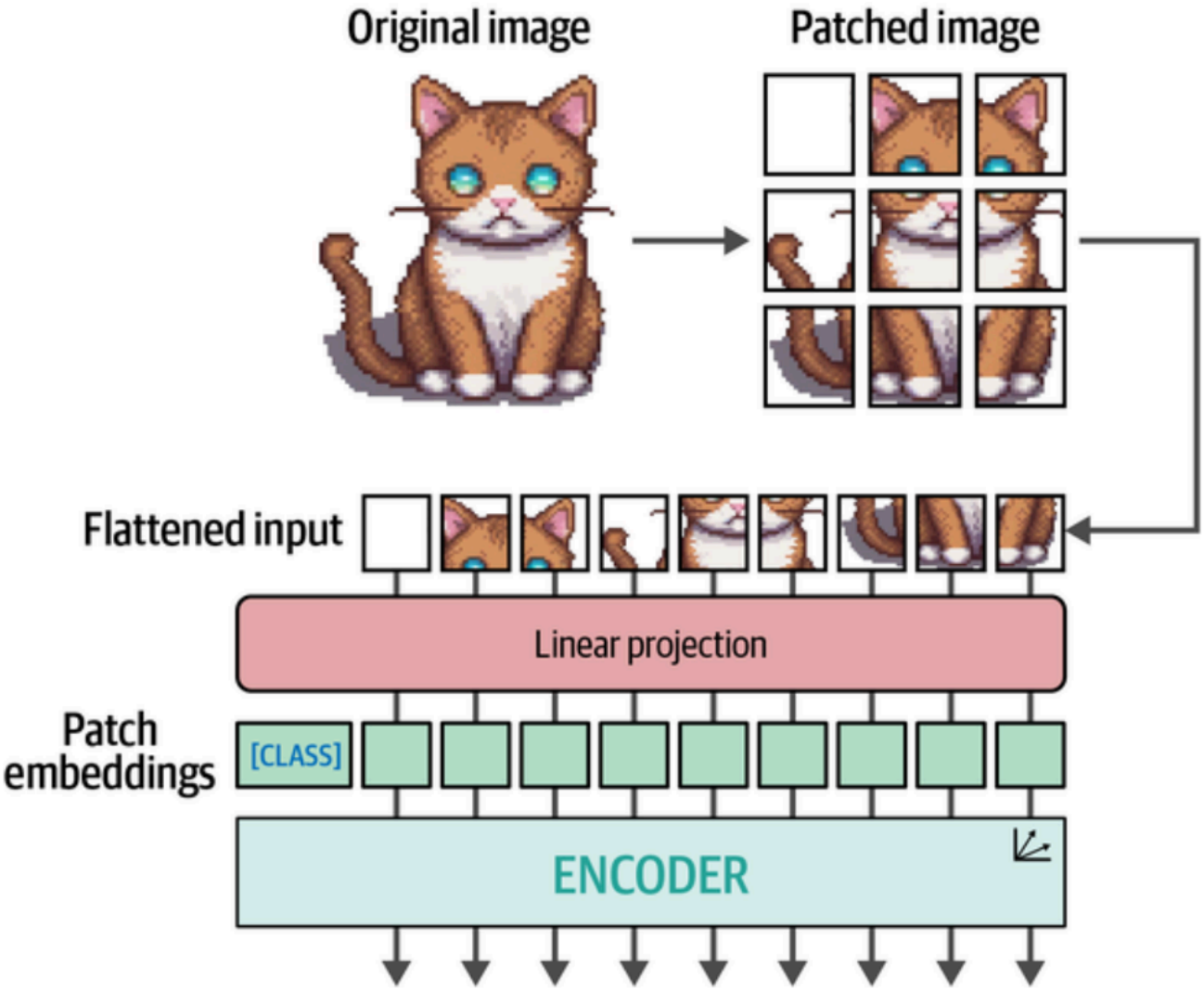


Figure 9-5. The main algorithm behind ViT. After patching the images and linearly projecting them, the patch embeddings are passed to the encoder and treated as if they were textual tokens.

For illustration, the examples here use 3 × 3 patches.

But the original implementation used 16 × 16 patches. The paper is called "An Image is Worth 16x16 Words."

After being transformed into embeddings (from the linear projection layer), there is no difference in how text or images are processed.

Because of this, ViT is often used to make language models multimodal.

Multimodal Embedding Models

In previous chapters, we used embedding models to capture the semantic content of text.

We saw that embeddings can be used to find similar documents, do classification, and perform topic modeling. But as we've just seen, we can also use embedding models (with a bit of modifications) to handle images.

Now, we will look into models that can take in these generated embeddings and be able to compare them against text embeddings in the same embedding space.

These models are called **multimodal embedding models**. The way they work is illustrated below.

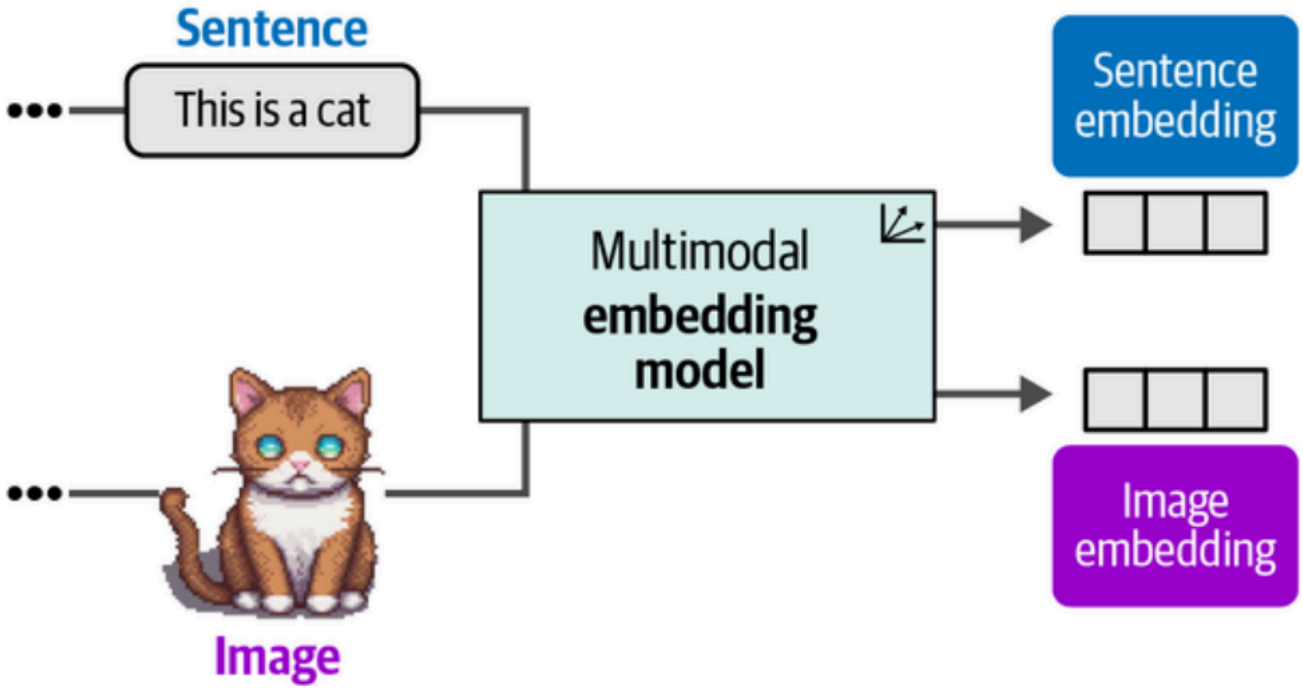


Figure 9-6. Multimodal embedding models can create embeddings for multiple modalities in the same vector space.

As you can imagine, one major advantage of using a multimodal embedding model is its ability to compare/process representations (vectors) of both images and texts.

This ability can be helpful for search engines, databases, and other use cases.

Having embeddings of both images and texts in the same embedding space means a search for 'puppies' using a text query and an image query will work.

For example:
"What is a puppy?" or "Which documents are most related to a given image?"

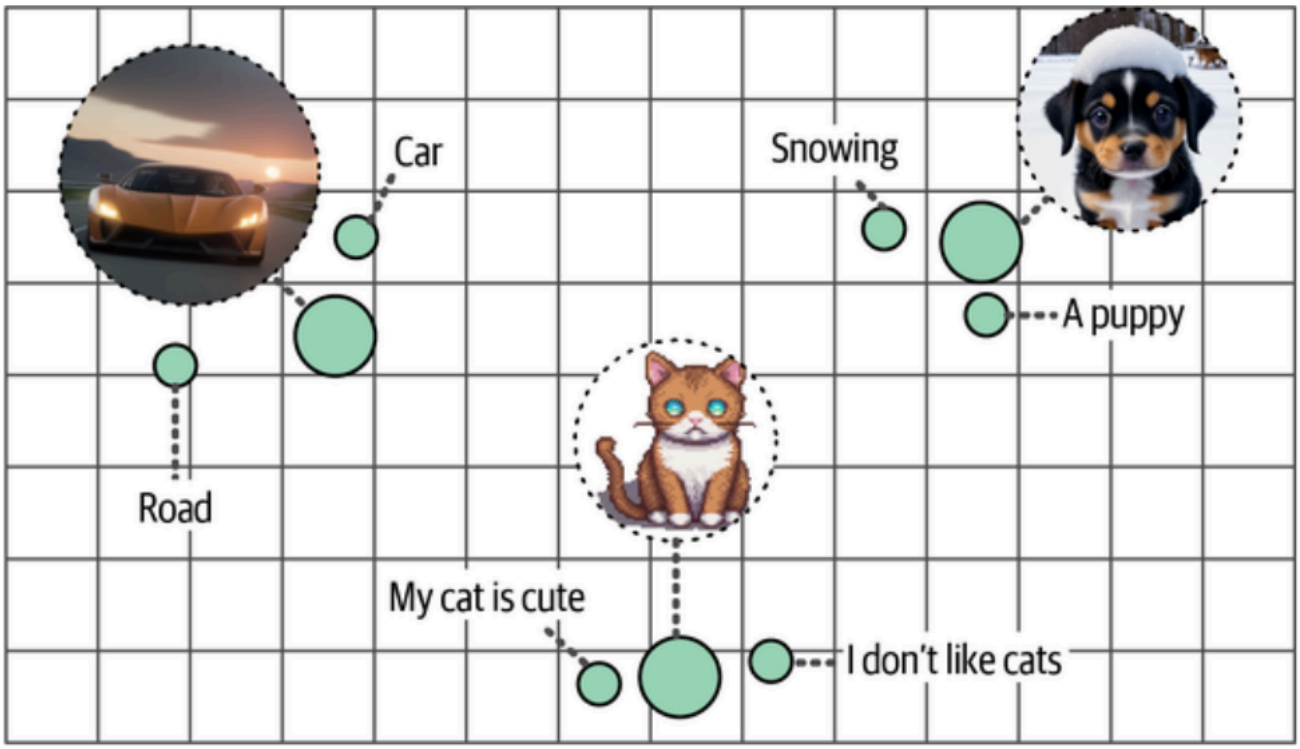


Figure 9-7. Despite having coming from different modalities, embeddings with similar meaning will be close to each other in vector space.

While there are several multimodal embedding models, the most well-known and widely used one is **CLIP (Contrastive Language-Image Pre-training)**.

CLIP: Connecting Text and Images

CLIP is an embedding model that can compute embeddings of both images and text.

The embeddings from CLIP lie in the same vector space.

As we've discussed shortly, this means you can compare image embeddings with text embeddings.

Because of its comparison capability, **CLIP** can be used for tasks such as:

- **Zero-shot Classification**
  Classes can be compared and most similar ones can be found using an embedding of, say, an image. Classify content into predefined categories by comparing embeddings, without needing training examples for each class.
- **Clustering**

  Cluster images and keywords together to see which keywords belong to which sets of images. This is helpful for exploratory analysis, as it reveals patterns and groups without requiring queries. As we know, the closer the embeddings are in the embedding space, the more similarities they share.

- **Search**
  Because it enables similarity search through embeddings, it is easy to find relevant text or images based on an input text or image.
- **Generation**
  CLIP, in this case, can aid generative models like Stable Diffusion by enabling text-to-image generation through a shared embedding space. The model converts the text prompt to an embedding, then iteratively generates an image while using CLIP to compare the embedding of the image being generated against the text embedding. This iterative process guides the generative model to adjust each step according to the similarity comparison, increasingly matching the text description's embedding.

How Can CLIP Generate Multimodal Embeddings?

CLIP's training process is straightforward. Imagine you have a dataset with millions of images and their captions.

Figure 9-8. The type of data that is needed to train a multimodal embedding model.

The dataset contains millions of such pairs, each consisting of one image and its corresponding text caption.

CLIP uses an image encoder and a text encoder, which we've discussed how each of them work previously.

After the embeddings for each are created, their similarity is compared using cosine similarity.

Cosine similarity is calculated by taking the dot product of the embeddings and dividing by the product of their norms (lengths).

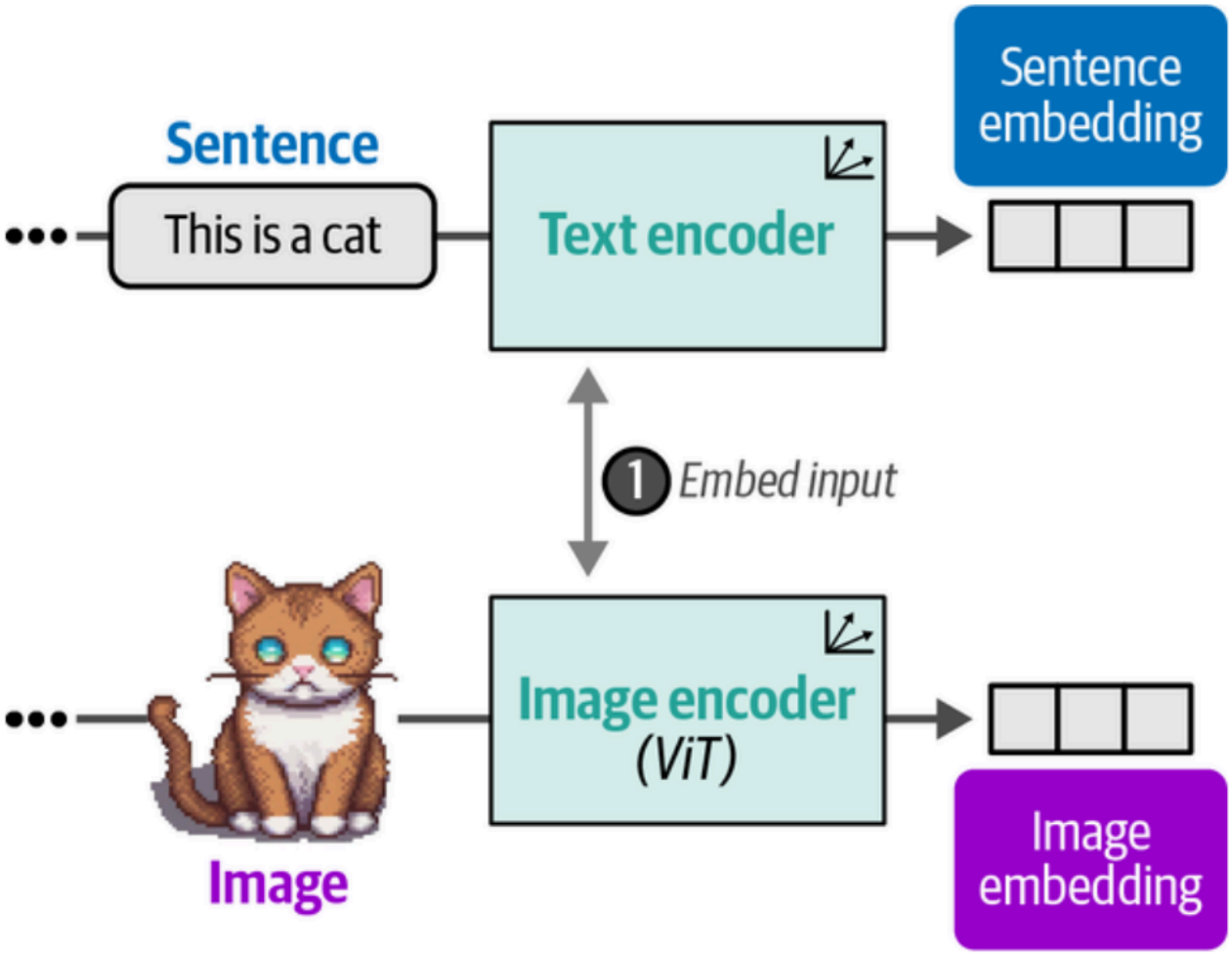


Figure 9-9. In the first step of training CLIP, both images and text are embedded using an image and text encoder, respectively.

When training starts, because the image and text encoders have random weights, their embeddings lie in different embedding spaces and their similarity would be random and unpredictable. The loss function updates both the image and text encoders' weights through backpropagation, gradually aligning their embeddings into a shared space.

As training progresses, the similarity between matching pairs is maximized while the similarity between non-matching pairs is minimized.

This method is called **contrastive learning**.

Unlike traditional classification which predicts a specific class label, contrastive learning focuses on **relative** similarity. It learns by ranking: making sure the correct match has higher similarity than all non-matching alternatives in the batch, rather than assigning absolute class labels.

We'll look at how it works in more detail in Chapter 10, where we'll create our own embedding model.

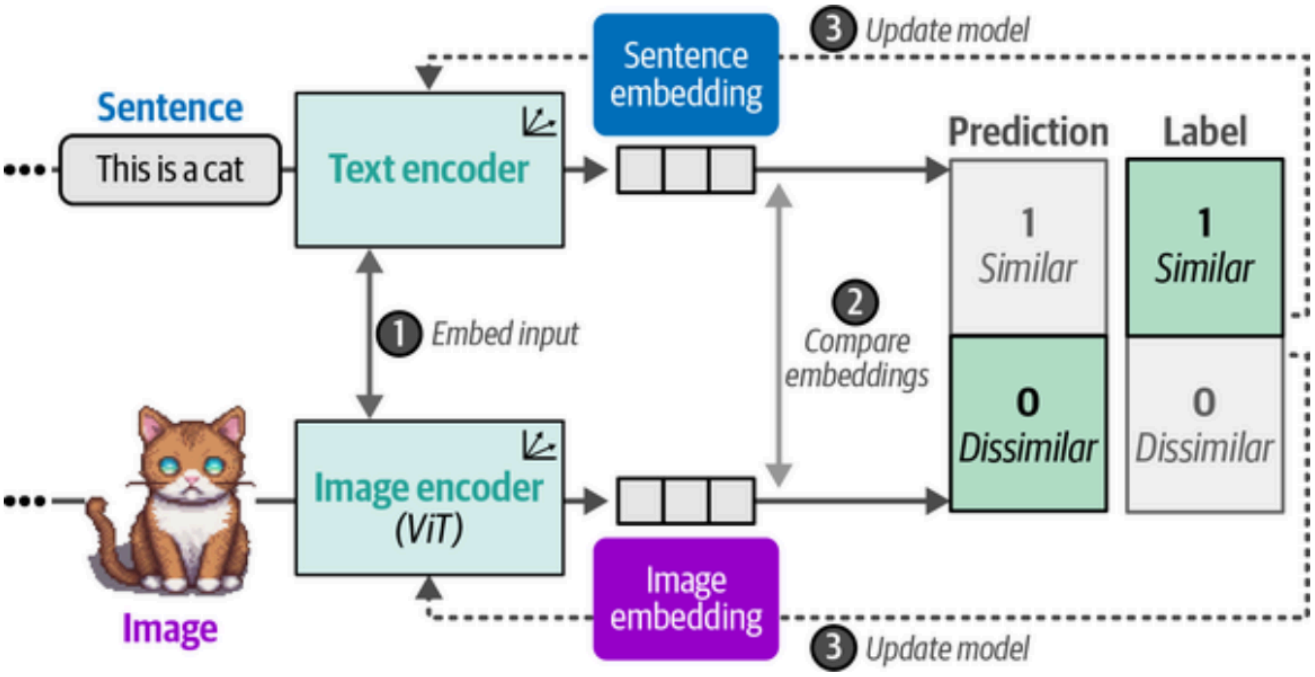


Figure 9-11. In the third step of training CLIP, the text and image encoders are updated to match what the intended similarity should be. This updates the embeddings such that they are closer in vector space if the inputs are similar.

Eventually, the embedding of an image of a cat should be similar to the embedding of "a picture of a cat."

To make sure representations are accurate, negative examples should be included in training.

These are images and captions that are not related. Modeling similarity means knowing what makes things similar. But it also means knowing what makes them different.

OpenCLIP

For our next example, we'll use **OpenCLIP**, the open source variant of CLIP.

Using OpenCLIP (or any CLIP model) comes down to two things:

1. Processing the textual and image inputs
2. Then passing them to the main model

Let's look at a small example.

We'll use an AI-generated image of a puppy playing in the snow:

```
from urllib.request import urlopen
from PIL import Image

# Load an AI-generated image of a puppy playing in the snow
```

```
puppy_path = "https://raw.githubusercontent.com/HandsOnLLM/Hands-On-Large-Language-Models/main/chapter09/images/puppy.png"
image = Image.open(urlopen(puppy_path)).convert("RGB")
caption = "a puppy playing in the snow"
```

Figure 9-12. An AI-generated image of a puppy playing in the snow.

Since we have a caption for this image, we can use OpenCLIP to generate embeddings for both.

To do so, we load three models:

- A **tokenizer** for tokenizing the textual input
- A **preprocessor** to preprocess and resize the image
- The **main model** that converts the previous outputs to embeddings

```
from transformers import CLIPTokenizerFast, CLIPProcessor, CLIPModel

model_id = "openai/clip-vit-base-patch32"
```

```
# Load a tokenizer to preprocess the text
clip_tokenizer = CLIPTokenizerFast.from_pretrained(model_id)

# Load a processor to preprocess the images
clip_processor = CLIPProcessor.from_pretrained(model_id)

# Main model for generating text and image embeddings
model = CLIPModel.from_pretrained(model_id)
```

After loading the models, preprocessing is straightforward.

We'll see what happens when we preprocess our input:

```
# Tokenize our input
inputs = clip_tokenizer(caption, return_tensors="pt")
inputs
```

This outputs a dictionary that contains the IDs of the input:

```
{'input_ids': tensor([[49406, 320, 6829, 1629, 530, 518, 2583, 49407]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1]])}
```

To see what those IDs represent, we can convert them to tokens using the aptly named `convert_ids_to_tokens` function:

```
# Convert our input back to tokens
clip_tokenizer.convert_ids_to_tokens(inputs["input_ids"][0])
```

This gives us the following output:

```
['<|startoftext|>',
 'a</w>',
 'puppy</w>',
 'playing</w>',
 'in</w>',
 'the</w>',
 'snow</w>',
 '<|endoftext|>']
```

As we've seen in previous chapters, the text is split into tokens.

We also see that the start and end of the text are marked with `<|startoftext|>` and `<|endoftext|>` tokens.

Unlike BERT-style models that use a `[CLS]` token for sequence representation, CLIP uses different approaches for text and images. For text, CLIP uses the end-of-text token's embedding as the sequence representation. For images, CLIP's vision encoder uses a separate learned class token.

Now that we have preprocessed our caption, we can create the embedding:

```
# Create a text embedding
text_embedding = model.get_text_features(**inputs)
text_embedding.shape
```

This results in an embedding that has 512 values for this single string:

```
torch.Size([1, 512])
```

Before we can create our image embedding, we need to preprocess it.

Just like with text, the model expects the input image to have certain characteristics including size and shape.

To do so, we can use the processor that we created before:

```
# Preprocess image
processed_image = clip_processor(
    text=None, images=image, return_tensors="pt"
)["pixel_values"]

processed_image.shape
```

The original image was 512 × 512 pixels.

The preprocessing reduced it to 224 × 224 pixels.

That's the expected size for this model:

```
torch.Size([1, 3, 224, 224])
```

Let's visualize the results of this preprocessing:

```
import torch
import numpy as np
import matplotlib.pyplot as plt
```

```
# Prepare image for visualization
img = processed_image.squeeze(0)
img = img.permute(*torch.arange(img.ndim - 1, -1, -1))
img = np.einsum("ijk->jik", img)

# Visualize preprocessed image
plt.imshow(img)
plt.axis("off")
```

Figure 9-13. The preprocessed input image by CLIP.

To convert this preprocessed image into embeddings, we can call the model as we did before and explore what shape it returns:

```
# Create the image embedding
image_embedding = model.get_image_features(processed_image)
image_embedding.shape
```

This gives us the following shape:

```
torch.Size([1, 512])
```

Notice the shape of the image embedding is the same as the text embedding. This is important because it lets us compare them to see if they're similar using cosine similarity

To do this, we first normalize the embeddings, then we calculate the dot product to get a similarity score:

```
# Normalize the embeddings
text_embedding /= text_embedding.norm(dim=-1, keepdim=True)
image_embedding /= image_embedding.norm(dim=-1, keepdim=True)

# Calculate their similarity
text_embedding = text_embedding.detach().cpu().numpy()
image_embedding = image_embedding.detach().cpu().numpy()
score = np.dot(text_embedding, image_embedding.T)
score
```

This gives us the following score:

```
array([[0.33149648]], dtype=float32)
```

We get a similarity score of 0.33. This is hard to interpret on its own, because we don't know what the model considers low versus high similarity.

To know this, we need to know what this score means relative to other pairs' scores. Let's extend the example with more images and captions.

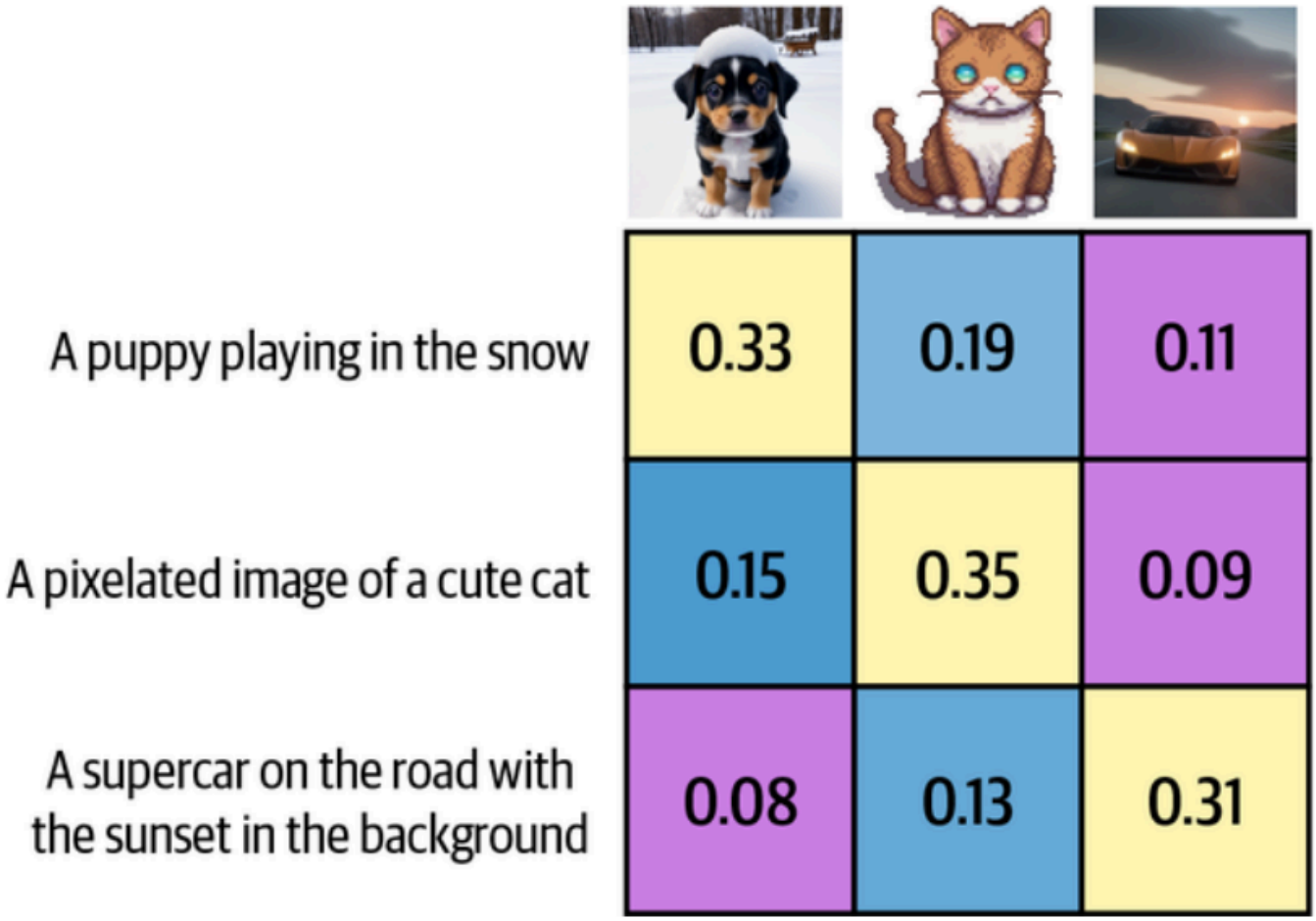


Figure 9-14. The similarity matrix between three images and three captions.

As you can see, a score of 0.33 is actually high in comparison to similarities with other images.

USING SENTENCE-TRANSFORMERS TO LOAD CLIP

`sentence-transformers` implements a few CLIP-based models that make it much easier to create embeddings. It only takes a few lines of code:

```
from sentence_transformers import SentenceTransformer, util

# Load SBERT-compatible CLIP model
model = SentenceTransformer("clip-ViT-B-32")

# Encode the images
image_embeddings = model.encode(images)

# Encode the captions
text_embeddings = model.encode(captions)

# Compute cosine similarities
sim_matrix = util.cos_sim(
```

```
    image_embeddings, text_embeddings
)
```

Making Text Generation Models Multimodal

Text generation models like Llama 2 and ChatGPT work with text. They're good at reasoning about textual information and responding in natural language.

But they're limited to text. As we've seen with multimodal embedding models, adding vision can make a model more capable.

For text generation models, we want them to reason about images.

For example, you could give it an image of a pizza and ask what ingredients it contains.

You could show it a picture of the Eiffel Tower and ask when it was built or where it's located.

This conversational ability is shown below.

Figure 9-15. An example of a multimodal text generation model (BLIP-2) that can reason about input images.

To bridge the gap between vision and language, researchers have worked on adding multimodality to existing models.

One method is called **BLIP-2** (**BLIP-2 stands for Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation 2**).

It's an easy-to-use technique that adds vision capabilities to existing language models.

BLIP-2: Bridging the Modality Gap

Creating a multimodal language model from scratch requires a lot of computing power and data, you'd need billions of images, text, and image-text pairs.

Instead of building from scratch, BLIP-2 builds a bridge between vision and language.

This bridge is called the **Querying Transformer (Q-Former)**. It connects a pretrained image encoder and a pretrained LLM.

Instead of having to train 2 encoder models, BLIP-2 only needs to train the bridge.

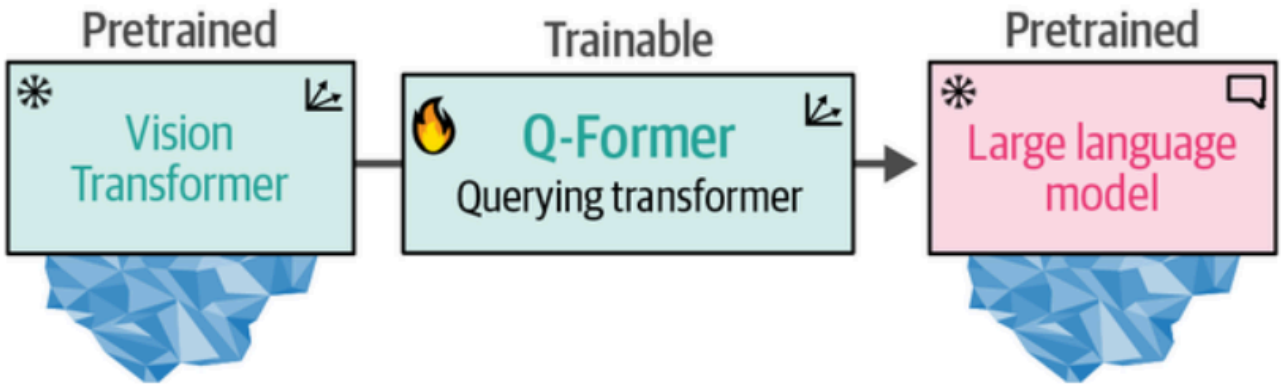


Figure 9-16. The Querying Transformer is the bridge between vision (ViT) and text (LLM) that is the only trainable component of the pipeline.

To connect the two pretrained models, the Q-Former copies their architectures. It has two modules that share attention layers:

- **Image Transformer**: Interacts with the frozen Vision Transformer to extract features
- **Text Transformer**: Interacts with the LLM

The Q-Former is trained in two stages, one for each modality:

- **Step 1**
  In step 1, image-document pairs are used to train the Q-Former. These pairs teach it to represent both images and text.
  The pairs are usually captions of images, like we saw with CLIP. The images images are

input into ViT to generate image embeddings, and the text is input into a text embedding model.

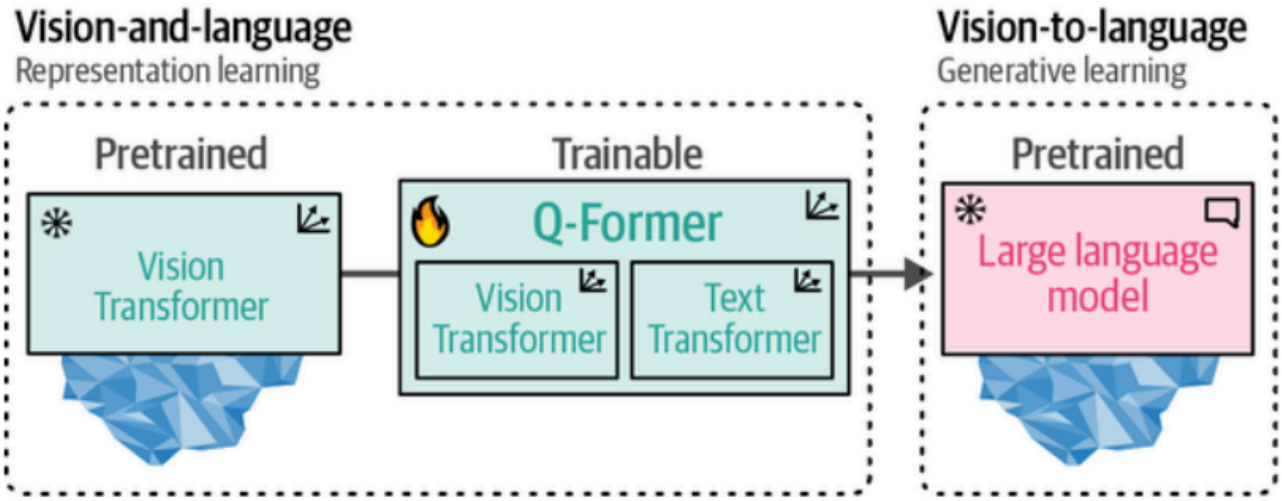


Figure 9-17. In step 1, representation learning is applied to learn representations for vision and language simultaneously. In step 2, these representations are converted to soft visual prompts to feed the LLM.

With these inputs, the Q-Former is trained on three tasks:

- **Image-text contrastive learning**: This task aligns pairs of image and text embeddings to maximize their mutual information.
- **Image-text matching**: A classification task that predicts whether an image and text pair match.
- **Image-grounded text generation**: Trains the model to generate text based on the input image.

These three objectives are optimized together.

This first step of BLIP-2 is shown below.

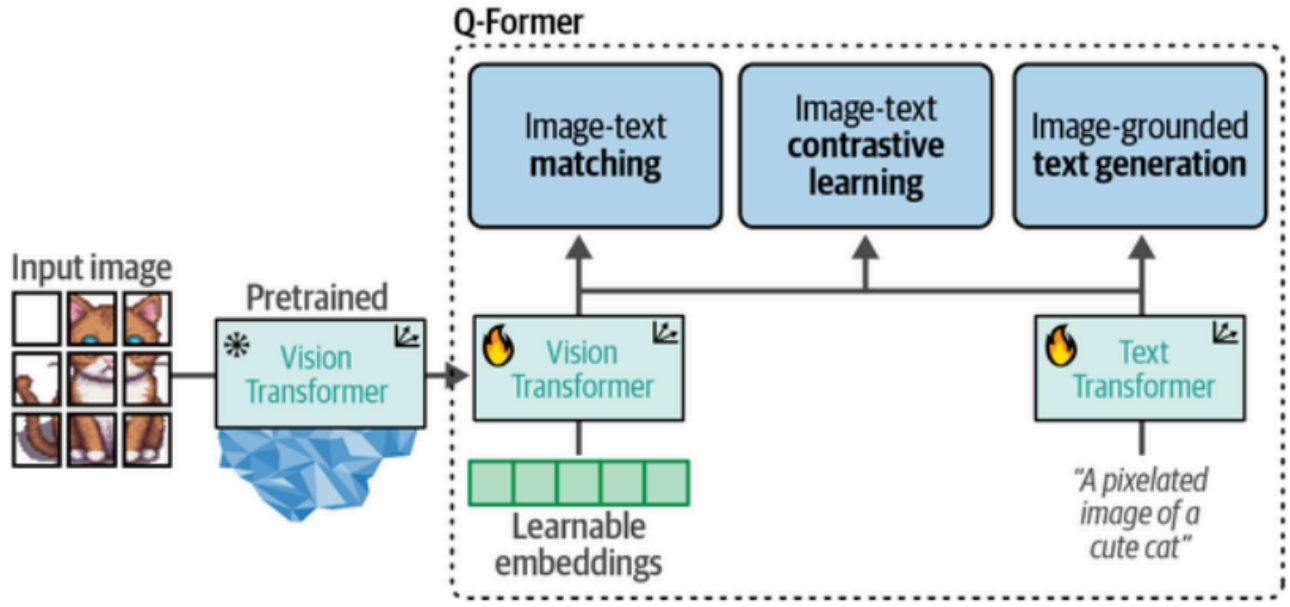


Figure 9-18. In step 1, the output of the frozen ViT is used together with its caption and trained on three contrastive-like tasks to learn visual-text representations.

- **Step 2**
  takes the learnable embeddings from step 1, which now contain visual information in the same dimensional space as text, and passes them to the LLM as soft visual prompts. These embeddings condition the LLM on the visual representations extracted by the Q-Former. A fully connected linear layer sits between them to ensure the embeddings have the right shape for the LLM. This second step converts vision to language.

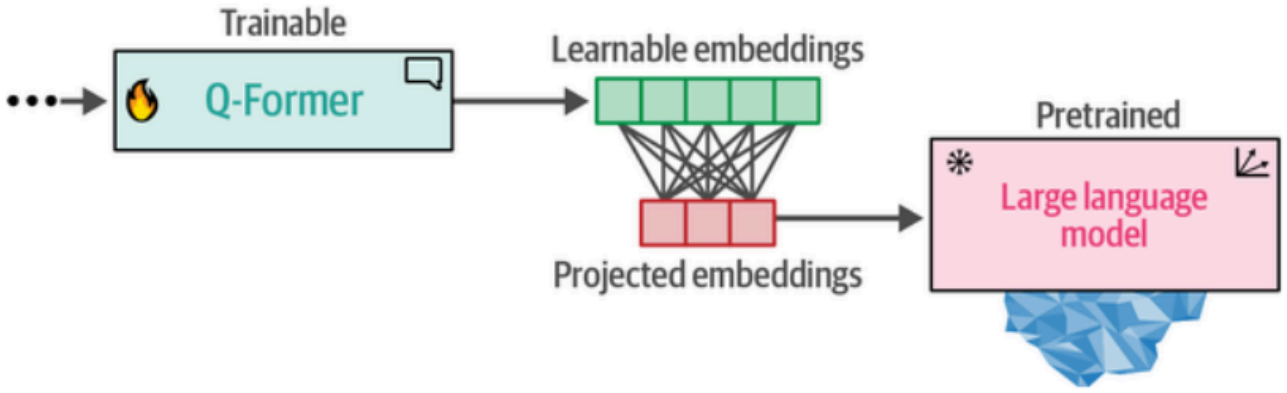


Figure 9-19. In step 2, the learned embeddings from the Q-Former are passed to the LLM through a projection layer. The projected embeddings serve as a soft visual prompt.

When combined, these two training steps enable the Q-Former to learn visual and textual representations in the same space. These representations serve as soft prompts to the LLM, providing it with information about the image similar to the context you'd provide when prompting. The full process is shown below.

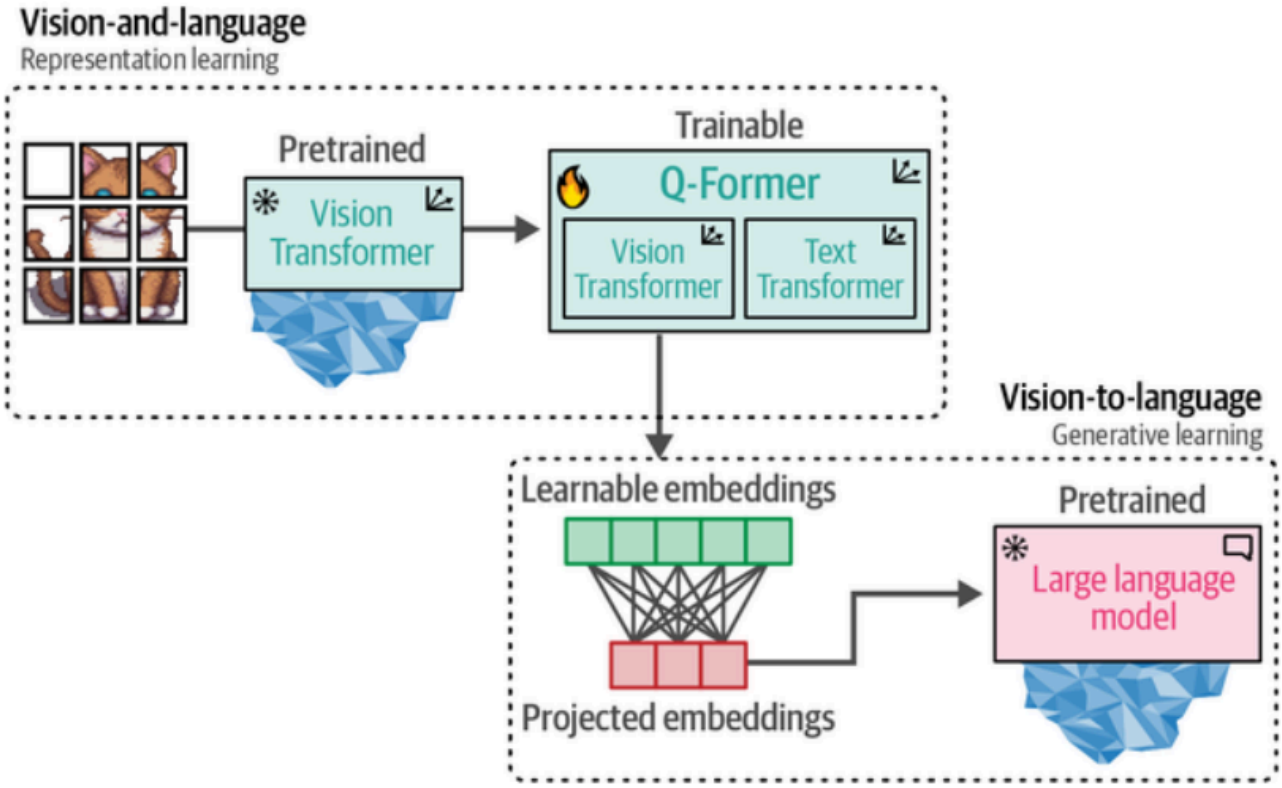


Figure 9-20. The full BLIP-2 procedure.

Since the introduction of BLIP-2, many other visual LLMs have been released.

For example, **LLaVA** is a framework for making textual LLMs multimodal.

Idefics 2 is an efficient visual LLM based on Mistral 7B.

Both models connect pretrained CLIP-like visual encoders with textual LLMs.

Like the Q-Former, they bridge the gap between images and text.

Preprocessing Multimodal Inputs

Now that we know how BLIP-2 is created, let's look at some use cases:

- Captioning images
- Answering visual questions
- Prompting

Before we look at use cases, let's load the model and see how to use it:

```
from transformers import AutoProcessor, Blip2ForConditionalGeneration
import torch

# Load processor and main model
blip_processor = AutoProcessor.from_pretrained("Salesforce/blip2-opt-2.7b")
```

```
model = Blip2ForConditionalGeneration.from_pretrained(
    "Salesforce/blip2-opt-2.7b",
    torch_dtype=torch.float16
)

# Send the model to GPU to speed up inference
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)
```

TIP

Using `model.vision_model` and `model.language_model`, we can see which ViT and generative model are used, respectively, in the BLIP-2 model we loaded.

We loaded two components that make up our full pipeline:

- **Processor**: Like the tokenizer for language models, it converts unstructured input (images and text) to representations the model expects
- **Model**: The main BLIP-2 model

Necessary adjustments to code

The code in the book and in the lab notebooks may not work. Add the following line:

```
if blip_model.config.text_config.bos_token_id is None:
    blip_model.config.text_config.bos_token_id =
blip_proc.tokenizer.bos_token_id
if blip_model.config.image_token_index is None:
    blip_model.config.image_token_index =
blip_proc.tokenizer.convert_tokens_to_ids('<image>')
```

Preprocessing images

Let's explore what the processor does to images.

We'll start by loading a very wide image for illustration:

```
# Load image of a supercar
car_path = "https://raw.githubusercontent.com/HandsOnLLM/Hands-On-Large-
Language-Models/main/chapter09/images/car.png"
```

```
image = Image.open(urlopen(car_path)).convert("RGB")
image
```

The image has 520 × 492 pixels, which is generally an unusual format. So let's see what our processor does to it:

```
# Preprocess the image
inputs = blip_processor(image, return_tensors="pt").to(device,
torch.float16)
inputs["pixel_values"].shape
```

This gives us the following shape:

```
torch.Size([1, 3, 224, 224])
```

The result is a 224 × 224 image.

Much smaller than what we started with. This means all images are processed into squares.

Preprocessing text

Let's continue exploring the processor with text.

First, we can access the tokenizer it uses:

```
blip_processor.tokenizer
```

This gives us the following output:

```
GPT2TokenizerFast(name_or_path='Salesforce/blip2-opt-2.7b',
vocab_size=50265,
model_max_length=1000000000000000019884624838656, is_fast=True,
padding_side='right', truncation_side='right', special_tokens=
{'bos_token': '</s>', 'eos_token': '</s>', 'unk_token': '</s>',
'pad_token': '<pad>'}, clean_up_tokenization_spaces=True),
added_tokens_decoder={
1: AddedToken("<pad>", rstrip=False, lstrip=False,
single_word=False, normalized=True, special=True),
2: AddedToken("</s>", rstrip=False, lstrip=False,
single_word=False, normalized=True, special=True),
}
```

This BLIP-2 model uses a GPT2Tokenizer. As we saw in Chapter 2, tokenizers can work differently.

Let's see how GPT2Tokenizer works.

We'll convert the sentence to token IDs, then convert them back to tokens:

```
# Preprocess the text
text = "Her vocalization was remarkably melodic"
token_ids = blip_processor(image, text=text, return_tensors="pt")
token_ids = token_ids.to(device, torch.float16)["input_ids"][0]

# Convert input ids back to tokens
tokens = blip_processor.tokenizer.convert_ids_to_tokens(token_ids)
tokens
```

This gives us the following tokens:

```
['</s>', 'Her', 'Ġvocal', 'ization', 'Ġwas', 'Ġremarkably',
'Ġmel', 'odic']
```

When we inspect the tokens, you might notice a strange symbol: `Ġ`.

This is actually supposed to be a space. An internal function takes characters and moves them up by 256 code points to make them printable.

The space (code point 32) becomes `Ġ` (code point 288).

We'll convert them to underscores for clarity:

```
# Replace the space token with an underscore
tokens = [token.replace("Ġ", "_") for token in tokens]
tokens
```

This gives us a nicer output:

```
['</s>', 'Her', '_vocal', 'ization', '_was', '_remarkably',
'_mel', 'odic']
```

The output shows that the underscore marks the beginning of a word.

This way, words made up of multiple tokens can be recognized.

Use Case 1: Image Captioning

The simplest use of **BLIP-2** is to create captions for images.

Example use cases:

- A store might want descriptions of its clothing
- A photographer might need to label 1,000+ pictures from a wedding

The captioning process follows these steps:

1. An image is converted to pixel values the model can read
2. These pixels are passed to BLIP-2
3. They're converted into soft visual prompts
4. The LLM uses these to decide on a caption

Let's take the image of a supercar and use the processor to derive pixels in the expected shape:

```
# Load an AI-generated image of a supercar
image = Image.open(urlopen(car_path)).convert("RGB")

# Convert an image into inputs and preprocess it
inputs = blip_processor(image, return_tensors="pt").to(device,
torch.float16)
image
```

The next step is converting the image into token IDs using BLIP-2.

We then convert the IDs into text. This gives us the generated caption:

```
# Generate image ids to be passed to the decoder (LLM)
generated_ids = model.generate(**inputs, max_new_tokens=20)

# Generate text from the image ids
generated_text = blip_processor.batch_decode(
    generated_ids, skip_special_tokens=True
)
generated_text = generated_text[0].strip()
generated_text
```

`generated_text` contains the caption:

```
an orange supercar driving on the road at sunset
```

This seems like a perfect description for this image.

Image captioning is a good way to experiment with this model. When you try it with different images yourself, you'll notice where it performs well and where it struggles.

The model tends to work best with common, everyday scenes because it was trained on public data.

Domain-specific content like specific cartoon characters or imaginary creations will likely confuse it since these weren't part of its training data.

Let's end with another example: an image from the Rorschach test. If you're not familiar, the Rorschach test is an old psychological experiment where people interpret inkblots, and what they see supposedly reveals aspects of their personality. It's entirely subjective, which makes it interesting for this model.

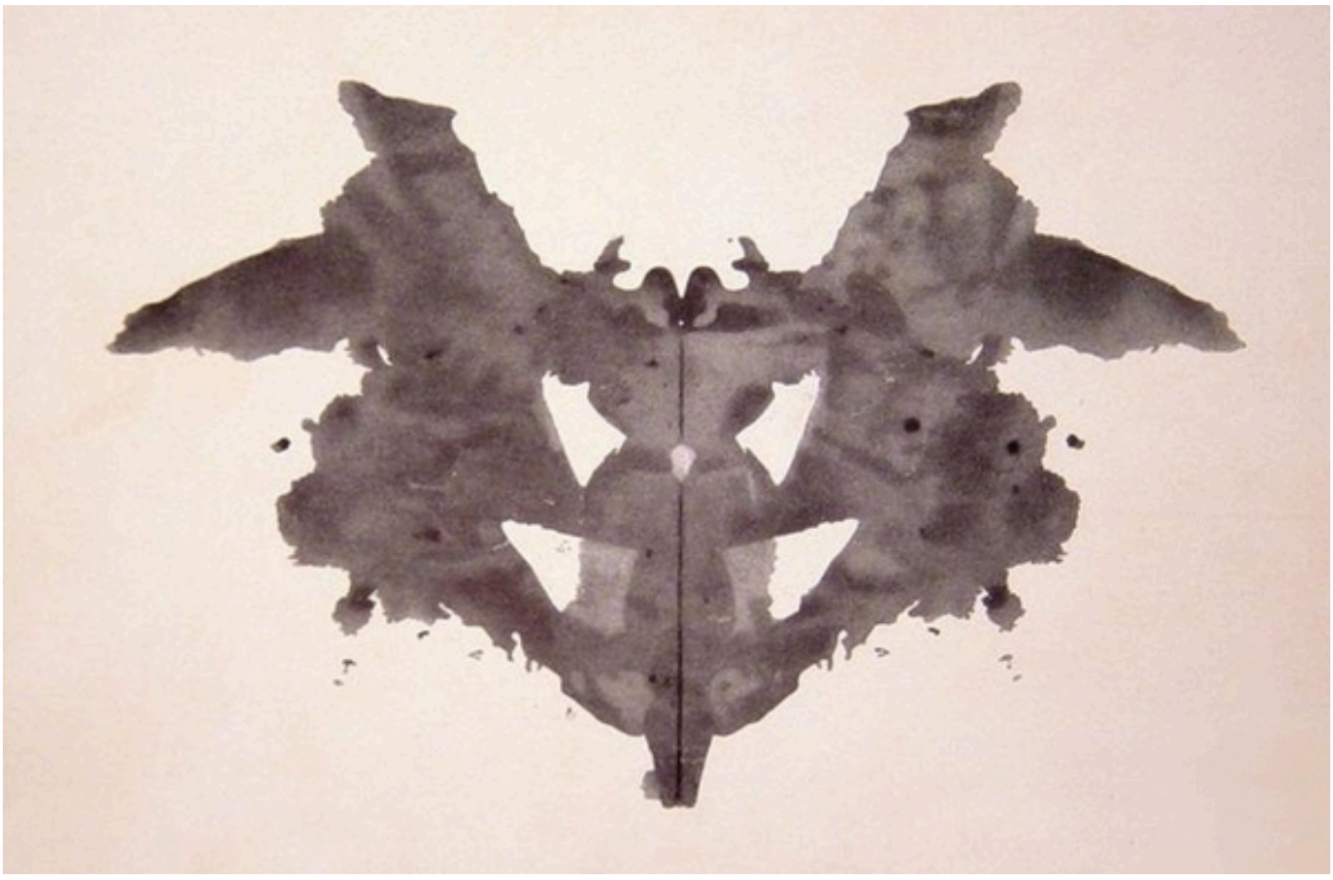


Figure 9-21. An image from the Rorschach test. What do you see in it?

Let's take the image and use that as our input:

```
# Load Rorschach image
url =
```

```
"https://upload.wikimedia.org/wikipedia/commons/7/70/Rorschach_blot_01.jpg"
image = Image.open(urlopen(url)).convert("RGB")

# Generate caption
inputs = blip_processor(image, return_tensors="pt").to(device,
torch.float16)
generated_ids = model.generate(**inputs, max_new_tokens=20)
generated_text = blip_processor.batch_decode(
    generated_ids, skip_special_tokens=True
)
generated_text = generated_text[0].strip()
generated_text
```

As before, when we inspect the `generated_text` variable, we can take a look at the caption:

```
a black and white ink drawing of a bat
```

Use Case 2: Multimodal Chat-Based Prompting

Captioning is useful, but we can extend it further.

In the previous example, we went from one modality (vision) to another (text). Instead of this linear structure, we can present both modalities at once. This is called **visual question answering**:

- We give the model an image and a question about that image
- The model needs to process both the image and the question together

Let's demonstrate with a picture of a car.

We'll ask BLIP-2 to describe the image, but first we preprocess the image like before:

```
# Load an AI-generated image of a supercar
image = Image.open(urlopen(car_path)).convert("RGB")
```

For visual question answering, we need to give BLIP-2:

- The image
- A prompt (without it, the model would just generate a caption)

Ask the model to describe the image:

```
# Visual question answering
prompt = "Question: Write down what you see in this picture. Answer:"
```

```
# Process both the image and the prompt
inputs = blip_processor(image, text=prompt, return_tensors="pt").to(device,
torch.float16)

# Generate text
generated_ids = model.generate(**inputs, max_new_tokens=30)
generated_text = blip_processor.batch_decode(
    generated_ids, skip_special_tokens=True
)
generated_text = generated_text[0].strip()
generated_text
```

This outputs:

```
A sports car driving on the road at sunset
```

It correctly describes the image, but this is a simple example.

Our question basically asks for a caption. We can do more by asking follow-up questions in a chat-based way.

To do this, we give the model our previous conversation, including its answer to our first question.

Then we ask a follow-up question:

```
# Chat-like prompting
prompt = "Question: Write down what you see in this picture. Answer: A
sports car driving on the road at sunset. Question: What would it cost me to
drive that car? Answer:"

# Generate output
inputs = blip_processor(image, text=prompt, return_tensors="pt").to(device,
torch.float16)
generated_ids = model.generate(**inputs, max_new_tokens=30)
generated_text = blip_processor.batch_decode(
    generated_ids, skip_special_tokens=True
)
generated_text = generated_text[0].strip()
generated_text
```

This gives us the following answer:

```
$1,000,000
```

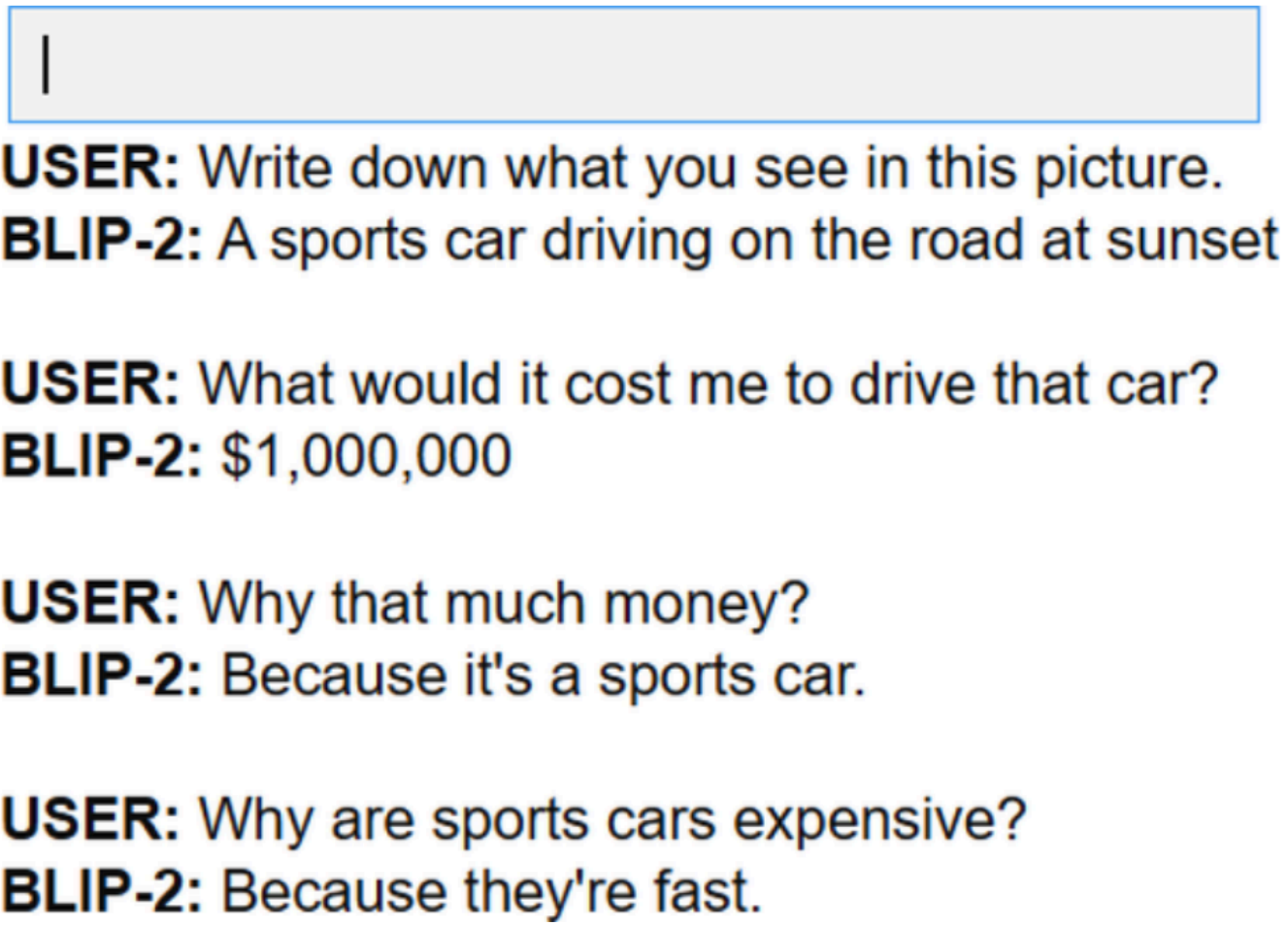
$1,000,000 is highly specific.

We can make this process smoother by creating an interactive chatbot. Let's use `ipywidgets`, an extension for Jupyter notebooks to let us create interactive buttons, input text, etc:

```
from IPython.display import HTML, display
import ipywidgets as widgets

def text_eventhandler(*args):
    question = args[0]["new"]
    if question:
        args[0]["owner"].value = ""

        # Create prompt
        if not memory:
            prompt = " Question: " + question + " Answer:"
        else:
            template = "Question: {} Answer: {}."
            prompt = " ".join(
                [
                    template.format(memory[i][0], memory[i][1])
                    for i in range(len(memory))
                ]
            ) + " Question: " + question + " Answer:"

        # Generate text
        inputs = blip_processor(image, text=prompt, return_tensors="pt")
        inputs = inputs.to(device, torch.float16)
        generated_ids = model.generate(**inputs, max_new_tokens=100)
        generated_text = blip_processor.batch_decode(
            generated_ids,
            skip_special_tokens=True
        )
        generated_text = generated_text[0].strip().split("Question")[0]

        # Update memory
        memory.append((question, generated_text))

        # Assign to output
        output.append_display_data(HTML("<b>USER:</b> " + question))
        output.append_display_data(HTML("<b>BLIP-2:</b> " + generated_text))
        output.append_display_data(HTML("<br>"))
```

```
# Prepare widgets
in_text = widgets.Text()
in_text.continuous_update = False
in_text.observe(text_eventhandler, "value")
output = widgets.Output()
memory = []

# Display chat box
display(
    widgets.VBox(
        children=[output, in_text],
        layout=widgets.Layout(display="inline-flex", flex_flow="column-
reverse"),
    )
)
```

The above code creates an interactive chatbot interface.



This allows for a chat-like conversation, that takes the chat history, between the user and the model.