Chapter 1. Introduction to Vision and Language

**A Note for Early Release Readers**

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *shunter@oreilly.com*.

We are social and visual animals. We evolved in groups, depending on one another for survival. Our brains are wired for communication, empathy, and connection. Even our emotional well-being is deeply tied to our relationships.

To strengthen our bonds, we developed language, a tool to optimize communication, and, soon after, writing systems that allowed us to reach beyond the limits of face-to-face interaction.

Yet, among all our senses, vision dominates our perception of the world. A huge part of our brain is dedicated to processing visual information. We rely on sight to understand our environment, navigate spaces, recognize others, and pick up subtle social cues like facial expressions and body language.

It’s no surprise, then, that when we set out to build intelligent systems, we were captivated by two grand challenges: teaching them *language* to communicate with us, and *vision* to perceive the world as we do.

At first, our systems were limited to handling either language or vision, but not both together. Over time, however, a set of breakthroughs allowed us to merge these two rich modalities into unified architectures, giving rise to what we now call*Vision-Language Models*.

In this chapter, we will briefly explore the history of machine learning and computer vision that led to this new paradigm and set the stage for the exciting advances ahead.

Brief Introduction to Computer Vision

Before the deep learning era, computer vision relied heavily on handcrafted feature extraction techniques. Modern methods coexist and inherit from earlier techniques, so understanding them is important.

Let’s briefly take a look at signal decomposition methods, filter & feature extraction kernels. Luckily, kernels are a good segway towards convolutional neural networks and vision transformers.

**Signal Decomposition Techniques**

While the name might sound intimidating, signal decomposition techniques are straightforward. Rather than looking directly at the pixel values of an image, they convert those leveraging sines, cosines and wavelets. Those new representations are to enable more efficient processing.

Several important transforms are commonly used; we’ll talk about three here. The first one is Fourier Transform, which decomposes images into frequency components, revealing patterns (like waves) that characterize image structures. These characteristics are often more readily analyzed or manipulated in the frequency domain than in the spatial domain. Fourier analysis became essential for filtering operations and was foundational for many early compression algorithms. It is also widely used for audio processing.

The Discrete Cosine Transform (DCT) similarly analyzes frequency content but focuses energy into a small number of coefficients, making it especially effective for compression. DCT is the cornerstone of JPEG compression and many video codecs like MPEG.

Lastly, Wavelet Transforms provide multi-resolution analysis, allowing for both spatial and frequency localization. This approach excels in lossless image compression (e.g., JPEG2000) and denoising applications. These transforms enable efficient processing by reducing dimensionality while preserving essential information.

In [Figure 1-1](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_1_1749495171820118), we take an image and convert it to the DCT space. In the DCT space, u and v are the horizontal and vertical frequency indices that define the number of cosine wave oscillations across a block. A high u with low v represents rapid changes left-to-right but smooth top-to-bottom, capturing vertical edges or textures. Conversely, low u with high v captures horizontal edges. Low values of both indicate smooth regions, while high values of both represent fine, detailed textures. Each (u, v) coefficient shows how much that directional pattern contributes to the image block.

When we look at the energy distribution in the space (second image in the figure) we can see that most of the energy of the original image is concentrated in a few coefficients in the DCT space, particularly those in the upper-left corner (low u,v values). JPEG and other image compression algorithms take advantage of this by thresholding the coefficients in the DCT space and keeping only those with high energy values (third image in the figure). Why does this work? This works because the coefficients closer to u=0, v=0 correspond to low frequencies, which are the most important for human visual perception. Higher frequencies (larger u,v values) represent fine details that our eyes are less sensitive to, so removing these coefficients reduces the file size significantly while maintaining perceptual quality. This property of natural images— their energy concentrates in low-frequency components—makes DCT-based compression so effective. This and many other media compression techniques would probably be ineffective for an alien civilization.

Even today, FFTs, DCT and Wavelet transforms remain in use in phone calls, video streams and media archives among many other applications.

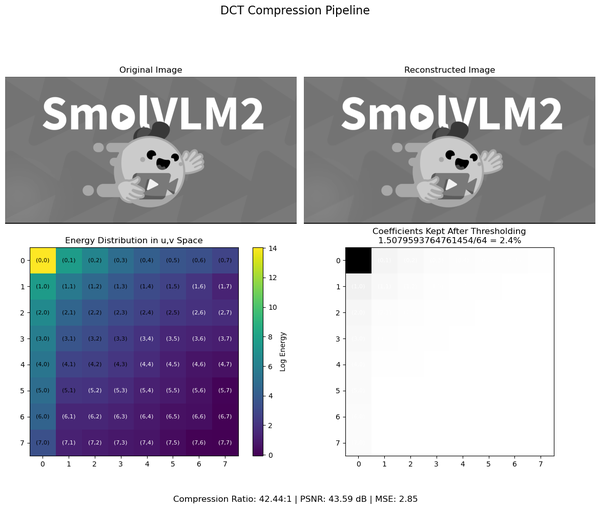


Figure 1-1. DCT compression pipeline

**Filters and Feature extraction kernels**

We are getting close to the two main breakthroughs that made modern computer vision so powerful: convolutional neural networks (CNNs) and vision transformers (ViTs)

We don’t want to be a buzz killer, but before digging into CNNs and ViTs, let’s talk about feature extraction kernels.

Understanding feature extraction kernels isn’t just prehistory—it’s the DNA of modern vision. What we manually designed yesterday, CNNs learn automatically today. These patterns are exactly what your CNN’s first layers will discover on their own; learning a bit about feature extraction kernels will give you a better understanding of why CNNs and later Vision Transformers work.

A*feature extraction kernel* is a small matrix (also called a filter) that is applied over an image to detect specific patterns. At each location, the kernel values multiply the underlying pixel values, and the results are summed to produce a new value.

Let’s visualize this intuitively. [Figure 1-2](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_2_1749495171820160) shows a sequence of values that change along a single dimension, this is a single dimension signal.

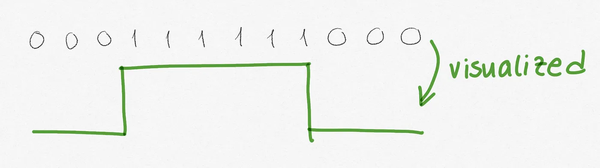


Figure 1-2. Visualization of a 1-D signal

As we are working with a one-dimensional signal in our example, let’s also assume a one-dimensional kernel. To be specific, let’s set our kernel to [-1 1].

In order to filter our 1D signal with our 1D kernel, we multiply the values of the signal by the kernel, we sum and write down the numbers, starting from left as shown in [Figure 1-3](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_3_1749495171820182). We do this successively for all the values of the signal.

Note that we invented a value in our signal! This is called padding, and it is just to avoid having a gap when we apply the filter. There are many different padding strategies, adding as many zeroes as needed to the left is a common one.

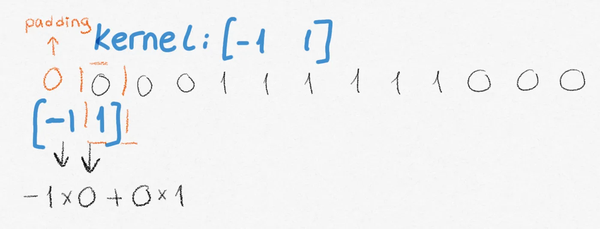


Figure 1-3. Applying a convolution to a sequence

This process is called “convolution”. Let’s visualize original data against convolved data in [Figure 1-4](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_4_1749495171820200) and [Figure 1-5](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_5_1749495171820215).



Figure 1-4. visual input and output of the convolution

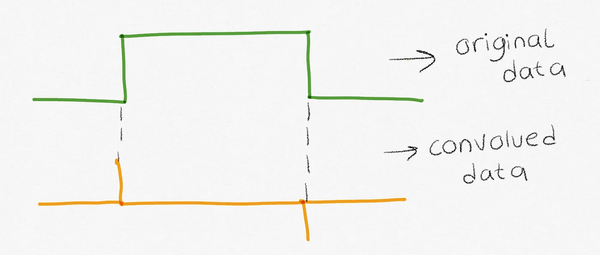


Figure 1-5. Visual input/output

See how this highlights the changes positively and negatively? The output of the convolution is named “feature map” because it highlights the features we can extract. This applies to changes in images too, we can detect edges like this. In fact, in [Figure 1-6](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_6_1749495171820231) you can see the kernels of two popular filters commonly used in image processing pipelines to identify edges.

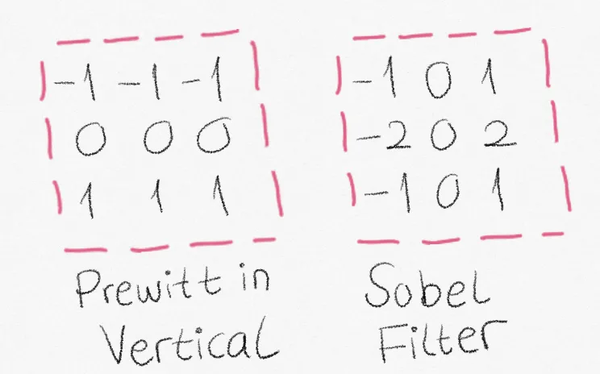


Figure 1-6. Prewitt and Sobel kernels for edge detection

As we just saw, the convolution emphasizes areas of the image where there are rapid changes in intensity — highlighting edges, corners, and other key points (such as SWIFT, defined below). Convolution is fundamental in many image processing tasks because it enables the extraction of meaningful structures from raw pixel data.

**Keypoint Detection and Description**

A significant breakthrough came with algorithms that could identify and describe distinctive points in images, not just provide new representations of the image. Those points and a bit of machine learning could fuel many classification tasks.

The first one is Scale-Invariant Feature Transform (SIFT). SIFT - presented in Figure 1.7 detects keypoints invariant to scaling, rotation, and illumination changes. SIFT descriptors captur local gradient information around these points. To improve SIFT’s computational efficiency while maintaining similar capabilities, Speeded-Up Robust Features (SURF) was developed. Oriented FAST and Rotated BRIEF (ORB) provided an even faster alternative while maintaining reasonable performance.

These algorithms enabled a powerful paradigm: extract a “bag of visual features” from images, then apply traditional machine learning methods (such as k-means clustering and SVMs) to perform tasks like object recognition and image classification.

This race to engineer better-handcrafted features dominated computer vision research for decades. While effective for controlled environments, these approaches struggled with the variability encountered in real-world scenarios—ultimately paving the way for the learning-based methods to revolutionize the field.

In the example of [Figure 1-7](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_7_1749495171820247) we extracted SIFT keypoints of the original image (left) and from a patch of that image (right).

A very popular technique to search for a specific logo in visual content is to construct a database of keypoints (SIFT, SURF, ORB, or others) and check for those keypoints in an image catalog. When a big number of points match across images, we consider a positive matching.

In our example, we extracted 26 SIFT keypoints from the smiley cute face and it matched against 15 of the 294 keypoints extracted from the full image on the left, meaning we probably have a match between a part of the full image on the left and the face detail on the right.

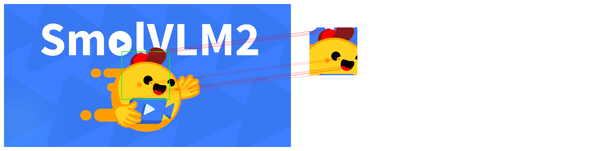


Figure 1-7. Keypoint matching based on SIFT

**From filters to convolutional neural networks**

Before convolutional neural networks (CNN), features like SIFT, SURF, or edge detectors like Prewitt, Sobel, among many other filters were engineered manually. Now, convolutional neural networks find the optimal kernels according to a problem at hand, be it image classification or object detection. It is widely theorized that when a CNN is applied to audio, the first layer learns a type of Fourier Transform, meaning the network learns to decompose audio signals into their frequency components… just as we used to do manually!

At a very rudimentary level, convolutional neural networks look like the following. Assuming you have 32 different kernels that you want to use to extract features from an image, you’ll have 32 resulting feature maps as illustrated in [Figure 1-8](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_8_1749495171820262). When training models, you initialize these kernels randomly, meaning the values in the matrix are random. As the model trains, it learns the best kernels for that dataset.

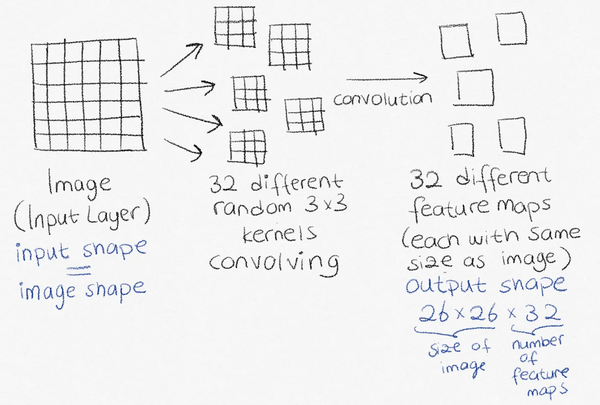


Figure 1-8. Illustrated Feature Maps

CNNs are good for solving classical computer vision tasks such as image classification, object detection, and image segmentation, with image classification operating as shown in [Figure 1-9](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_9_1749495171820277).

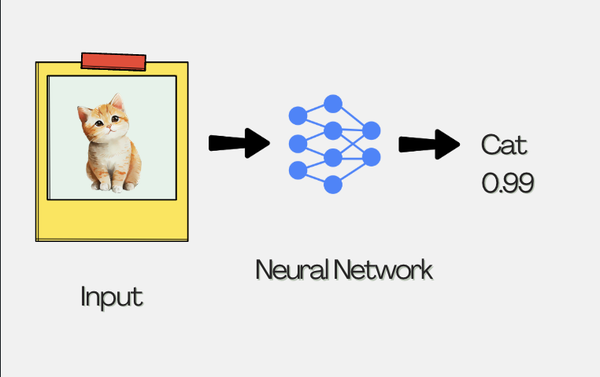


Figure 1-9. Basic Image Classification

Let’s dig in further. You can use convolutional neural networks (CNNs) mostly to extract the features and later feed them to a classification layer. Unlike standard neural networks, CNNs were originally designed for image processing with specialized layers that automatically detect important visual features. A CNN consists of interconnected “nodes” (also called neurons) organized into different types of “layers.” Each node is a computational unit that processes an input, applies an activation function, and produces an output.

The early layers in a CNN are convolutional layers that detect edges, textures, and patterns, while later layers identify more complex features like ears, tails, or fur patterns. These extracted features are eventually passed to a classification layer. Assuming you are classifying between cats and dogs, the final classification layer has two nodes, one for cat and one for dog. These output nodes each produce a probability value indicating how likely the input image is to belong to each class. The output layer contains exactly as many nodes as there are classification options because each node specializes in identifying one specific class, allowing the network to decide by selecting the class with the highest probability, as shown in [Figure 1-10](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_10_1749495171820292).

**Other basic CNN based pipelines**

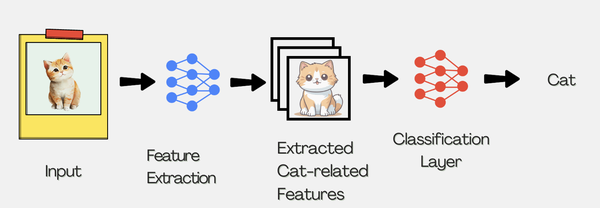


Figure 1-10. CNN with Image Classification head

Let’s talk about a few other CNN pipelines. All those other applications have something in common (in addition to cats in the example): they rely on a common backbone for feature extraction. Think of the backbone as the main body of the neural network that handles the core image processing work. It’s the part that extracts important features from images before any task-specific parts take over.

You briefly saw classification in [Figure 1-10](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_10_1749495171820292) and inspired by the classification pipeline, you can see the object detection pipeline in [Figure 1-11](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_11_1749495171820307).

Can you spot the differences? For object detection, there is a different layer on top of extracted features called *object detection head*.

Wait, what is this? The object detection head identifies multiple objects by predicting both their categories and bounding box coordinates. The final layer outputs class probabilities along with the locations of detected objects of interest within the image, as shown in [Figure 1-11](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_11_1749495171820307).

Unlike a classification head that simply categorizes whole images, the object detection head works on the rich feature maps produced by the backbone network. This head is specifically designed to process these extracted features in a way that enables both localization and classification of multiple objects.

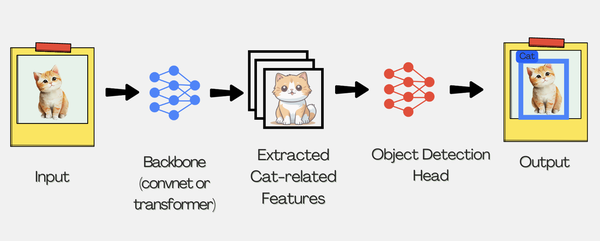


Figure 1-11. Object Detection Pipeline

The next pipeline is image segmentation. Image segmentation is a more fine-grained task that occurs on pixel level, where each pixel is assigned a class probability. When class probabilities are high in adjacent pixels, those pixels form a segmentation mask. Commonly, semantic segmentation assigns a single segmentation mask to all instances of an object in a given image, as shown in [Figure 1-12](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_12_1749495171820322).

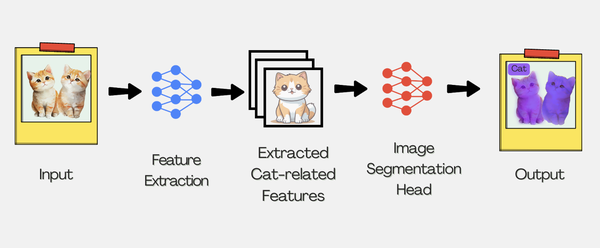


Figure 1-12. Semantic Segmentation

Instance segmentation detects different instances as you can see in the output of [Figure 1-13](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_13_1749495171820336).

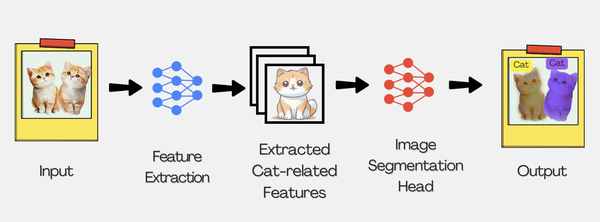


Figure 1-13. Instance Segmentation

Doing a quick recap, each task-specific model above is shaped by the task-specific head, whereas feature extraction layers (or backbone) have little change. When trained on enough data, feature extraction layers detect universal features. Universal features mean edges, corners, and curves. A task-specific feature, for cats and dogs classification, for example, would be a whisker of a cat or the nose of a dog. As you approach the head, these layers are trained to detect lower-level task-specific layers. Given this, does it make any sense to use the backbone for a task that it was not trained for? That’s what you achieve with transfer learning.

**Transfer Learning**

Transfer learning allows practitioners to use large pre-trained models for their specific use cases. Instead of spending compute and time for a model to learn universal features such as edges and corners, or general objects, the model training only focuses on the domain-specific task at hand. With a reduced amount of data and compute, you could repurpose an existing CNN to fulfill your specific application, and this way, transfer learning achieves a good bang for the buck.

To train the large models, you initialize the convolutional neural network kernels randomly, and through the training, the optimization process tweaks the kernels to detect features. When trained on diverse data, the feature extraction kernels become more universal. This means you can repurpose them. This is called “transfer learning”, where you transfer the information learned in a neural network to another neural network for the task at hand. The repurposed models are also called “pre-trained,” and the pre-trained feature extraction models are often referred to as “backbones”. Modern computer vision tasks mostly utilize pre-trained models with a sophisticated task-specific head with feature refiner networks on top of them. These refiners are referred to as *neck*. The neck helps creating richer representations that help the network better understand objects at various scales and positions in the image.

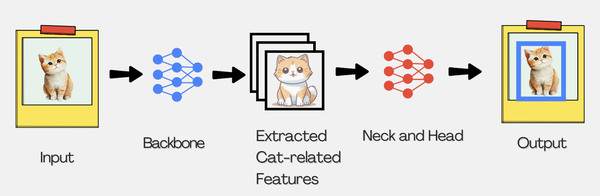


Figure 1-14. Modern computer vision pipelines

But what makes a pre-trained model good enough to be used for transfer?

Many pre-trained models stood the test of time and have been adopted widely by practitioners. However, there is no one-backbone-fits-all type of models, some backbones are used widely across many tasks, and some of them are used the most for a given task. Let’s review some of the most popular backbones.

**ResNet**

The earliest example of a pre-trained model is ResNet. Early convolutional deep neural networks were hard to train due to vanishing and exploding gradients and diminishing errors in accuracy, and then errors in later stages of training. ResNetv1 was the first deep convolutional neural network to solve this using something called “residual connections”. The idea behind residual connections is very simple: pass the input as is to the layer after the next activation. This prevents gradients from vanishing (or exploding) which helps when scaling the network for depth. ResNet employs ReLU activations. ReLU (Rectified Linear Unit) activations sound very fancy but are very simple activation functions: the output is equal to the input if it is positive. If the input is negative, the output is zero. Its simplicity makes ReLU activations highly efficient.

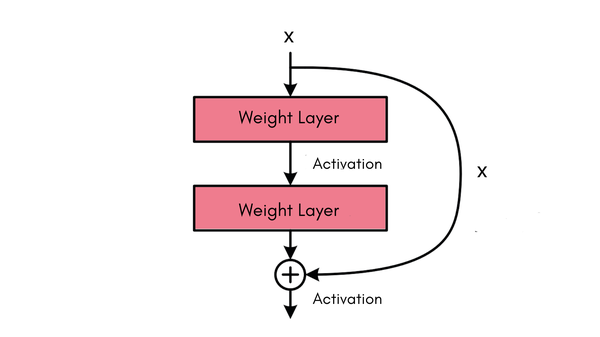


Figure 1-15. Image Description: Illustration of Residual Connection

Directly adding the input is “identity shortcut,”. Another type of residual connection is “projection shortcut” which uses 1x1 convolution to match the input shape to the output shape of a layer. ResNet employs projection shortcut across two consecutive layers.

ResNet has built upon a few other pre-existing ideas to scale to 100+ layers. The first one is bottleneck convolutions, which is simply applying convolutions with filters of size (1, 1) to reduce the dimension before more complex convolutions with filters of size (3,3) or (5, 5). After, the dimensions are rescaled to the original shape by applying another convolution with a filter of size (1, 1). ResNet combines this with residual blocks like [Figure 1-16](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_16_1749495171820383).

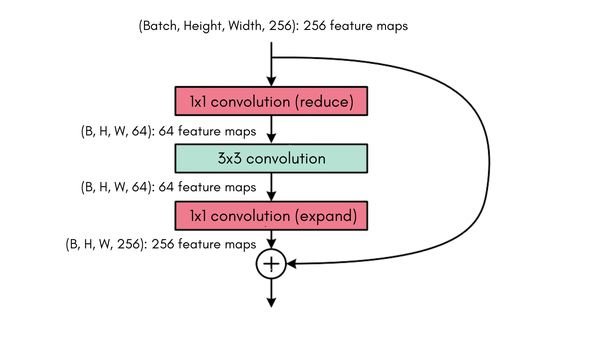


Figure 1-16. Image Description: Bottleneck Residual Layers Illustrated

The last important element in ResNet’s success is batch normalization, forcing layer input batches to have a mean of 0 and standard deviation of 1. This helps stabilize the training of deep neural networks. ResNetv1 normalizes the processed signal after the convolutions and before ReLU activations.

**MobileNet**

MobileNet introduced a fundamental shift in architecture design. Compared to the already very popular ResNet that was focusing on a big number of layers, MobileNet focused on efficiency, making neural networks viable for real-time applications on resource-limited devices without requiring specialized hardware.

Its key innovation lies in depth wise separable convolutions, which divide standard convolutions into two smaller steps: depthwise and pointwise operations. This dramatically reduces the number of parameters and computation by 8-9 times while maintaining comparable accuracy.

**U-Net**

U-Net had a radically different origin and goal than the pre-trained models we’ve discussed so far. It originated from the biomedical world with a focus on image segmentation tasks rather than classification.

While ResNet introduced skip connections to improve gradient flow during training, U-Net expanded this concept with a symmetric “U-shaped” architecture featuring a contracting path that captures context and an expanding path that enables precise localization as shown in [Figure 1-17](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_17_1749495171820407). This enables U-Net to perform pixel-precise segmentation with remarkable accuracy, even when trained on limited datasets (as you can imagine, data scarcity and precision are two common topics in medical imaging)

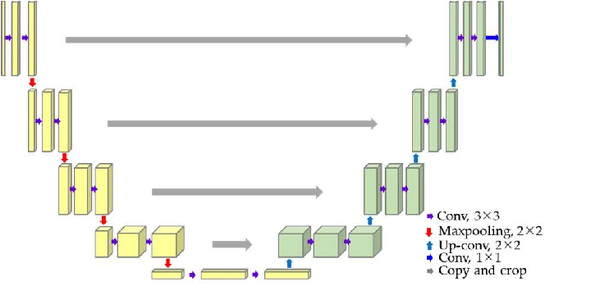


Figure 1-17. Image description: U-Net architecture used for image segmentation, featuring a contracting path on the left side (downsampling) and an expanding path on the right side (upsampling), with skip connections between corresponding layers to preserve spatial information.

U-Net is widely used today in tasks like medical image segmentation, diffusion-based image generation, and semantic segmentation in fields ranging from healthcare and robotics to satellite imaging.

**Transformers and its origins in language**

Let’s start with a brief definition of transformers: transformers are a type of neural network that use attention to understand and process sequences of data more effectively than older models. Transformers were first used in text and rapidly expanded to other content types. We will dig into text processing to understand the path that led to the invention of transformers and its growth to other domains like vision.

Text is handled differently than images. In images there are pixels spread in spatial space and across different channels. In text the data is spread sequentially, and we want to learn semantic relationships between words and how they exist or relate with the real world.

Early natural language processing applications tried to use word to word co-occurrence to estimate the context of a given sentence or a paragraph with very basic models. These applications got us only so far though, we not only wanted the application to understand language, but we wanted it to generate it as well.

The first successful deep learning applications in NLP were based on an architecture called recurrent neural network (RNN). Practitioners divided the text into smaller pieces, building a mapping from these text fragments to numbers called tokens, which were then fed into the RNN. Initially, tokens were often created at the word level, but this approach proved inefficient over time, as more advanced tokenization methods at the subword level allowed for better handling of vocabulary and out-of-vocabulary words.

**RNNs and LSTMs**

An RNN is primarily composed of cells that output a vector, which is repeatedly fed back into the cell along with the next input. In the first step, the cell receives only the first input — for example, a word. In the second step, the cell takes the next word and the output from the previous step as its inputs. These outputs, called hidden states, represent the information the network is learning and carrying forward through the sequence. As we progress through the paragraph, unrolling an RNN reveals how each word is processed step-by-step along with the evolving hidden states, as shown in [Figure 1-18](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_18_1749495171820423).

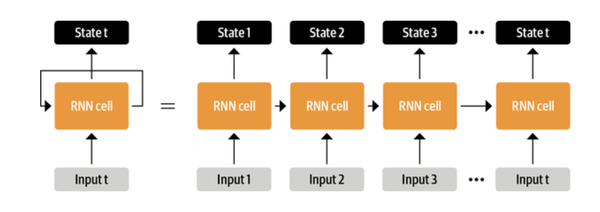


Figure 1-18. Image description: RNN input-state diagram

Naturally, the effect of the initial inputs to the hidden state shrinks as the sequences grow. Moreover, it’s hard for the gradients in the network to be in the small range between vanishing and exploding, making it harder for models to learn. Long-short-term memory networks (LSTM) and gated recurrent units (GRU) were introduced to address these problems. LSTMs simply have “gates”: functions that determine if a given hidden state will be kept, added, removed, or updated. GRUs are simplified versions of LSTMs with gates that update, remove or add new hidden states.

What happens to the last hidden state is one of the key differences in how RNNs are used. The last hidden state is a compressed representation of the input sequence, summarizing what the network has learned from the entire text. This is sometimes referred to as an “encoding.” RNNs that focus only on creating these encodings without producing step-by-step outputs are called “encoders.”

You could feed these encodings to a classification layer to train an RNN on problems like sentiment analysis. This is pretty similar to the transfer learning paradigm we have talked about above, one can have pre-trained text vectors that know about real world and semantic relationships between words and transfer this information to a classification problem.

Since encodings are technically compressions of text, they can be decompressed by another neural network to predict the next word, aka generating text. These networks are called “decoders”. Both encoder-decoder networks and decoder-only networks can predict the next word.

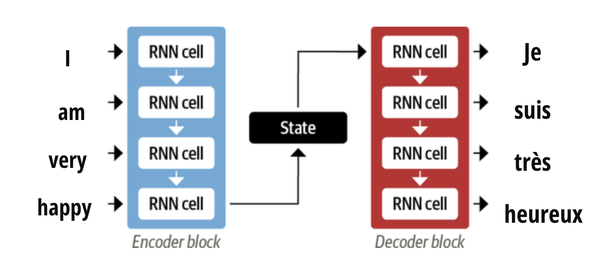


Figure 1-19. Image-description: encoder-decoder LSTM

Encoder-decoder LSTMs are still very limited in the length they can process as the decoder receives a compression of the whole input to decode. You could feed all hidden states of the encoder to all cells of the decoder, but it would be very computationally intensive. To address the limitations of RNNs, and pick which hidden states to particularly *attend to*, an attention mechanism was introduced. Neural networks with an attention mechanism are called transformers, and they are the architecture used by advanced systems such as ChatGPT. We will go through the attention mechanism in detail in upcoming chapters.

**The boom of transformers in text**

Transformers had their initial boom with the BERT model as it enabled multiple successful applications to this day.

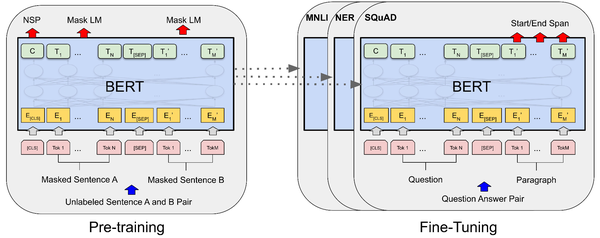


Figure 1-20. Image-description: Bert architecture overview and training approach

BERT was easy to train due to its *self-supervised objective .*Essentially, a small portion of words in the text were masked randomly and the model was trained to predict those words, which allowed it to learn from any corpus of text*.* After the initial training, BERT can be used in transfer learning. The machine learning community has trained many BERT models in different languages and domains thanks to the lack of dependency on labelled data.

BERT is the most successful encoder-only model: it revolutionized NLP by achieving state-of-the-art results across numerous language tasks.

GPT-2 was the most successful decoder-only model, it demonstrated remarkable text generation capabilities across diverse domains with minimal task-specific training. This model is trained through masking the next word and trying to predict it. This is called “next token prediction” or “autoregressive training” and shown in [Figure 1-21](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_21_1749495171820468).



Figure 1-21. Autoregressive modelling

T5is another model that became popular given its possibilities to accomplish a wide range of tasks, training on instruction-answer pairs (see [Figure 1-22](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_22_1749495171820482) to see examples of instruction-answer pairs). Its possibilities covered translation, summarization to detecting sentiments, or assessing entailment between two sentences. T5 was the most adopted encoder-decoder model.

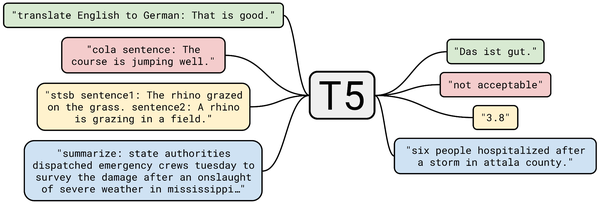


Figure 1-22. T5 model, figure taken from [the paper](https://arxiv.org/pdf/1910.10683)

**Vision Transformers**

Now you know how transformers work with text. Do they function the same way with images? Vision Transformers revolutionized the field by splitting images into tiny patches and applying the attention mechanism on those patches.

The patches are fixed-size grids of pixels, where the attention mechanism learns the meaning and importance across patches. The initial vision transformer (ViT) was similar to BERT, shown in [Figure 1-23](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_23_1749495171820498).

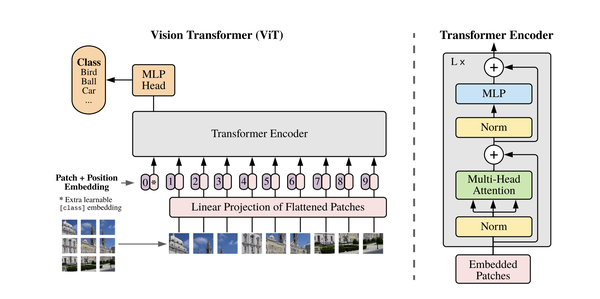


Figure 1-23. Vision Transformer Architecture, image taken from [the paper](https://arxiv.org/pdf/2010.11929)

ViT took images of 224x224 resolution and split them into 16x16 patches, which yield (224 / 16)**2** = 196 patches. These patches are then flattened into a 1-dimensional vector of length 768 (16**2**x 3-channels) and fed to a linear layer, converting these patches into a 768-dimensional embedding vector. The system adds a special token at the beginning of the image patches to signal the classification goal and includes positional information with each patch. It then sends these enhanced patches through the transformer network. Finally, a simple multi-layer perceptron takes the transformer’s output and determines the image classification.

ViT changed the paradigm in computer vision as evidenced by the large list of ViT variants, including dynamic-sized patches, modifying attention, adding convolutions on top of the attention layers or applying hierarchy through stages of ViT, learning patches (just like a tokenizer learn tokens), and different supervision techniques. We will go through some of the noteworthy architectures that advanced vision capabilities of multimodal models.

First, let’s try ViT with only a few lines of code! We will explain the code later, but to get started quickly, you need to install the Hugging Face transformers library.

# bash

pip install transformers

To initialize an image classification pipeline, think of it as a box that contains the model and everything else you need to do image classification. We will use the ViT model in [this link](https://huggingface.co/google/vit-base-patch16-224), trained on many classes.

# python

from transformers import pipeline

pipe = pipeline("image-classification", model="google/vit-base-patch16-224")

Now that we initialized the pipeline, let’s use the model to run inference on the image:

# python

pipe("https://huggingface.co/datasets/vlmbook/images/resolve/main/bee.jpg")

[{'label': 'bee', 'score': 0.943755030632019},

{'label': 'pot, flowerpot', 'score': 0.013617068529129028},

{'label': 'daisy', 'score': 0.008730421774089336},

{'label': 'ant, emmet, pismire', 'score': 0.003053140128031373},

{'label': 'fly', 'score': 0.0029011331498622894}]

And voila! The model correctly predicts that the image contains a bee and a flower.

**ViT variations**

**Masked Autoencoder (MAE)** is a standard ViT model with a twist: it divides images into patches, removes some of the image patches, and learns the image features through reconstructing images. This idea was later used in some of the initial image captioning models.

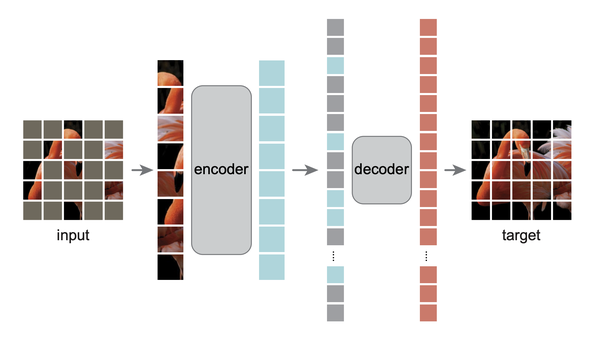


Figure 1-24. Description: Masked autoencoders divide images into patches, only take some of the patches, and try to learn through reconstructing the original image.

The community has built many models with different image handling approaches to improve performance. . We’ve seen BERT for text before, **BEiT**is a similar approach to BERT for images. BEiT learns to represent image patches as visual tokens, and the model is pre-trained with masked tokens to do so.

Another important modification on top of vanilla ViT is **hierarchical ViTs**. These models take in images in high resolution and split them into smaller patches like 4x4, then repeatedly downsample them to extract features, making extracted features bigger and bigger. This is in contrast to vanilla ViT, where the model learns details first and looks at the bigger picture in later stages, introducing locality bias. This trick is used for models that predict segmentation masks or bounding boxes from text description of objects of interest.

The last major modification on top of **ViT** is **self-supervision** techniques. Vision transformers require a lot of data for pre-training, so scaling over data as well as the architecture is necessary. Enter **Data Efficient Image Transformer (DeiT)**: it’s a model trained using a larger model to label data for the training of the small model (also called distillation). While the distillation technique used in DeiT is more complex, for now, you just need to know that this technique was used in early multimodal models. The MAE model we mentioned above also uses self-supervision: the training objective is to predict the masked image patches rather than class labels. MAE was used in early image captioning models as image backbone, and we still come across MAE or it’s variants in new models.

Now that we learned everything that led to the multimodality revolution, i.e. mixing text and images, let’s talk about the first successful open application of multimodality, CLIP.

**Connecting Images and Text: CLIP**

How do we associate the objects we see to language? In the early stage of our lives, the objects are always pointed out with saying what they are for us to learn. By asking a lot of questions, we learn concepts verbally, and we associate attributes with objects. Models also learn similarly: they are exposed to a lot of image and text pairs until they associate certain words with certain attributes and objects in images. Phrases are associated with images differentially, the phrase and text similarity match is scored like in [Figure 1-25](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_25_1749495171820531).

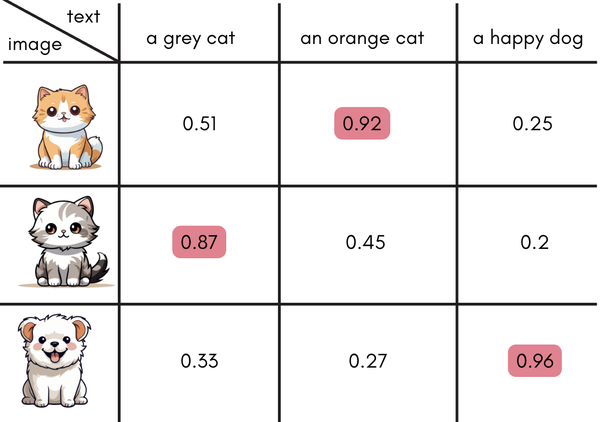


Figure 1-25. Similarity matrix

Notice how the phrases with the word cat inside are assigned a higher score to cat images meanwhile dog is isolated. This is the first and most accepted way of teaching models how to connect modalities.

Contrastive Language–Image Pre-training (CLIP) is the first successful attempt that connected the two modalities, understanding both image and text and translating one to another. The idea behind CLIP is as follows: an image encoder and a text encoder are pre-trained in contrastive manner. The training data consists of image and text pairs, and the training objective is to bring matching image-text pairs’ embeddings closer together while pushing dissimilar ones apart, *contrastively*.

In detail, each image and text in the training set (think of each combination as one example) are passed through their encoders, and these embeddings are then normalized and multiplied with dot product. After this, image-to-text prediction loss and text-to-image prediction loss are calculated separately, and final loss is the average of two.

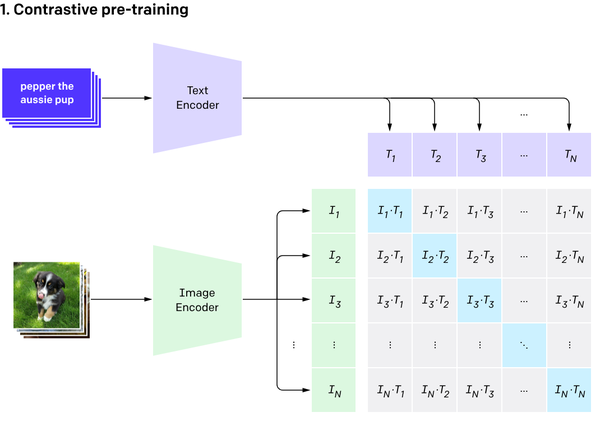


Figure 1-26. Contrastive Pre-training, Image taken from [the paper](https://arxiv.org/pdf/2103.00020). CLIP encodes image and separately, and tries to maximize similarity between the image-text pairs

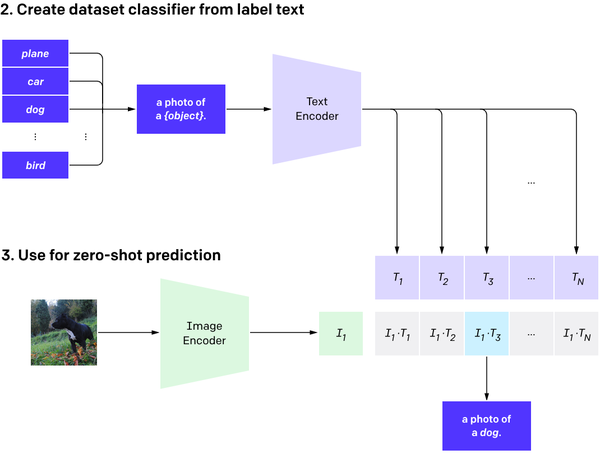


Figure 1-27. Inference with CLIP, image taken from[the paper](https://arxiv.org/pdf/2103.00020). During the inference, the model takes in an image, and multiple candidate labels, and returns high similarity score for text that describes image the best.

CLIP was the first ever widely adopted vision language model. Later on, the term evolved to models that can take image and text input and output text, like gpt-4o.

CLIP has found so many use in the industry, from image-generation pipelines to image-text search. CLIP can do zero-shot image classification. If this is your first encounter with the term zero-shot, it’s essentially used for tasks and models that can generalize for inputs and outputs beyond what it’s trained with. A model trained with yellow cats and black dogs can still recognize yellow dogs, it is a zero-shot image classification model.

In short, when given an image, and a bunch of open-ended labels to match the image, zero-shot classification models can assign similarity scores to match image to text (and vice versa).

CLIP-like models are still used today as the vision backbone for most successful Vision-Language models.

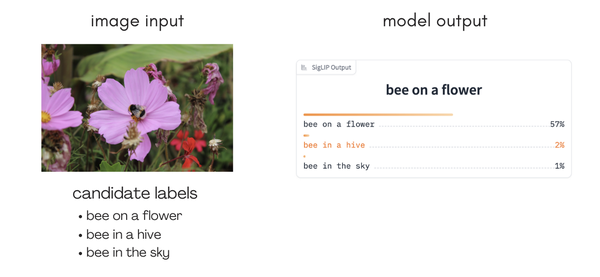


Figure 1-28. Zero-shot Image Classification with SigLIP model. On the right there is candidate labels and their similarity scores against the image assigned by the model

Let’s try using the CLIP model. To do so, we can initialize as follows, with the model in [this link](https://huggingface.co/openai/clip-vit-large-patch14).

# python

from transformers import pipeline

pipe = pipeline("zero-shot-image-classification", model="openai/clip-vit-large-patch14")

We will use the same image as we did in the ViT model example, but with three different candidate open-ended labels. The model will assess the most likely image description.

pipe(image="https://huggingface.co/datasets/vlmbook/images/resolve/main/bee.jpg", candidate\_labels=["bee on a flower", "bee in the hive", "bee in the sky"])

[{'score': 0.8133557438850403, 'label': 'bee on a flower'},

{'score': 0.1714152991771698, 'label': 'bee in the hive'},

{'score': 0.015228924341499805, 'label': 'bee in the sky'}]

As you can see, the model predicts the image correctly!

**Brief Introduction to Hugging Face Open-Source Ecosystem**

A typical machine learning workflow requires either training a model from scratch or taking a pre-trained model and fine-tuning it on a dataset with low-level PyTorch implementation of model architectures. For multimodal use cases, using a pre-trained model is preferred as training from scratch is too costly. But where do you find these pre-trained models, and how hard is it to fine-tune them?

The Hugging Face ecosystem makes it easy for the community to fine-tune and share models. The success of the transfer learning paradigm is mostly owed to the open-source libraries of Hugging Face.

Hugging Face ecosystem consists of two main components: a hub and a set of libraries that work together to simplify access to models, datasets, and demos with an easy-to-use code layer.

**Hugging Face Hub**

Hugging Face Hub is the largest platform for sharing machine learning models, datasets and demos. As of April 2025, it hosts over 1.6 million models and 360k datasets for many machine learning tasks as shown in [Figure 1-29](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_29_1749495171820594).

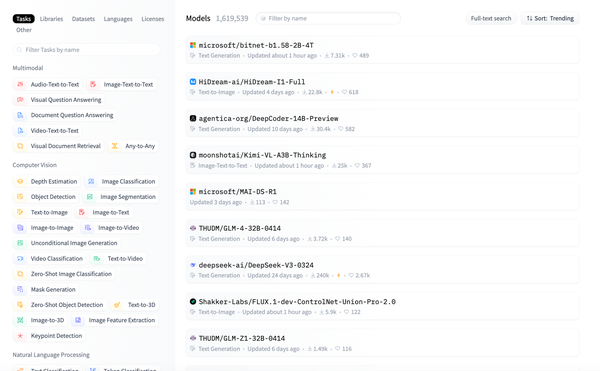


Figure 1-29. [Hugging Face Models Page](https://huggingface.co/models)

We will use Hugging Face model repositories heavily throughout this book so let’s take a look.

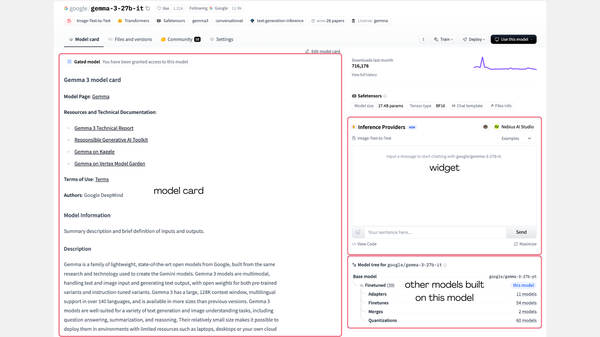


Figure 1-30. Hugging Face Model Repository

Each model repository ([Figure 1-30](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_30_1749495171820610)) consists of three tabs: model card, files and versions and community. Model card tab contains information related to the model, a widget to use the model, and a model tree. Model card contains metadata and documentation related to the model. Metadata contains information to filter for this model, such as task, the library, and license of the model. Documentation part of the model contains inputs and outputs, a snippet to get started with the model, training details, performance metrics, hardware benchmarks and more. We highly recommend checking model cards to get started with models in case of API changes over time.

Files and versions section contains model weight files and configuration files required to load the model. Community tab contains discussions and pull requests of the model repository. Discussions are very useful to ask questions about the model and get answers from the model owners and the community.

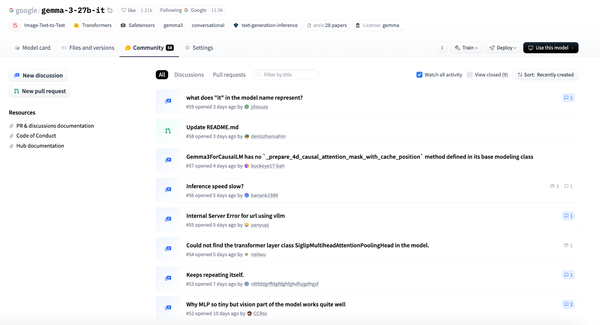


Figure 1-31. Hugging Face Model Repository Discussions Tab

Hugging Face Hub allows token-based access to modify and access repositories and many other operations. Throughout this book, you will read and write repositories. To do that, let’s create a token.

Go to [token settings](http://huggingface.co/settings/token), and on top right, select “Create new token”. Tokens come in three access types: fine-grained, read and write. Fine-grained allows creation of tokens with read or write access to specific repositories or repositories under a specific user and organizations. We really encourage using fine-grained tokens for security reasons. Read and write tokens allow read and write to all the repositories you have access to, including organizations where you have read and write access.

**Libraries**

Now that you have learned about the Hub, let’s take a look at the open-source libraries of Hugging Face.

**Transformers**

[Transformers](https://huggingface.co/docs/transformers/en/index) library provides easy to use abstractions for pre-trained models so you can easily fine-tune state-of-the-art models. It is now the central format in which all new pre-trained models are implemented: transformers library is integrated to different libraries both within and out of the Hugging Face ecosystem. A model implemented in transformers can be easily exported to different formats for different hardwares or inference solutions, so it is a must-to-learn for every machine learning practitioner working on cutting edge solutions.

Transformers provides three main abstractions to load, infer with and fine-tune models. We can summarize three of the abstractions that we will use below:

*AutoModel and AutoProcessor:*

In transformers, each model has its own class, e.g. CLIPModel is the class for CLIP. AutoModel classes automatically load a model weights for a specific task at hand. These model weights can be from a local folder or a model repository on Hugging Face Hub. For instance, we load vision language models with AutoModelForImageTextToText class, and zero-shot image classification models with AutoModelForZeroShotImageClassification class. This class loads the whole model, but we can also load parts of models. For example, we can use CLIPTextModel to load the text tower of CLIP.

We need to load the models along with the preprocessors to process inputs before passing the model. To load a processor locally or from the Hugging Face Hub model repository, we can use the AutoProcessor class. Processors contain both text processor (the tokenizer) and the image processor.

Here’s an example on loading CLIP and inferring with an image.

# python

from PIL import Image

import requests

from transformers import AutoProcessor, AutoModelForZeroShotImageClassification

model = AutoModelForZeroShotImageClassification.from\_pretrained("openai/clip-vit-large-patch14").to(“cuda”)

processor = AutoProcessor.from\_pretrained("openai/clip-vit-large-patch14")

url = "https://huggingface.co/datasets/vlmbook/images/resolve/main/bee.jpg"

image = Image.open(requests.get(url, stream=True).raw)

inputs = processor(text=["a bee on a flower", "a bee in a hive"], images=image, return\_tensors="pt", padding=True).to(“cuda”)

outputs = model(\*\*inputs)

probs = outputs.logits\_per\_image.softmax(dim=1)

print(probs)

# tensor([[0.5359, 0.4641]], grad\_fn=<SoftmaxBackward0>)

# probabilities for each class respectively

*pipeline:*

we have already used the pipeline abstraction above. Think of [the pipeline](https://huggingface.co/docs/transformers/en/main_classes/pipelines) as a box that contains processor and model, with preprocessing of inputs, inference and post-processing the model outputs are abstracted.

# python

from transformers import pipeline

image\_classifier = pipeline(model="openai/clip-vit-large-patch14", task="zero-shot-image-classification")

image\_classifier(image = "https://huggingface.co/datasets/vlmbook/images/resolve/main/bee.jpg", candidate\_labels = ["a bee on a flower", "a bee in a hive"])

# [{'score': 0.9376115202903748, 'label': 'a bee on a flower'}, {'score': 0.06238847225904465, 'label': 'a bee in a hive'}]



Figure 1-32. Transformers Zero-shot Image Classification Pipeline

*Trainer:*

Model fine-tuning and training loops follow a common workflow. [Trainer](https://huggingface.co/docs/transformers/en/main_classes/trainer) is a plug-and-play abstraction to handle passing in datasets, handling loss calculation as well as advanced tricks to get the best out of small hardware such as gradient accumulation and gradient checkpointing. Trainer is also supported by Accelerate library for training on multiple hardware. We will see how to use Trainer more in detail in upcoming chapters.

**Accelerate**

Fine-tuning large models requires different tricks such as mixing different weight precision, data parallel training and using multiple hardware. Accelerate is a library that enables the same PyTorch code to be run across any distributed configuration by adding just four lines of code. It handles all of these under the hood through Trainer, easily enabling using advanced tricks to train large models.

**Datasets**

Working with large datasets across different modalities is challenging. These datasets often require heavy memory larger than RAM and take a lot of time to process. [Datasets](https://huggingface.co/docs/datasets/en/index) library helps reduce the memory constraints and enable fast processing of datasets. It also comes with different features such as streaming datasets from Hugging Face Hub, which enables easy discovery of datasets. The library comes with support for all modalities, and is tailored for different needs.

Let’s take a look at how a dataset is loaded. First, make sure to have datasets installed.

pip install datasets

We will use load\_dataset to load a small visual question answering dataset.

# python

from datasets import load\_dataset

ds = load\_dataset("merve/vqav2-small")

print(dataset)

ds here is an object of type DatasetDict and this dictionary contains different split dataset contains. Each split is a Dataset object with features and number rows.

DatasetDict({

validation: Dataset({

features: ['multiple\_choice\_answer', 'question', 'image'],

num\_rows: 21435 }) })

To actually bring the dataset itself, simply access the split by calling ds["validation"]. To get the first example, call ds["validation"][0].

{'multiple\_choice\_answer': 'carnival ride',

'question': 'Where are the kids riding?',

'image': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=640x424>}

We can use map to preprocess the dataset like below. For instance, below example lowercases all the questions.

# python

def preprocess(example):

example["question"] = example["question"].lower()

return example

ds["validation"] = ds["validation"].map(preprocess)

print(ds["validation"][0]["question"])

# where are the kids riding?

You will see how to enable batch processing and using datasets with torch DataLoader for training later in the Chapter 5.2 post-training.

**Coding Example: Searching for Images from Text**

Now that you have learnt about the fundamentals in the transformers ecosystem, let’s build your first multimodal application. You will build an application that can search for artworks based on a text query. To do so, you will use the SigLIP2 model. SigLIP2 is the state-of-the-art CLIP-like model released by Google. Like CLIP, it consists of an image and text encoder that we call a “tower”.

Doing search with CLIP-like models is straightforward. In our case, we will search for images from text. It consists of two steps:

1. Index all images using the image tower of [SigLIP](https://huggingface.co/docs/transformers/en/model_doc/siglip" \t "_blank), get the embeddings and store them side by side with images as shown in [Figure 1-33](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_33_1749495171820653).
2. During the inference, you will first get the embedding of the text query through as shown in [Figure 1-34](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_34_1749495171820666) and then we would find the closest image embeddings and retrieve those

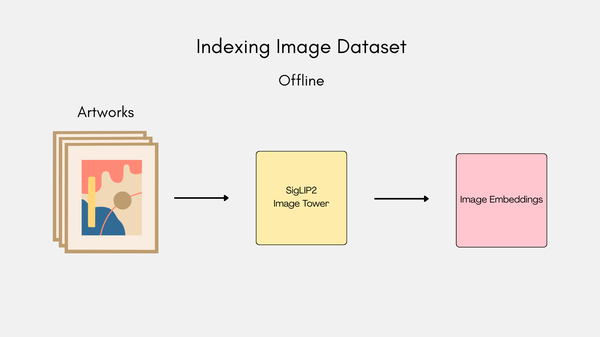


Figure 1-33. Offline image indexing

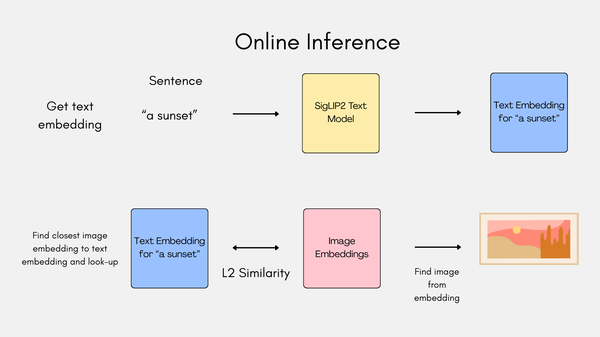


Figure 1-34. Online Inference

You will use [FAISS](https://faiss.ai/index.html) to store and search the embeddings. If you are on a GPU, make sure to check the CUDA version with the command `nvidia-smi`, and install the appropriate faiss-gpu version. For CUDA 12.x it is faiss-gpu-cu12.

You also need the sentencepiece library, so let’s make sure you have everything we need.

pip install datasets faiss-gpu-cu12 transformers sentencepiece

You will use [siglip-base-patch16-224](https://huggingface.co/google/siglip-base-patch16-224), this is the SigLIP with patch size 16 and resolution 224, 224.

# python

import torch

from transformers import AutoProcessor, AutoModel, AutoTokenizer

device = torch.device('cuda' if torch.cuda.is\_available() else "cpu")

model\_id = "google/siglip-base-patch16-224"

model = AutoModel.from\_pretrained(model\_id, device\_map="auto").eval()

processor = AutoProcessor.from\_pretrained(model\_id)

tokenizer = AutoTokenizer.from\_pretrained(model\_id)

This is a toy example, so you will be using a small portion of the dataset. You will stream 100 examples from the dataset. To do so, you need to authenticate yourself first. You can use either login of huggingface\_hub library, or in command line interface, use huggingface-cli login.

# python

from huggingface\_hub import login

login()

The tokens are stored under the environmental variable HF\_TOKEN.

#python

import os

API\_TOKEN = os.environ.get('HF\_TOKEN')

Let’s stream 100 examples from the dataset.

# python

import requests

headers = {"Authorization": f"Bearer {API\_TOKEN}"}

API\_URL = "https://datasets-server.huggingface.co/rows?dataset=vlmbook/wikiart&config=default&split=train&offset=1&length=100"

def query():

response = requests.get(API\_URL, headers=headers)

return response.json()

data = query()

# take a look at first example

data["rows"][0]

# output

{'row\_idx': 1,

'row': {'image': {'src': 'https://datasets-server.huggingface.co/assets/vlmbook/wikiart/--/469f082b983c178a3457953695a9e90aa11da171/--/default/train/1/image/image.jpg',

'height': 1659,

'width': 1382},

'artist': 20,

'genre': 7,

'style': 4},

'truncated\_cells': []}

Let’s see how the image looks like. The images are large, you need to downscale them to see properly in the notebook.

# python

from PIL import Image

url = data["rows"][0]["row"]["image"]["src"]

image = Image.open(requests.get(url, stream=True).raw)

width = 300

ratio = (width / float(image.size[0]))

height = int((float(image.size[1]) \* float(ratio)))

img = image.resize((width, height), Image.Resampling.LANCZOS)

display(img)



Figure 1-35. Example Image from Dataset

Now let’s get to indexing the dataset. Below are helper functions to index the dataset. add\_vector adds a vector to FAISS and embed\_siglip processed the image and gets image embeddings from SigLIP model, so we will first call embed\_siglip and store the embedding output with add\_vector.

# python

import numpy as np

import faiss

def add\_vector(embedding, index):

vector = embedding.detach().cpu().numpy()

vector = np.float32(vector)

faiss.normalize\_L2(vector)

index.add(vector)

def embed\_siglip(image):

with torch.no\_grad():

inputs = processor(images=image, return\_tensors="pt").to(device)

image\_features = model.get\_image\_features(\*\*inputs)

return image\_features

Now, let’s initialize the FAISS index, and start indexing the dataset. SigLIP model outputs 1152-dimensional vectors, and you need to pass it to IndexFlatL2 to create your index.

# python

from torchvision import transforms

index = faiss.IndexFlatL2(1152)

for elem in data["rows"]:

url = elem["row"]["image"]["src"]

image = Image.open(requests.get(url, stream=True).raw)

clip\_features = embed\_siglip(image)

add\_vector(clip\_features, index)

You can save this index so you can do the search with it later on. To save, you can call write\_index and to load, you can use read\_index.

# python

faiss.write\_index(index,"./siglip\_70k.index")

index = faiss.read\_index("./siglip\_70k.index")

You can now search with the text query. Let’s get the artworks with a woman in them.

To do this, you need to process the text input and get the text embeddings, then you need to pass the text through the tokenizer, which can be accessed with processor.tokenizer.

# python

prompt="a woman"

text\_token = processor.tokenizer([prompt], return\_tensors="pt").to(device)

text\_features = model.get\_text\_features(\*\*text\_token)

text\_features = text\_features.detach().cpu().numpy()

text\_features = np.float32(text\_features)

faiss.normalize\_L2(text\_features)

You will get the two most matching artworks. index.search returns the distance between your query and closest images, along with the indices of them, which can be mapped to the dataset to get the actual images.

# python

distances, indices = index.search(text\_features, 2)

print(indices)

# array([[50, 18]])

Let’s see the retrieved images, shown below in Figures [1-36](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_36_1749495171820693) and [1-37](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_37_1749495171820707)

# python

for elem in indices[0]:

url = data["rows"][elem]["row"]["image"]["src"]

image = Image.open(requests.get(url, stream=True).raw)

# downscale

width = 300

ratio = (width / float(image.size[0]))

height = int((float(image.size[1]) \* float(ratio)))

img = image.resize((width, height), Image.Resampling.LANCZOS)

display(img)



Figure 1-36. First Inference Result

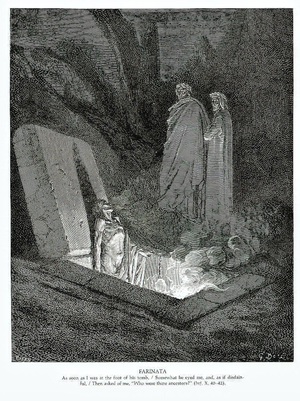


Figure 1-37. Second Inference Result

And voila! You have built your first multimodal application.

If you’re interested, there’s a deployed version of this application where you can draw images to search in WikiArt database [here](https://huggingface.co/spaces/merve/draw_to_search_art) (see [Figure 1-38](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch01.html#ch01_er_figure_38_1749495171820727)).

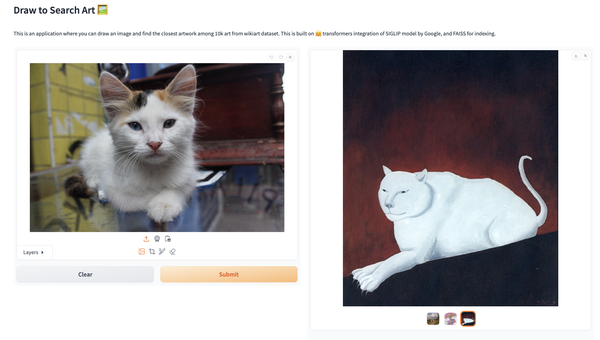


Figure 1-38. Image Search Application Interface

Conclusion

We have taken you through a journey in time to see all the things that happened until today where we have powerful multimodal models under our fingertips, and built our first multimodal application. We are just getting started: we will train and fine-tune our own models to accomplish a huge variety of tasks. Throughout this book, we will train models, fine-tune and align them for helpfulness, learn how to shrink and deploy them, and even see special niche use cases like document AI or agentic vision language use cases like web automation.

In the next chapter, we’ll see what today’s vision language models can accomplish, such as visual question answering or zero-shot object detection, and how to find models on Hugging Face Hub and infer with them using transformers. Let’s go!

--

Chapter 2. Vision Language Model Applications

**A Note for Early Release Readers**

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *shunter@oreilly.com*.

Vision Language Models (VLMs) are models that can interpret both image and text. VLMs, typically trained on massive vision, vision-language, and language datasets, can accomplish a wide variety of vision tasks like identifying objects, actions, and scenes, as well as understand text for multimodal tasks.

In this chapter, take a look at multimodal tasks and models, how they are employed practically, and how to evaluate them.

Image Captioning

Image captioning is the task of describing the visual content of an image in natural language​, see [Figure 2-1](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_1_1749506846817479) to quickly get it. Image captioning sits at the intersection of computer vision and natural language processing. It requires a model to *understand an image*(identify objects, attributes, and their relationships) and *generate a coherent sentence* describing the image. This task is a *fundamental problem* in artificial intelligence that *connects vision and language*​.

A diagram of a picture of a street

AI-generated content may be incorrect.

Figure 2-1. Image captioning input, output, and rough model architecture

**Background**

To solve this problem, early methods adopted a **visual encoder** (typically a convolutional neural network, CNN) that transforms the image into a feature representation, and a **text decoder** (usually a recurrent neural network, RNN) to generate a sentence from that representation (see [Figure 2-2](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_2_1749506846817532)). This end-to-end approach learned to *“translate”* an image to a caption, treating the image as the source language and English as the target language​.

One of the first successful models was Neural Image Caption (NIC) from Google (see [Figure 2-2](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_2_1749506846817532)). The system takes an image, processes it with a pre-trained CNN to extract visual features, and then feeds these features into an LSTM. This LSTM generates the caption one word at a time. How does it learn this? During training, the LSTM learns to predict the next word of actual human-written captions, creating a connection between visual content and language to generate appropriate descriptions for new images.

A diagram of two people

AI-generated content may be incorrect.

Figure 2-2. Image captioning, from ‘Show and Tell: A neural Image caption Generator’

Captioning was quickly subsumed into general Vision-Language Models (VLMs) with the introduction of **BLIP-2**, a compact yet powerful model that efficiently bridges visual encoders with pretrained large language models (LLMs). BLIP-2 leverages advanced techniques such as multimodal alignment and instruction-based training, enabling it to excel at image captioning tasks while maintaining modest computational requirements as shown in [Figure 2-3](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_3_1749506846817558).

A diagram of a language model

AI-generated content may be incorrect.

Figure 2-3. bridging image encoders with large language models, from BLIP-2 paper.

Recent models adopt CLIP-like powerful visual encoders with pre-trained large language models (LLMs) to improve caption generation. Contrastive learning objectives and extensive pretraining on massive, diverse image-text datasets achieve state-of-the-art performance in image captioning. Many generalistic vision language models can caption images in detail, as well as perform other vision-language tasks.

See [Figure 2-4](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_4_1749506846817578) to get a grasp of the architecture of modern vision language models.

A diagram of a computer game

AI-generated content may be incorrect.

Figure 2-4. SmolVLM architecture combining image pre-processing, vision encoders, tokenizer and a LLM.

**Image Captioning Evaluation Metrics.**

How can we evaluate a good captioning model? Most text generation models work over subword (also known as n-grams) overlaps. Let’s look at an example where our reference text is ‘This image shows a sunny view’ and the output of the model is ‘This image depicts a sunny day’, as shown in [Figure 2-5](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_5_1749506846817595). 1-gram refers to single word overlaps (‘This’, ‘image’..), 2-gram refers to overlap of phrases within 2-word windows (‘This image’, ‘image shows’..). A simple evaluation includes counting 1-grams or 2-grams in a sentence, and dividing the number of 1-gram or 2-gram overlaps by total number of 1-grams or 2-grams. This is called “BLEU Score”.

A screenshot of a math test

AI-generated content may be incorrect.

Figure 2-5. Overlap Counting

**ROUGE (Recall-Oriented Understudy for Gisting Evaluation)**

Designed for document summarization, ROUGE score compares the generated text against the ground truth caption, and the overlap of tokens is counted. The N at the end refers to the overlap of N subsequent tokens, for example, ROUGE-2 is the overlap of two subsequent tokens, calculated like [Figure 2-6](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_6_1749506846817611).

A black text on a white background

AI-generated content may be incorrect.

Figure 2-6. Rouge Score

Let’s return to the same example from above. You can see the same overlapping parts and how we calculate the ROUGE score in [Figure 2-7](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_7_1749506846817626).

A math equation with numbers and symbols

AI-generated content may be incorrect.

Figure 2-7. ROUGE Calculation

Visual Question Answering (VQA)

Visual Question Answering (VQA) is the task of answering questions about an image. VQA demands focused understanding and accurate multimodal reasoning. It can be complex: the model has to combine and interpret image and text inputs. Imagine questions that involve counting, localizing and comparing.

In other words, the model needs to understand how objects are arranged, how they might interact, and use common sense about everyday life to output accurate responses.

Let’s look at the evolution of VQA models in the following sections.

**Feature-Based approaches**

The early VQA models kept things pretty simple: they grabbed image features using CNNs and encoded questions with RNNs, then combined them with basic classifiers to predict answers. These models worked okay for simple questions like “What color is the car?” but hit a wall when faced with anything requiring deeper reasoning.

**Attention Mechanism-based approaches**

Later models employed the attention mechanism to dynamically focus on regions of interest in an image based on the question itself. The co-attention mechanism, which we will see in later sections, further refines this approach by attending to both image features and linguistic inputs.

**Transformer-based models**

Today’s cutting-edge VQA models use transformers trained on massive image-text datasets. These models are much more impressive - they generalize well to new scenarios and can handle complex questions by bringing in outside knowledge that isn’t even in the image. They’ve learned from seeing millions of image-text pairs, developing a kind of visual common sense that earlier models simply didn’t have.

Despite these advancements, models continue to struggle with complex, multi-step reasoning, abstract concepts, and “what if” cases. Often, models exploit dataset biases instead of truly understanding, and transparency in their reasoning remains limited. Multi-step conversations are even harder to solve, as models need to remember the context. With ongoing research, models keep making progress towards human-like visual reasoning.

During the training of a new model, the evaluation on VQA is usually done through two approaches:

*Multi-choice Q&A*

The model must choose the correct answer from a multiple-choice list.

*Free-style answer*

The model provides a free-style answer, and another entity (a person or another model) evaluates the correctness of the answer.

Visual Reasoning

Visual reasoning represents a key challenge in multimodal AI. Visual reasoning demands systems not to only recognize visual elements, but also interpret complex relationships, infer properties, and reason logically from visual information. Unlike simpler tasks of visual understanding, visual reasoning involves understanding how objects relate, move, or interact in a given scene. Accomplishing this task is essential for bridging perception and cognition to approach human-like intelligence.

Visual Language Retrieval

Vision Language Retrieval (VLR) is the task of creating meaningful connections between images and text. VLR includes tasks that retrieve relevant images based on open-ended text input and vice versa. Unlike generative captioning or VQA, retrieval tasks identify the most relevant items from existing databases to manage and search extensive image-text repositories to keep the meaning across two modalities.

**How VLR works**

Vision Language Retrieval models have to work on a joint image-text vector space. In this space, images and texts that share attributes are closer to each other. The distance between elements in that space is measured with cosine similarity or Euclidean distance.

Retrieval is formulated as a ranking problem. Since there are text and images, we’re dealing with three situations:

*Text to image retrieval:*

Images are ranked by relevance to a textual query

*Image to text retrieval:*

Textual descriptions are ranked based on their relevance to an image query

*Image to image retrieval:*

Images are ranked by relevance to an image query

And actually… you can combine image+text and do retrieval, this is particularly useful when you have an example image and want a slight variation of it: ‘same dress with red details’.

Models like CLIP (introduced in Chapter 1) are heavily used for VLR. These models are trained to maximize similarity between matching image-text pairs and minimize similarity with irrelevant pairs.

Search engines (present in e-commerce, video platforms, social networks), accessibility, content moderation (especially in social networks) are some of the applications of VLRs.

Evaluation methods for Vision-Language Retrieval are expanding beyond basic accuracy metrics. Increasingly, researchers are looking at factors like complex query understanding (avoid relying on superficial correlations), result diversity (whether the retrieved items show variety), fairness (making sure results don’t unintentionally favor certain groups), and computational efficiency (how quickly and resource-efficiently systems operate).

Document Understanding

Document understanding is one of the most practical and commercially important applications of Vision Language Models (VLMs). Unlike typical photos or images, documents pack information into text, tables, charts, diagrams, and other visual elements organized in specific layouts.

Traditionally, document understanding relied mainly on Optical Character Recognition (OCR) combined with handcrafted rules to interpret layouts and content. While these OCR-based methods worked fine for simple, standardized documents, they often struggled with complex layouts, poor-quality scans, or complex visuals. Thankfully, the deep learning revolution has dramatically improved this, offering robust, adaptable solutions that handle diverse document types and extract richer meanings as shown in [Figure 2-8](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_8_1749506846817642).

A diagram of a diagram

AI-generated content may be incorrect.

Figure 2-8. InfoVQA Example

Modern document understanding systems operate in hierarchical layers. They begin with recognizing text, detecting visual elements, and parsing document layouts. Then, they analyze structure by identifying components like paragraphs, tables and figures while mapping their spatial and hierarchical relationships. Finally, these systems integrate visual and textual information to extract deeper meanings, enabling complex tasks such as question answering, summarization, and knowledge extraction.

Let’s dig into two key aspects: document retrieval and information extraction.

**Document Retrieval**

**Document retrieval** is a crucial component of document understanding that helps users find relevant documents in large collections. Traditional systems relied on keyword matching using techniques like TF-IDF (Term Frequency-Inverse Document Frequency), which works by giving higher importance to words that appear frequently in a document but rarely across the entire collection. This helps identify distinctive terms in each document.

Modern retrieval systems have evolved to use dense retrieval methods that capture deeper meaning. These approaches use neural networks to convert both documents and search queries into numeric representations (embeddings) within the same semantic space. By incorporating text, visual elements, and layout information, these embeddings better understand document meaning beyond just keywords.

Multimodal retrieval techniques are particularly powerful because they can distinguish between documents that have similar text but different visual elements or layouts, making search results more accurate. Some specialized systems are even tailored for specific industries like legal or healthcare, incorporating domain-specific terminology and document structures to improve results.

**Information extraction**

**Information extraction** involves pulling structured data from complex documents with diverse layouts. This process goes beyond simply extracting text—it requires understanding how information relates spatially and visually within documents. Traditional approaches using templates or rigid rules worked for standard documents but struggled with variations.

Modern deep learning methods now excel at extracting key-value pairs, recognizing named entities, and analyzing document layouts. Models like Mask R-CNN have significantly improved our ability to segment different regions in documents. Table extraction, which remains particularly challenging, typically follows a step-by-step process: identifying the table location, detecting its rows and columns, and then extracting the content from each cell. Graph neural networks have proven especially effective for handling these complex relationships across different document formats.

**Looking ahead**

Document understanding is challenging: endless document layouts, poor image quality, and complex reasoning requirements (like multi-step calculations) remain difficult problems. Additionally, most current models require substantial labeled data, limiting their effectiveness in specialized domains or low-resource environments.

Recent breakthroughs come primarily from document-specific vision-language models. Systems like LayoutLM, DocFormer, and UDOP smartly combine layout and text information during pre-training, leading to impressive results on standard benchmarks such as FUNSD (for form understanding) and CORD (for receipt analysis). Currently, there are generic vision language models like GOT-OCR 2 with swiss army knife-like capabilities for documents. These models can tackle a variety of tasks like document OCR, chart captioning and more, rendering complex documents to markdown.

There are many subtasks solved within document understanding. For example, ChartQA involves answering tricky questions about charts and graphs, while InfoVQA takes visual question answering into infographics. Another one is DocVQA, which focuses on answering questions about documents. We will go through them in Document AI chapter in-depth.

Video Understanding

Video is just a collection of consecutive images; what would make it particularly different from what we have seen until now? Good question!

Video understanding in Vision-Language Models (VLMs) requires interpreting dynamic visual content alongside text descriptions. Unlike static images, videos contain evolving scenes with motion, actions, and temporal interactions that change over time. Processing videos effectively demands sophisticated methods that can track sequences of events while maintaining context throughout changing scenes as shown in [Figure 2-9](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_9_1749506846817658).

A screenshot of a video question

AI-generated content may be incorrect.

Figure 2-9. Video Understanding

Video understanding faces several distinct challenges compared to image analysis:

* The *temporal dimension* requires models to continuously track objects, recognize ongoing actions, and maintain coherent understanding across multiple frames.
* Typically, videos have *lower information* density than images, with significant events often scattered among many less informative frames.
* *Computational resources*: processing video content demands substantially more resources due to longer durations and increased complexity.
* *Datasets* for training and evaluating: obtaining video content and creating comprehensive labeled video-text datasets requires significant effort and resources, which means the variety and availability of data are limited.

At this point of the chapter, you probably guessed it: a major breakthrough came with Transformer-based architectures, which use self-attention mechanisms to capture extensive temporal relationships in video content. When combined with large-scale pretraining on vast datasets of video-text pairs, these Transformer models dramatically improved video understanding capabilities.

Let’s look at a few common video understanding tasks:

*Video-text retrieval*

The process of finding relevant video content based on text prompts. Modern video-text retrieval models are CLIP-like models with encoders for both video and text, trained on narrated video clips.

*Video captioning*

To generate descriptions for video content. Recent captioning models are based on

*Temporal grounding*

Identifies specific timestamps within videos for given text prompts. This allows users to pinpoint exact moments in videos using text descriptions rather than manual searching.

As highlighted in our introduction, the scarcity of comprehensive datasets continues to challenge video understanding models. Researchers are actively developing more data-efficient approaches through self-supervised, weakly supervised, and few-shot learning techniques to enable robust generalization across new domains and real-world scenarios. We will talk about these and evaluation techniques in Chapter 8 Video Language Models.

Instance Localization with VLMs

Vision-Language Models (VLMs) show state-of-the-art results in instance localization from text prompts. Unlike traditional models that are trained on a fixed set of classes, vision language models can detect arbitrary objects through text descriptions or visual prompts (boxes around the object or point inside the object in the image). We will now go through instance localization applications, including zero-shot object detection and image-guided detection, and show how to infer with the existing set of models.

**Zero-shot Object Detection**

Zero-shot object detection models can detect objects from open-ended text prompts as shown in [Figure 2-10](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_10_1749506846817672). OWLv2 and GroundingDINO are the two advanced open-source models for this task. OWLv2, released by Google, uses a vision encoder combined with a text encoder similar to CLIP, to detect bounding boxes through text prompts. The vision encoder is followed by a simple linear layer to detect the bounding boxes with their labels. GroundingDINO is very similar to OWLv2, it has an image encoder and a text encoder, with feature enhancer modules after them.

A diagram of a flower

AI-generated content may be incorrect.

Figure 2-10. Zero shot object detection

Vision language models in zero-shot object detection are practically less cost-efficient to run in production than their pure computer vision peers. Nevertheless, VLMs are great to create training datasets for vision models: they are often used in automatic bounding box annotation for training of smaller object detectors.

Another challenge with working with zero-shot object detectors is filtering for their results. These models take a confidence threshold input used to filter out incorrect bounding boxes. There is no free lunch when picking confidence thresholds, and they depend on how fine-grained the task is.

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure 2-11. OWLv2

To use OWLv2 through UI, visit [this link](https://huggingface.co/spaces/merve/owlv2). Now, we will see how to infer. We will first load the model.

# python

from transformers import AutoProcessor, AutoModelForZeroShotObjectDetection

model\_name = "google/owlv2-base-patch16-ensemble"

processor = AutoProcessor.from\_pretrained(model\_name)

model = AutoModelForZeroShotObjectDetection.from\_pretrained(model\_name).to(“cuda”)

Now let’s load the image and write our text prompts. We will pass them to processor.

# python

image\_url = "https://huggingface.co/datasets/vlmbook/images/resolve/main/bee.jpg"

image = Image.open(requests.get(image\_url, stream=True).raw)

text\_queries = [["bee on the flower", “bee in the hive”]]

inputs = processor(images=image, text=text\_queries, return\_tensors="pt")

We can now infer and post-process.

# python

outputs = model(\*\*inputs)

results = processor.post\_process\_grounded\_object\_detection(outputs, inputs.input\_ids, threshold=0.3, target\_sizes=[image.size[::-1]])

for box, score, label in zip(results[0]["boxes"], results[0]["scores"], results[0]["labels"]):

print(f"Detected '{label}' with confidence {score:.2f} at {box.tolist()}")

**Image-guided Detection**

Image-guided detection is used to detect objects in images by showing an image reference. The model compares the reference image to different parts of the target image and finds potential matches based on visual similarity. Pragmatically, this is useful when some objects are hard to describe in words or when you want to find Wally with his red stripe shirt.

A diagram of a flower

AI-generated content may be incorrect.

Figure 2-12. Image Guided Detection

Looking at how a state of the art model like OWLv2 achieves image-guided detection, we learn that it embeds a reference image into the shared feature space of the model, which then calculates similarity between this reference embedding and regions in the target image to identify matching objects without needing text prompts.

The more similar the reference image and the instance of the object in the target image, the higher chances of matching success. While this was already a problem with manually engineered features (remember SIFT in Chapter 1), overall the VLM based solution provides a more robust one-shot detection capability.

**Object Counting**

If we can detect objects through descriptions or visual references, can we count them accurately?

Intuitively, counting should be as simple as tallying the bounding boxes of detected objects. This approach has precedent in traditional computer vision methods that grouped SIFT keypoints.

The versatility of modern models like OWLv2 enables detection-based counting of specific objects—“red cars” or particular animal species—through simple descriptive prompts, eliminating the need for specially trained detectors. This represents a substantial advancement in flexible, zero-shot object counting capabilities.

**Image segmentation**

While detection models identify objects with bounding boxes, segmentation takes precision to the next level by delineating exact object boundaries.

Meta’s Segment Anything Model changed the game by working with simple inputs like just a point within the object and a bounding box surrounding the object. With visual prompts and training over over a billion masks, it’s the state-of-the-art segmentation model. The second iteration of SAM, SAM2 allowed object tracking across frames by keeping a separate memory of the same object across time.

Combining SAM with zero-shot object detectors like OWLv2 and GroundingDINO yields performant zero-shot image segmentation pipelines. Let’s learn how you can build one.

A cat lying on a blanket

AI-generated content may be incorrect.

Figure 2-13. Zero-shot Image Segmentation with OWLSAM Pipeline

OWLSAM is a pipeline combining OWL and SAM models for zero-shot image segmentation, as seen in [Figure 2-13](https://learning.oreilly.com/library/view/vision-language-models/9798341624030/ch02.html#ch02_er_figure_13_1749506846817717). First a detection model like OWLv2 finds objects based on a text prompt and then SAM takes the bounding boxes and segments the objects in them. Feel free to try it [here](https://huggingface.co/spaces/merve/OWLSAM).

A screenshot of a computer

AI-generated content may be incorrect.