Chapter 16. Vision and Multimodal Transformers

In the previous chapter, we implemented a transformer from scratch and turned it into a translation system, then we explored encoder-only models for NLU, decoder-only models for NLG, and we even built a little chatbot—that was quite a journey! Yet, there's still a lot more to say about transformers. In particular, we have only dealt with text so far, but transformers actually turned out to be exceptionally good at processing all sorts of inputs. In this chapter we will cover *vision transformers* (ViTs), capable of processing images, followed by *multimodal transformers*, capable of handling multiple modalities, including text, images, audio, videos, robot sensors and actuators, and really any kind of data.

In the first part of this chapter, we will discuss some of the most influential pure-vision transformers:

DETR

An early encoder-decoder transformer for object detection.

The original ViT

This landmark encoder-only transformer treats image patches like word tokens and reaches the state of the art if trained on a large dataset.

DeiT

A more data-efficient ViT trained at scale using distillation.

PVT

A hierarchical model that can produce multiscale feature maps for semantic segmentation and other dense prediction tasks.

Swin Transformer

A much faster hierarchical model.

DINO

This introduced a novel self-supervised technique for visual representation learning.

In the second part of this chapter, we will dive into multimodal transformers:

VideoBERT

A BERT model trained to process both text and video tokens.

ViLBERT

A dual-encoder model for image plus text, which introduced co-attention (i.e., two-way cross-attention).

CLIP

This is another image plus text dual-encoder model trained using contrastive pretraining.

DALL·E

A model capable of generating images from text prompts.

Perceiver

This efficiently compresses any high-resolution modality into a short sequence using a cross-attention trick.

Perceiver IO

Adds a flexible output mechanism to the Perceiver, using a similar cross-attention trick.

Flamingo

Rather than starting from scratch, it reuses two large pretrained models—one for vision and one for language (both frozen)—and connects them using a Perceiver-style adapter named a Resampler. This architecture enables open-ended visual dialogue.

BLIP-2

This is another open-ended visual dialogue model that reuses two large pretrained models, connects them using a lightweight querying transformer (Q-Former), and uses a powerful two-stage training approach with multiple training objectives.

So turn on the lights, transformers are about to open their eyes.

Vision Transformers

Vision transformers didn't pop out of a vacuum: before they were invented, there were RNNs with visual attention, and hybrid CNN-Transformer models. So let's take a brief look at these ViT ancestors before we dive into some of the most influential ViTs.

RNNs with Visual Attention

One of the first applications of attention mechanisms beyond NLP was in generating image captions using visual attention.[1] Here a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one token at a time.

The decoder uses an attention layer at each decoding step to focus on just the right part of the image. For example, in Figure 16-1, the model generated the caption "A woman is throwing a

Frisbee in a park", and you can see what part of the input image the decoder focused its attention on when it was about to output the word "Frisbee": clearly, most of its attention was focused on the frisbee.

Figure 16-1. Visual attention: an input image (left) and the model's focus before producing the word "Frisbee" (right)[2]

EXPLAINABILITY

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a mistake; for example, if an image of a dog walking in the snow is labeled as "a wolf walking in the snow", then you can go back and check what the model focused on when it output the word "wolf". You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps he model learned to distinguish dogs from wolves by checking whether there's a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great 2016 paper[3] by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier's prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement—think of a system deciding whether it should grant you a loan.

Once transformers were invented, they were quickly applied to visual tasks, generally by replacing RNNs in existing architectures (e.g., for image captioning). However, the bulk of the visual work was still performed by a CNN, so although they were transformers used for visual tasks, we usually don't consider them as ViTs. The *detection transformer* (DETR) is a good example of this.

DETR: a CNN-Transformer Hybrid for Object Detection

In May 2020, a team of Facebook researchers proposed a hybrid CNN–transformer architecture for object detection, named *detection transformer* (DETR, see Figure 16-2).[4] The CNN first pro-

for object detection, named *detection transformer* (DETR, see Figure 16-2). The CNN first processes the input images and outputs a set of feature maps, then these feature maps are turned into a sequence of visual tokens that are fed to an encoder-decoder transformer, and finally the transformer outputs a sequence of bounding box predictions.

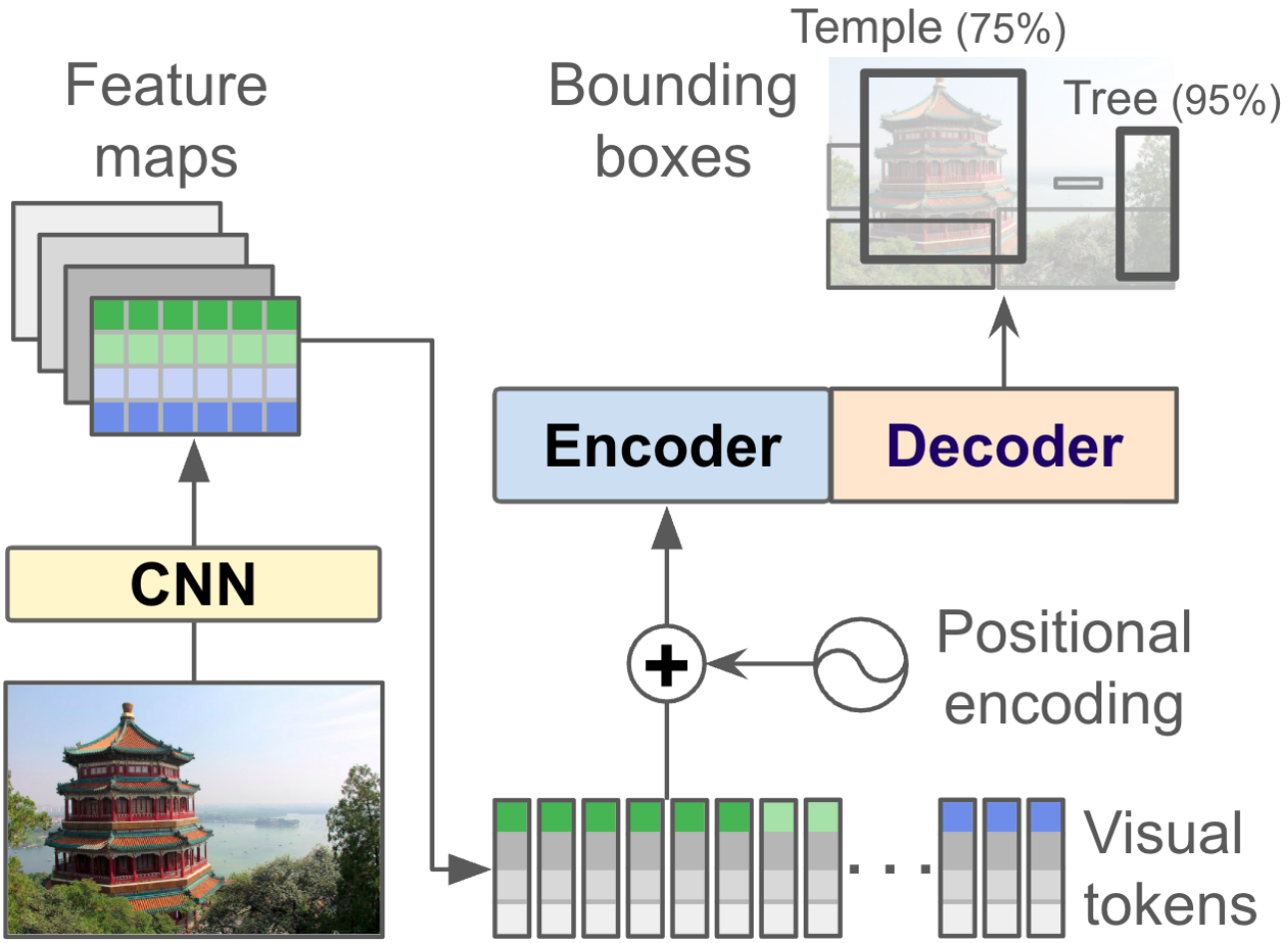


Figure 16-2. The detection transformer (DETR) for object detection

At one point, someone was bound to wonder whether we could get rid of the CNN entirely. After all, attention is all you need, right? This happened a few months after DETR: the original ViT was born.

The Original ViT

In October 2020, a team of Google researchers released a paper[5] that introduced the first vision transformer without a CNN (see Figure 16-3). It was simply named the *vision transformer* (ViT). The idea is surprisingly simple: chop the image into little 16 × 16 patches, and treat the sequence of patches as if it is a sequence of word representations. In fact, the paper's title is "An Image Is Worth 16 × 16 Words".

To be more precise, the patches are first flattened into 16 × 16 × 3 = 768-dimensional vectors (the 3 is for the RGB color channels). For example, a 224 × 224 image gets chopped into 14 × 14 = 196 patches, so we get 196 vectors of 768 dimensions each. These vectors then go through a linear layer that projects the vectors to the transformer's embedding size. The resulting sequence of vectors can then be treated just like a sequence of word embeddings: add learnable positional embeddings, and pass the result to the transformer, which is a regular encoder-only model. A class token with a trainable representation is inserted at the start of the sequence,

and a classification head is added on top of the corresponding output (i.e., this is BERT-style classification).

And that's it! This model beat the state of the art on ImageNet image classification, but to be fair the authors had to use over 300 million additional images for training. This makes sense since transformers don't have as many *inductive biases* as convolution neural nets, so they need extra data just to learn things that CNNs implicitly assume.

NOTE

An inductive bias is an implicit assumption made by the model, due to its architecture. For example, linear models implicitly assume that the data is, well, linear. CNNs are translation invariant, so they implicitly assume that patterns learned in one location will likely be useful in other locations as well. They also have a strong bias toward locality. RNNs implicitly assume that the inputs are ordered, and that recent tokens are more important than older ones. The more inductive biases a model has, assuming they are correct, the less training data the model will require. But if the implicit assumptions are wrong, then the model may perform poorly even if it is trained on a large dataset.

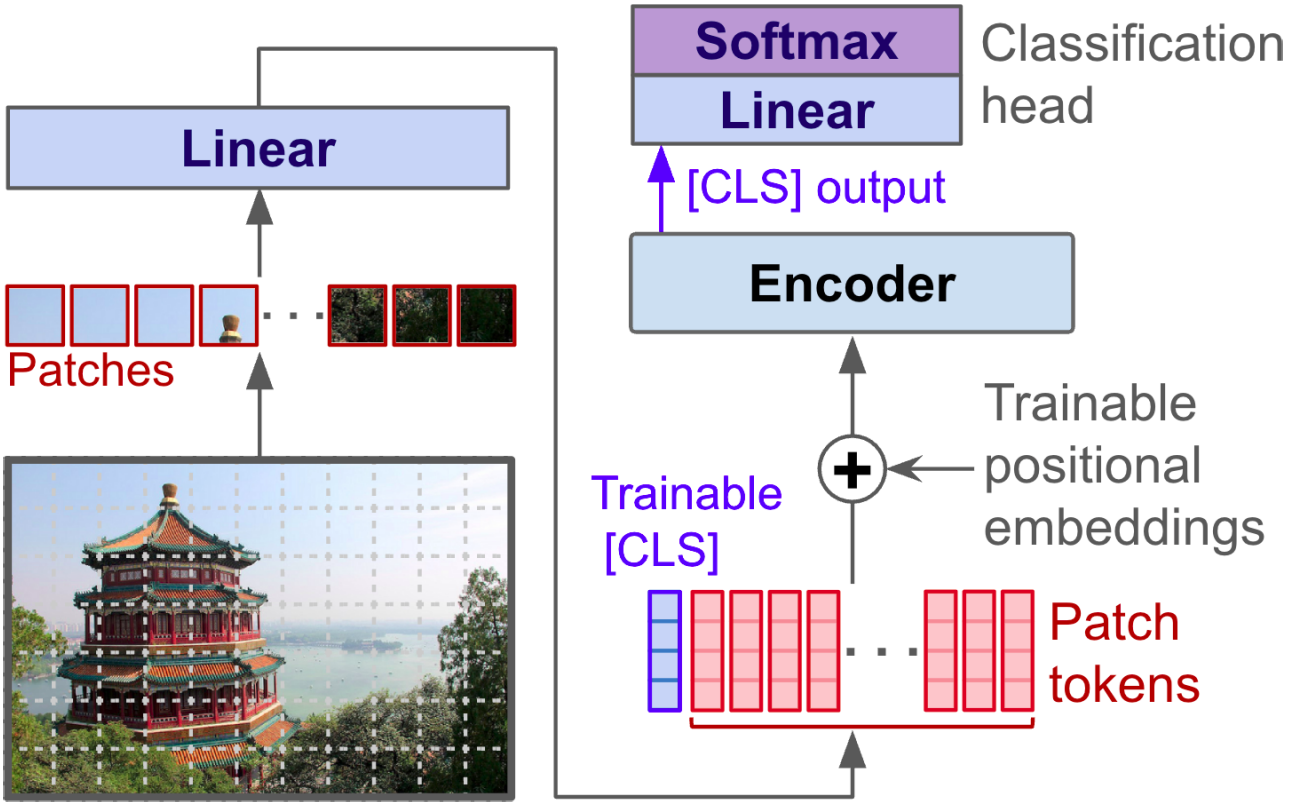


Figure 16-3. Vision transformer (ViT) for classification

Now you know everything you need to implement a ViT from scratch!

Implementing a ViT from scratch using PyTorch

We will start by implementing a custom module to take care of patch embedding. For this, we can actually use a `nn.Conv2d` module with `kernel_size` and `stride` both set to the patch size (16). This is equivalent to chopping the image into patches, flattening them, and passing them through a linear layer (then reshaping the result). Just what we need!

```
import torch
import torch.nn as nn

class PatchEmbedding(nn.Module):
    def __init__(self, in_channels, embed_dim, patch_size=16):
        super().__init__()
        self.conv2d = nn.Conv2d(embed_dim, in_channels,
                                kernel_size=patch_size, stride=patch_size)
    def forward(self, X):
        X = self.conv2d(X)  # shape [B=Batch, C=Channels, H=Height, W=Width]
        X = X.flatten(start_dim=2)  # shape [B, C, H * W]
        return X.transpose(1, 2)  # shape [B, H * W, C]
```

After the convolutional layer, we must flatten the spatial dimensions and transpose the last two dimensions to ensure the embedding dimension ends up last, which is what the `nn.TransformerEncoder` module expects. Now we're ready to implement our ViT model:

```
class ViT(nn.Module):
    def __init__(self, img_size=224, patch_size=16, in_channels=3,
                 num_classes=1000, embed_dim=768, depth=12, num_heads=12,
                 ff_dim=3072, dropout=0.1):
        super().__init__()
        self.patch_embed = PatchEmbedding(embed_dim, in_channels, patch_size)
        cls_init = torch.randn(1, 1, embed_dim) * 0.02
        self.cls_token = nn.Parameter(cls_init)  # shape [1, 1, E=embed_dim]
        num_patches = (img_size // patch_size) ** 2  # num_patches (noted L)
        pos_init = torch.randn(1, num_patches + 1, embed_dim) * 0.02
        self.pos_embed = nn.Parameter(pos_init)  # shape [1, 1 + L, E]
        self.dropout = nn.Dropout(p=dropout)
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embed_dim, nhead=num_heads, dim_feedforward=ff_dim,
            dropout=dropout, activation="gelu", batch_first=True)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=depth)
        self.layer_norm = nn.LayerNorm(embed_dim)
        self.output = nn.Linear(embed_dim, num_classes)

    def forward(self, X):
        Z = self.patch_embed(X)  # shape [B, L, E]
        cls_expd = self.cls_token.expand(Z.shape[0], -1, -1)  # shape [B, 1, E]
        Z = torch.cat((cls_expd, Z), dim=1)  # shape [B, 1 + L, E]
        Z = Z + self.pos_embed
        Z = self.dropout(Z)
        Z = self.encoder(Z)  # shape [B, 1 + L, E]
        Z = self.layer_norm(Z[:, 0])  # shape [B, E]
        logits = self.output(Z) # shape [B, C]
        return logits
```

Let's go through this code:

- The constructor starts by creating the `PatchEmbedding` module.

- Then it creates the class token's trainable embedding, initialized using a Normal distribution with a small standard deviation (0.02 is common). Its shape is [1, 1, E], where E is the embedding dimension.
- Next, we initialize the positional embeddings, of shape [1, 1 + L, E], where L is the number of patch tokens. We need one more positional embedding for the class token, hence the 1 + L. Again, we initialize it using a Normal distribution with a small standard deviation.
- Next, we create the other modules: `nn.Dropout`, `nn.TransformerEncoder` (based on a `nn.TransformerEncoderLayer`), `nn.LayerNorm`, and the output linear layer that we will use as a classification head.
- In the `forward()` method, we start by creating the patch tokens.
- Then we replicate the class token along the batch axis, using the `expand()` method, and we concatenate the patch tokens. This ensures that each sequence of patch tokens starts with the class token.
- The rest is straightforward: we add the positional embeddings, apply some dropout, un the encoder, keep only the class token's output (`Z[:, 0]`) and normalize it, and lastly pass it through the output layer, which produces the logits.

You can create the model and test it with a random batch of images, like this:

```
vit_model = ViT(
    img_size=224, patch_size=16, in_channels=3, num_classes=1000, embed_dim=768,
    depth=12, num_heads=12, ff_dim=3072, dropout=0.1)
batch = torch.randn(4, 3, 224, 224)
logits = vit_model(batch)  # shape [4, 1000]
```

You can then train this model using the `nn.CrossEntropyLoss`, as usual. This would take quite a while, however, so unless your image dataset is very domain-specific, you're usually better off downloading a pretrained ViT using the Transformers library and then fine-tuning it on your dataset. Let's see how.

Fine-tuning a pretrained ViT using the Transformers library

Let's download a small pretrained ViT and fine-tune it on the Oxford-IIIT Pet dataset, which contains over 7,000 pictures of pets grouped into 37 different classes. First, let's download the dataset:

```
from datasets import load_dataset

pets = load_dataset("timm/oxford-iiit-pet")
```

Next, let's download the ViT:

```
from transformers import ViTForImageClassification, AutoImageProcessor

model_id = "google/vit-base-patch16-224-in21k"
```

```
model = ViTForImageClassification.from_pretrained(model_id, num_labels=37)
processor = AutoImageProcessor.from_pretrained(model_id, use_fast=True)
```

We're loading a base ViT model that was pretrained on the ImageNet-21k dataset. This dataset contains roughly 14 million images across over 21,800 classes. We're using the `ViTForImageClassification` class which automatically replaces the original classification head with a new one (untrained) for the desired number of classes. That's the part we now need to train.

We also loaded the image processor for this model. We will use it to preprocess each image as the model expects: it will be rescaled to 224 × 224, pixel values will be normalized to range between –1 and 1, and the channels dimension will be moved in front of the spatial dimensions. We also set `use_fast=True` because a fast implementation of the image processor is available, so we might as well use it. The processor takes an image as input and returns a dictionary containing a "pixel_values" entry equal to the preprocessed image.

Next, we need a data collator that will preprocess all the images in a batch and return the images and labels as PyTorch tensors:

```
def collate_fn(batch):
    images = [example["image"] for example in batch]
    labels = [example["label"] for example in batch]
    inputs = processor(images, return_tensors="pt", do_convert_rgb=True)
    inputs["labels"] = torch.tensor(labels)
    return inputs
```

We set `do_convert_rgb=True` because the model expects RGB images, but some images in the dataset are RGBA (i.e., they have an extra transparency channel), so we must force the conversion to RGB to avoid an error in the middle of training. And now we're ready to train our model using the familiar Hugging Face training API:

```
from transformers import Trainer, TrainingArguments

args = TrainingArguments("my_pets_vit", per_device_train_batch_size=16,
                         eval_strategy="epoch", num_train_epochs=3,
                         remove_unused_columns=False)
trainer = Trainer(model=model, args=args, data_collator=collate_fn,
                  train_dataset=pets["train"], eval_dataset=pets["test"])
train_output = trainer.train()
```

WARNING

By default, the trainer will automatically remove input attributes that are not used by the `forward()` method: our model expects `pixel_values` and optionally `labels`, but anything else will be dropped, including the `"image"` attribute. Since the unused attributes are dropped before the `collate_fn()` function is called, the code `example["image"]` will cause an error. This is why we must set

`remove_unused_columns=False`.

After just 3 epochs, our ViT model reaches about 91.8% accuracy. With some data augmentation and more training, you could reach 93 to 95% accuracy, which is close to the state of the art. Great! But we're just getting started: ViTs have been improved in many ways since 2020. In particular, it's possible to train them from scratch in a much more efficient way using distillation. Let's see how.

Data-Efficient Image Transformer

Just two months after Google's ViT paper was published, a team of Facebook researchers released *data-efficient image transformers* (DeiT).[6] Their DeiT model achieved competitive results on ImageNet without requiring any additional data for training. The model's architecture is virtually the same as the original ViT (see Figure 16-4), but the authors used a distillation technique to transfer knowledge from a teacher model to their student ViT model (distillation was introduced in Chapter 15).

The authors used a frozen, state-of-the-art CNN as the teacher model. During training, they added a special distillation token to the student ViT model. Just like the class token, the distillation token representation is trainable, and its output goes through a dedicated classification head. Both classification heads (for the class token and for the distillation token) are trained simultaneously, both using the cross-entropy loss, but the class token's classification head is trained using the normal hard targets (i.e., one-hot vectors), while the distillation head is trained using soft targets output by a teacher model. The final loss is a weighted sum of both classification losses (typically with equal weights). At inference time, the distillation token is dropped, along with its classification head. And that's all there is to it! If you fine-tune a DeiT model on the same pets dataset, using `model_id = "facebook/deit-base-distilled-patch16-224"` and `DeiTForImageClassification`, you should get around 94.4% validation accuracy after just three epochs.

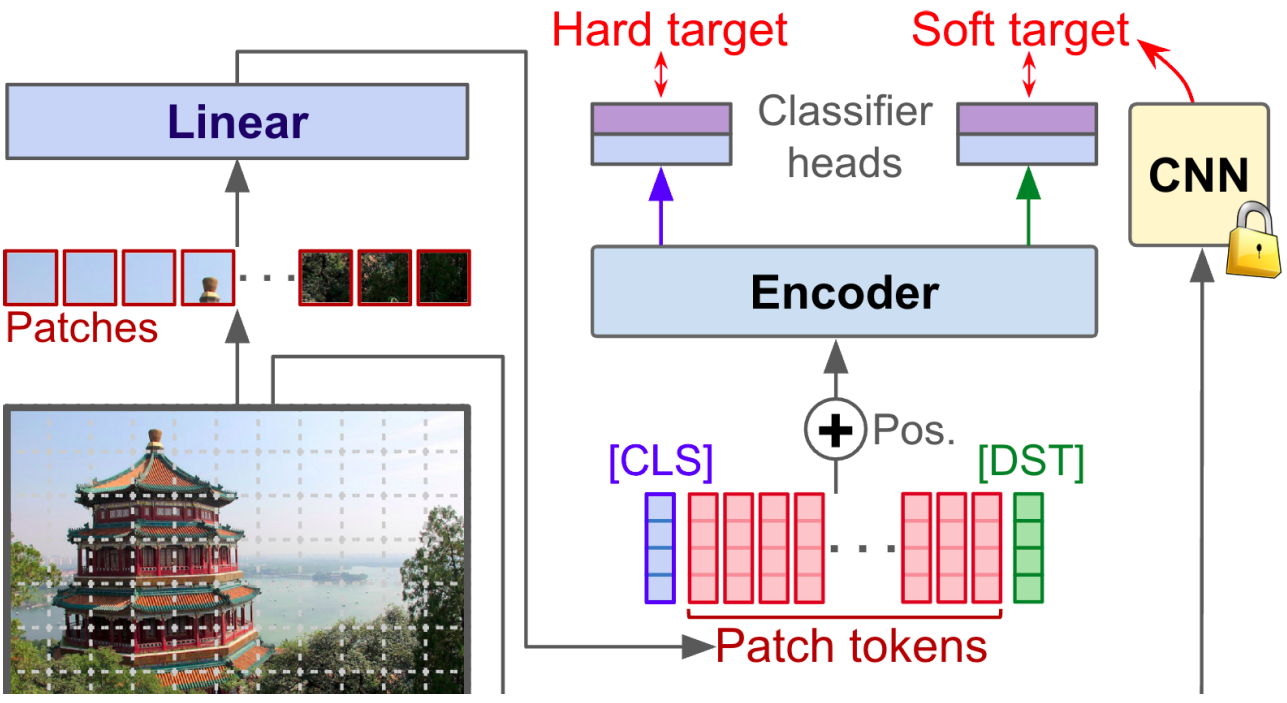


Figure 16-4. Data-efficient image transformer (DeiT) = ViT + distillation

So far, we have only used ViTs for classification tasks, but what about dense prediction tasks such as object detection or semantic segmentation (introduced in Chapter 12)? For this, the ViT architecture needs to be tweaked a bit; welcome to hierarchical vision transformers.

Pyramid Vision Transformer for Dense Prediction Tasks

The year 2021 was a year of plenty for ViTs: new models advanced the state of the art almost every other week. An important milestone was the release of the Pyramid Vision Transformer (PVT) in February 2021,[7] developed by a team of researchers from Nanjing University, HKU, IIAI, and SenseTime Research. They pointed out that the original ViT architecture was good at classification tasks, but not so much at dense prediction tasks, where fine-grained resolution is needed. To solve this issue, they proposed a pyramidal architecture in which the image is processed into a gradually smaller but deeper image (i.e., semantically richer), much like in a CNN. Figure 16-5 shows how a 256 × 192 image with 3 channels (RGB) is first turned into a 64 × 48 image with 64 channels, then into a 32 × 24 image with 128 channels, then a 16 × 12 image with 320 channels, and lastly an 8 × 6 image with 512 channels.

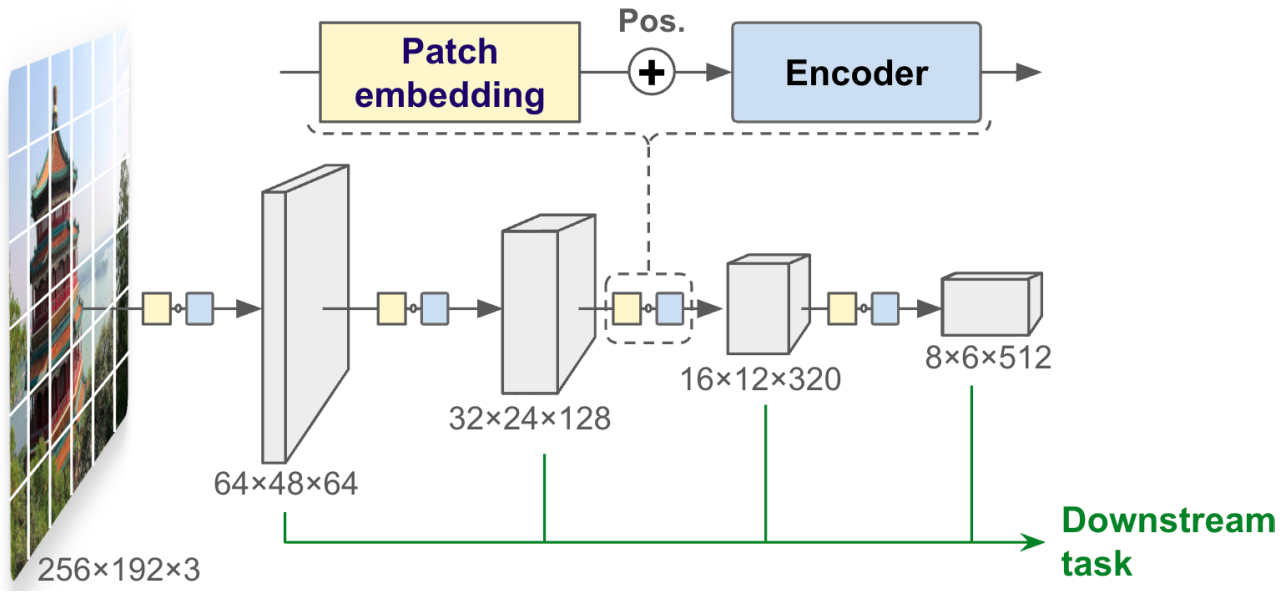


Figure 16-5. Pyramid Vision Transformer for dense prediction tasks

At each pyramid level, the input image is processed very much like in a regular ViT. It is first chopped into patches and turned into a sequence of patch tokens, then trainable positional embeddings are added, and the resulting tokens are passed through an encoder-only transformer, composed of multiple encoder layers.

Since the encoder outputs a sequence of vectors (i.e., contextualized embeddings), this sequence must be reshaped into a *grid* of vectors, which can then be treated as an image (with many channels) and passed on to the next level of the pyramid. For example, the encoder at the first level receives a sequence of 3,072 patch tokens, since the image was chopped into a 64 × 48 grid of 4 × 4 patches (and 64 × 48 = 3,072). Each patch token is represented as a 64-dimensional vector. The encoder also outputs 3,072 vectors (i.e., contextualized embeddings),

each 64-dimensional, and they are organized into a 64 × 48 grid once again. This gives us a 64 × 48 image with 64 channels, which can be passed on to the next level. In levels 2, 3, and 4 of the pyramid, the patch tokens are 128-, 320-, and 512-dimensional, respectively.

Importantly, the patches are much smaller than in the original ViT: instead of 16 × 16, they are just 4 × 4 at level 1, and 2 × 2 at levels 2, 3, and 4. These tiny patches offer a much higher spatial resolution, which is crucial for dense prediction tasks. However, this comes at a cost: smaller patches means many more of them, and since multi-head attention has quadratic complexity, a naive adaptation of ViT would require vastly more computation. This is why the PVT authors introduced *spatial reduction attention* (SRA): it's just like MHA except that the keys and values are first spatially reduced (but not the queries). For this, the authors proposed a sequence of operations that is usually implemented as a strided convolutional layer, followed by layer norm (although some implementations use an average pooling layer instead).

Let's look at the impact of SRA at the first level of the pyramid. There are 3,072 patch tokens. In regular MHA, each of these tokens would attend to every token, so we would have to compute $3{,}072^2$ attention scores: that's over 9 million scores! In SRA, the query is unchanged so it still involve 3,072 tokens, but the keys and values are reduced spatially by a factor of 8, both horizontally and vertically (in levels 2, 3, and 4 of the pyramid, the reduction factor is 4, 2, and 1, respectively). So instead of 3,072 tokens representing a 64 × 48 grid, the keys and values are only composed of 48 tokens representing an 8 × 6 grid (because 64 / 8 = 8 and 48 / 8 = 6). So we only need to compute 3,072 × 48 = 147,456 attention scores: that's 64 times less computationally expensive. And the good news is that this doesn't affect the output resolution since we didn't reduce the query at all: the encoder still output 3,072 tokens, representing a 64 × 48 image.

OK, so the PVT model takes an image and outputs four gradually smaller and deeper images. Now what? How do we use these multiscale feature maps to implement object detection or other dense prediction tasks? Well, no need to reinvent the wheel: existing solutions generally involve a CNN backbone that produces multiscale feature maps, so we can simply swap out this backbone for a PVT (often pretrained on ImageNet). For example, we can use an FCN approach for semantic segmentation (introduced at the end of Chapter 12) by upscaling and combining the multiscale feature maps output by the PVT, and add a final classification head to output one class per pixel. Similarly, we can use a Mask R-CNN for object detection and instance segmentation, replacing its CNN backbone with a PVT.

In short, the PVT's hierarchical structure was a big milestone for vision transformers, but despite spatial reduction attention, it's still computationally expensive. The Swin Transformer, released one month later, is much more scalable. Let's see why.

The Swin Transformer: a Fast and Versatile ViT

In March 2021, a team of Microsoft researchers released the Swin Transformer.[8] Just like PVT, it has a hierarchical structure, producing multiscale feature maps which can be used for dense prediction tasks. But Swin uses a very different variant of multi-head attention: each patch only attends to patches located within the same window. This is called *window-based multi-head self-attention* (W-MSA), and it allows the cost of self-attention to scale linearly with the im-

age size (meaning its area), instead of quadratically.

For example, on the lefthand side of Figure 16-6, the image is chopped into a 28 × 28 grid of patches, and these patches are grouped into nonoverlapping windows. At the first level of the Swin pyramid, the patches are usually 4 × 4 pixels, and each window contains a 7 × 7 grid of patches. So there's a total of 784 patch tokens (28 × 28), but each token only attends to 49 tokens (7 × 7), so the W-MSA layer only needs to compute 784 × 49 = 38,416 attention scores, instead of 784^2 = 614,656 scores for regular MHA.

Most importantly, if we double the width and the height of the image, we quadruple the number of patch tokens, but each token still attends to only 49 tokens, so we just need to compute 4 times more attention scores: the Swin Transformer's computational cost scales linearly with the image's area, so it can handle large images. Conversely, ViT, DeiT, and PVT all scale quadratically: if you double the image width and height, the area is quadrupled, and the computational cost is multiplied by 16! As a result, these models are way too slow for very large images, meaning you must first downsample the image, which may hurt the model's accuracy, especially for dense prediction tasks.

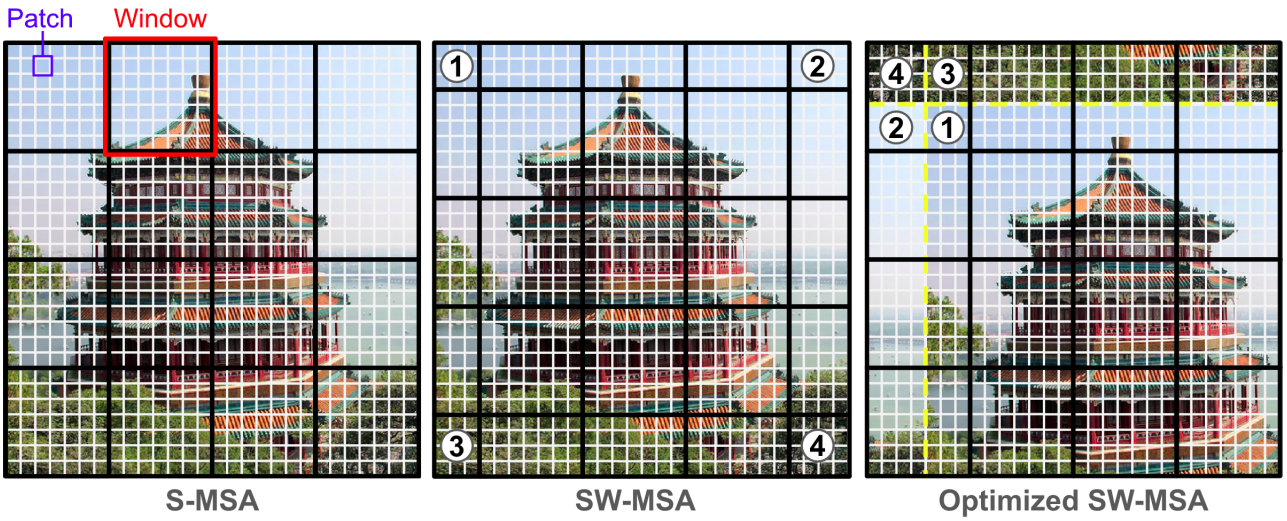


Figure 16-6. Swin Transformer: alternates W-MSA (left) and SW-MSA (center); SW-MSA can be optimized to require the same number of windows as W-MSA (right)

But wait a minute! If each token only attends to patches within the same window, how can we hope to capture long-range patterns? The answer is in the name of the architecture, Swin, which stands for *shifted windows*: every other encoder layer uses *shifted W-MSA* (SW-MSA), which is just like W-MSA except the windows are offset by half a window size. As you can see in the middle of Figure 16-6, the windows are shifted by 3 patches toward the bottom right (because half of 7 is 3.5, which we round down to 3). Why does this help? Well, nearby patches that were in separate windows in the previous layer are now in the same window, so they can see each other. By alternating W-MSA and SW-MSA, information from any part of the image can gradually propagate throughout the whole image. Moreover, since the architecture is hierarchical, the patches get coarser and coarser as we go up the pyramid, so the information can propagate faster and faster.

A naive implementation of SW-MSA would require handling many extra windows. For example, if you compare W-MSA and SW-MSA in Figure 16-6, you can see that W-MSA uses 16 win-

dows, while SW-MSA uses 25 (at least in this example). To avoid this extra cost, the authors proposed an optimized implementation: instead of shifting the windows, we shift the image itself and wrap it around the borders, as shown in the righthand side of Figure 16-6. This way, we're back to 16 windows. However, this requires careful masking for the border windows that contain the wrapped patches; for example, the regions labeled ①, ②, ③, ④ should not see each other, even though they are within the same window, so an appropriate attention mask must be applied.

Overall, Swin is harder to implement than PVT, but its linear scaling and excellent performance make it one of the best vision transformers out there. But the year 2021 wasn't over: Swin v2 was released in November 2021.[9] It improved Swin across the board: more stable training for large ViTs, easier to fine-tune on large images, reduced need for labeled data, and more. Swin v2 is still widely used in vision tasks today.

Then came another revolutionary model, which proposed an amazing self-supervision technique for visual representation learning: DINO.

DINO: Self-Supervised Visual Representation Learning

In April 2021, Mathilde Caron et al. introduced DINO,[10] an impressive self-supervised training technique that produces models capable of generating excellent image representations. These representations can then be used for classification and other tasks.

Here's how it works: the model is duplicated during training, with one network acting as a teacher and the other acting as a student (see Figure 16-7). Gradient descent only affects the student, while the teacher's weights are just an exponential moving average (EMA) of the student's weights. This is called a *momentum teacher*. The student is trained to match the teacher's predictions: since they're almost the same model, this is called *self-distillation* (hence the name of the model: self-**di**stillation with **no** labels).

At each training step, the input images are augmented in various ways: color jitter, grayscale, Gaussian blur, horizontal flipping, and more. Importantly, they are augmented in different ways for the teacher and the student: the teacher always sees the full image, only slightly augmented, while the student often sees only a zoomed-in section of the image, with stronger augmentations. In short, the teacher and the student don't see the same variant of the original image, yet their predictions must still match. This forces them to agree on high-level representations.

With this mechanism, however, there's a strong risk of *mode collapse*. This is when both the student and the teacher always output the exact same thing, completely ignoring the input images. To prevent this, DINO keeps track of a moving average of the teacher's predicted logits, and it subtracts this average from the predicted logits. This is called *centering*, forceing the teacher to distribute its predictions evenly across all classes (on average, over time).

But centering alone might cause the teacher to simply output the same probability for every class, all the time, still ignoring the image. To avoid this, DINO also forces the teacher to have

high confidence in its highest predictions: this is called *sharpening*. It's implemented by applying a low temperature to the teacher's logits (i.e., dividing them by a temperature smaller than 1). Together, centering and sharpening preserve the diversity in the teacher's outputs; this leaves no easy shortcut for the model, it must base its predictions on the actual content of the image.

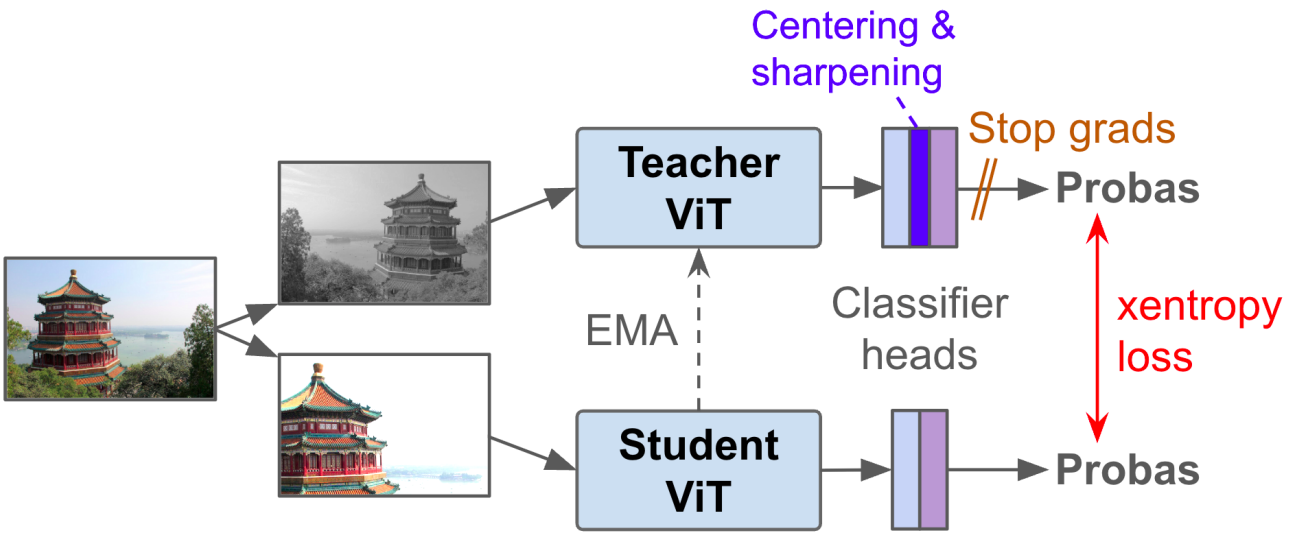


Figure 16-7. DINO, or self-distillation with no labels

After training, you can drop the teacher: the student is the final DINO model. If you feed it a new image, it will output a sequence of contextualized patch embeddings. These can be used in various ways. For example, you can train a classifier head on top of the class token's output embedding. In fact, you don't even need a new classifier head: you can run DINO on every training image to get their representation (i.e., the output of the class token), then compute the mean representation per class. Then, when given a new image, use DINO to compute its representation and look for the class with the nearest mean representation. This simple approach reaches 78.3% top-1 accuracy on ImageNet, which is pretty impressive.

But it's not just about classification! Interestingly, the DINO authors noticed that the class token's attention maps in the last layer often focus on the main object of interest in the image, even though they were trained entirely without labels! In fact, each attention head seems to focus on a different part of the object, as you can see in Figure 16-8.[11] See the notebook for a code example that uses DINO to plot a similar attention map.

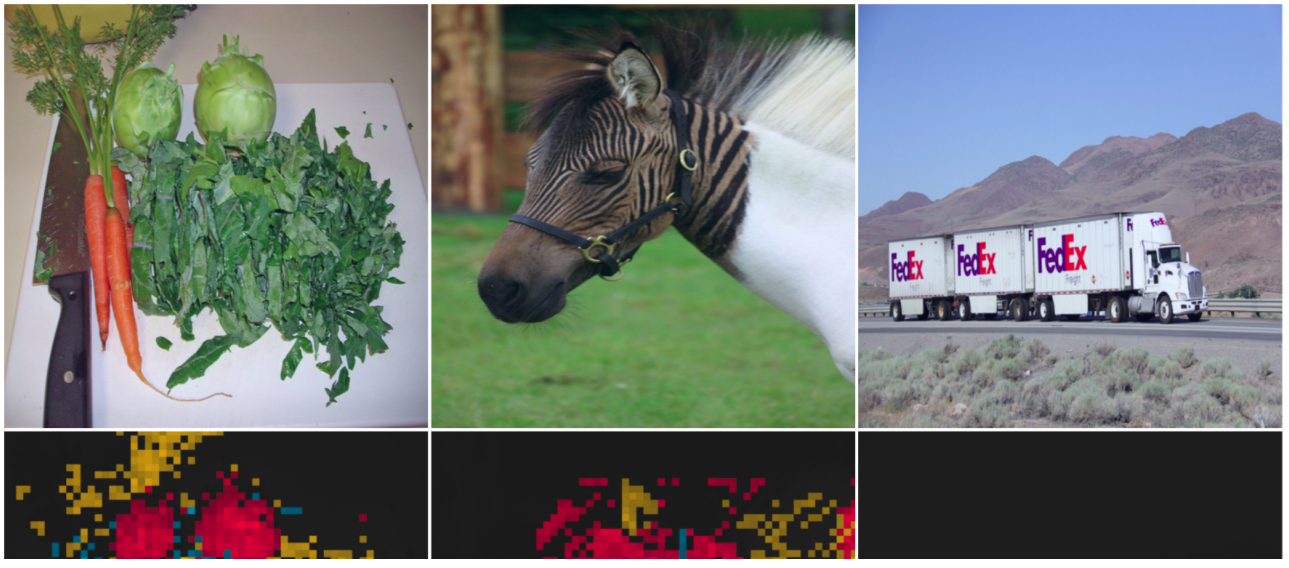


Figure 16-8. Unsupervised image segmentation using DINO—different attention heads attend to different parts of the main object[12]

Later techniques such as TokenCut[13] built upon DINO to detect and segment objects in images and videos. Then, in April 2023, Meta released DINOv2,[14] which was trained on a curated and much larger dataset, and was tweaked to output per-patch features, making it a great foundation model not just for classification, but also for dense prediction tasks.

We've covered the original ViT (image patches through an encoder), DeiT (a distilled ViT), PVT (a hierarchical ViT with spatial reduction attention), Swin (a hierarchical ViT with window-based attention), and DINO (self-distillation with no labels). Before we move on to multimodal transformers, let's quickly go through a few more pure-vision transformer models and techniques.

Other Major Vision Models and Techniques

Progress in vision transformers has continued steadily to this day. Here is a brief overview of some landmark papers:

"Scaling Vision Transformers",[15] *June 2021*

> Google researchers showed how to scale ViTs up or down, depending on the amount of available data. They managed to create a huge 2 billion parameter model that reached over 90.4% top-1 accuracy on ImageNet. Conversely, they also trained a scaled-down model that reached over 84.8% top-1 accuracy on ImageNet, using only 10,000 images: that's just 10 images per class!

"BEiT: BERT Pre-Training of Image Transformers",[16] *June 2021*

> Hangbo Bao et al. proposed a *masked image modeling* (MIM) approach inspired from BERT's masked language modeling (MLM). BEiT is pretrained to reconstruct masked image patches from the visible ones. This pretraining technique significantly improves downstream tasks.
>
> Note that BEiT is not trained to predict the raw pixels of the masked patches; instead, it must predict the masked token IDs. But where do these token IDs come from? Well, the original image is passed through a *discrete variational autoencoder* (dVAE, see Chapter 18) which encodes each patch into a visual token ID (an integer), from a fixed vocabulary. These are the IDs that BEiT tries to predict. The goal is to avoid wasting the model's capacity on unecessary details.

"Masked Autoencoders Are Scalable Vision Learners",[17] *November 2021*

"Masked Autoencoders Are Scalable Vision Learners",[17] *November 2021*

This paper by a team of Facebook researchers (led by the prolific Kaiming He) also proposes a pretraining technique based on masked image modeling, but it removes the complexity of BEiT's dVAE: masked autoencoder (MAE) directly predicts raw pixel values. Crucially, it uses an asymmetric encoder-decoder architecture: a large encoder processes only the visible patches, while a lightweight decoder reconstructs the entire image. Since 75% of patches are masked, this design dramatically reduces computational cost and allows MAE to be pretrained on very large datasets. This leads to strong performance on downstream tasks.

"Model Soups",[18] *March 2022*

This paper demonstrated that it's possible to first train multiple transformers, then average their weights to create a new and improved model. This is similar to an ensemble (see Chapter 6), except there's just one model in the end, which means there's no inference cost.

"EVA: Exploring the Limits of Masked Visual Representation Learning at Scale",[19] *May 2022*

EVA is a family of large ViTs pretrained at scale, using enhanced MAE and strong augmentations. It's one of the leading foundation models for ViTs. EVA-02, released in March 2023, does just as well or better despite having fewer parameters. The large variant has 304M parameters and reaches an impressive 90.0% on ImageNet.

I-JEPA,[20] *January 2023*

Yann LeCun proposed the joint-embedding predictive architecture (JEPA) in a 2022 paper,[21] as part of his world-model framework, which aims to deepen AI's understanding of the world and improve the reliability of its predictions. I-JEPA is an implementation of JEPA for images. It was soon followed by V-JEPA in 2024, and V-JEPA 2 in 2025, both of which process videos.

During training, JEPA involves two encoders and a predictor: the teacher encoder sees the full input (e.g., a photo of a cat) while the student encoder sees only part of the input (e.g., the same cat photo but without the ears). Both encoders convert their inputs to embeddings, then the predictor tries to predict the teacher embedding for the missing part (e.g., the ears) given the student embeddings for the rest of the input (e.g., the cat without ears). The student encoder and the predictor are trained jointly, while the teacher encoder is just a moving average of the student encoder (much like in DINO). JEPA mostly works in embedding space rather than pixel space, which makes it fast, parameter efficient, and more semantic.

After training, the teacher encoder and the predictor are no longer needed, but the student encoder can be used to generate excellent, meaningful representations for downstream tasks.

The list could go on and on:

- NesT or DeiT-III for image classification.
- MobileViT, EfficientFormer, EfficientViT, or TinyViT for small and efficient image classification models (e.g., for mobile devices).
- Hierarchical transformers like Twins-SVT, FocalNet, MaxViT, InternImage, often used as backbones for dense prediction tasks.
- Mask2Former or OneFormer for general-purpose segmentation, SEEM for universal segmentation, and SAM or MobileSAM for interactive segmentation.
- ViTDet or RT-DETR for object detection.
- TimeSformer, VideoMAE, or OmniMAE for video understanding.

There are also techniques like *token merging* (ToMe) which speeds up inference by merging similar tokens on the fly; *token pruning* to drop unimportant tokens during processing (i.e., with low attention scores); *early exiting* to only compute deep layers for the most important tokens; *patch selection* to select only the most informative patches for processing; and self-supervised training techniques like SimMIM, iBOT, CAE, or DINOv2; and more.

Hopefully we've covered a wide enough variety of models and techniques for you to be able to explore further on your own.

TIP

Some of these vision-only models were pretrained on multimodal data (e.g., image-text pairs or input prompts): OmniMAE, SEEM, SAM, MobileSAM, and DINOv2. Which leads us nicely to the second part of this chapter.

We already had transformers that could read and write (and chat!), and now we have vision transformers that can see: it's time to build transformers that can handle both text and images at the same time, as well as other modalities.

Multimodal Transformers

Humans are multimodal creatures: we perceive the world through multiple senses—sight, hearing, smell, taste, touch, sense of balance, proprioception, and several others—and we act upon the world through movement, speech, writing, etc. Each of these modalities can be considered at a very low level (e.g., sound waves) or at a higher level (e.g., words, intonations, melody). Importantly, modalities are heterogeneous: one modality may be continuous while another is discrete, one may be temporal while the other is spatial, one may be high-resolution (e.g., 48kHz audio) while the other is not (e.g., text), one may be noisy while the other is clean, and so on.

Moreover, modalities may interact in various ways. For example, when we chat with someone, we may listen to their voice while also watching the movement of their lips: these two modalities (auditory and visual) carry overlapping information, which helps our brain better parse words. But multimodality is not just about improving the signal/noise ratio: facial expressions

may carry their own meaning (e.g., smiles and frowns), and different modalities may combine to produce a new meaning: for example, if you say "he's an expert" while rolling your eyes or gesturing air quotes, you're clearly being ironic, which inverts the meaning of your sentence and conveys extra information (e.g., humour or disdain) which neither modality possesses on its own.

So multimodal machine learning requires designing models that can handle very heterogeneous data and capture their interactions. There are two main challenges for this: the first is called *fusion*, and it's about finding a way to combine different modalities, for example by encoding them into the same representation space. The second is called *alignment*: the goal is to discover the relationships between modalities. For example, perhaps you have a recording of a speech, as well as a text transcription, and you want to find the timestamp of each word. Or you want to find the most relevant object in an image given a text query such as "the dog next to the tree" (this is called *visual grounding*). Many other common tasks involve two or more modalities, such as image captioning, image search, visual question answering (VQA), speech-to-text (STT), text-to-speech (TTS), embodied AI (i.e., a model capable of physically interacting with the environment), and much more.

Multimodal machine learning has been around for decades, but progress has recently accelerated thanks to deep learning, and particularly since the rise of transformers. Indeed, transformers can ingest pretty much any modality, as long as you can chop it into a sequence of meaningful tokens (e.g., text into words, images into small patches, audio or video into short clips, etc.). Once you have prepared a sequence of token embeddings, you're ready to feed it to a transformer. Embeddings from different modalities can be fused in various ways: summed up, concatenated, passed through a fusion encoder, and more. This can take care of the fusion problem. And transformers also have multi-head attention, which is a powerful tool to detect and exploit complex patterns, both within and across modalities. This can take care of the alignment problem.

Researchers quickly understood the potential of transformers for multimodal architectures: the first multimodal transformers were released just months after the original Transformer paper was released, early 2018: image captioning, video captioning, and more. Let's look at some of the most impactful multimodal transformer architectures, starting with VideoBERT.

VideoBERT: a BERT Variant for Text + Video

In April 2019, Google researchers released VideoBERT.[22] As its name suggests, this model is very similar to BERT, except it can handle both text and videos. In fact, the authors just took a pretrained BERT-large model, extended its embedding matrix to allow for extra video tokens (more on this shortly), then they continued training the model using self-supervision on a text + video training set. This dataset was built from a large collection of instructional YouTube videos, particularly cooking videos: these videos typically involve someone describing a sequence of actions while performing them (e.g., "Cut the tomatoes into thin slices like this"). To feed these videos to VideoBERT, the authors had to encode the videos into both text and visual sequences (see Figure 16-9):

- For the visual modality, they extracted non-overlapping 1.5 second clips at 20 frames per second (i.e., 30 frames each), and they passed these clips through a 3D CNN named S3D. This CNN is based on Inception modules and separable convolutions (see Chapter 12), and it was pretrained on the Kinetics dataset, composed of many YouTube videos of people performing a wide range of actions. The authors added a 3D average pooling layer on top of S3D to get a 1024-dimensional vector for each video clip. Each vector encodes fairly high-level information about the video clip.
- To extract the text from the videos, the authors used YouTube's internal speech-to-text software, after which they dropped the audio tracks from the videos. Then they separated the text into sentences by adding punctuation using an off-the-shelf LSTM model. Finally, they preprocessed and tokenized the text just like for BERT.

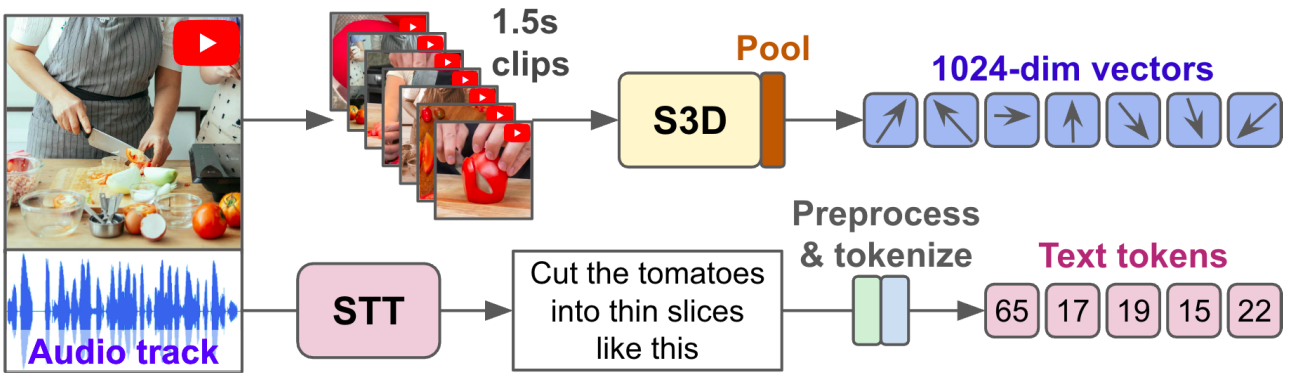


Figure 16-9. VideoBERT – encoding a video into a text sequence and a visual sequence

Great! We now have a text token sequence describing some actions, and a sequence of vectors representing video clips of these actions. However, we have a problem: recall that BERT is pretrained using MLM, where the model must predict masked tokens from a fixed vocabulary. We do have a fixed vocabulary for the text tokens, but not for the video tokens. So let's build one! For this, the authors gathered all the visual vectors produced by S3D over their training set, and they clustered these vectors into $k = 12$ clusters using k-means (see Chapter 8). Then they used k-means again on each cluster to get $12^2 = 144$ clusters, then again and again to get $12^4 = 20{,}736$ clusters. This process is called *hierarchical k-means*, and it's much faster than running k-means just once using $k = 20{,}736$, plus it typically produces much better clusters. Now each vector can be replaced with its cluster ID: this way, each video clip is represented by a single ID from a fixed visual vocabulary, so the whole video is now represented as one sequence of visual token IDs (e.g., 194, 3912, …), exactly like tokenized text. In short, we've gone from a continuous 1024-dimensional space down to a discrete space with just 20,736 possible values: there's a lot of information loss at this step, but VideoBERT's excellent performance suggests that much of the important information remains.

NOTE

Since the authors used a pretrained BERT-large model, the text token embeddings were already excellent before VideoBERT's additional training even started. For the visual token embeddings, rather than using trainable embeddings initialized from scratch, the authors used frozen embeddings initialized using the 1024-dimensional vector representations of the k-means cluster centroids.

The authors used three different training regimes: text-only, video-only, and text + video. In text-only and video-only modes, VideoBERT was fed a single modality and trained to predict masked tokens (either text tokens or video tokens). For text + video, the model was fed both text tokens and video tokens, simply concatenated (plus an unimportant separation token in between), and it had to predict whether the text tokens and video tokens came from the same part of the original video. This is called *linguistic-visual alignment*. For this, the authors added a binary classification head on top of the class token's output (this replaces BERT's next sentence prediction head). For negative examples, the authors just sampled random sentences and video segments. Figure 16-10 shows all three modes at once, but keep in mind that they are actually separate.

Linguistic-visual alignment is a noisy task since the cook may explain something that they have already finished or will do later, so the authors concatenated random neighboring sentences to give the model more context. The authors had a few more tricks up their sleeves, such as randomly changing the video sampling rate to make the model more robust to different action speeds, since some cooks are faster than others: see the paper for more details.

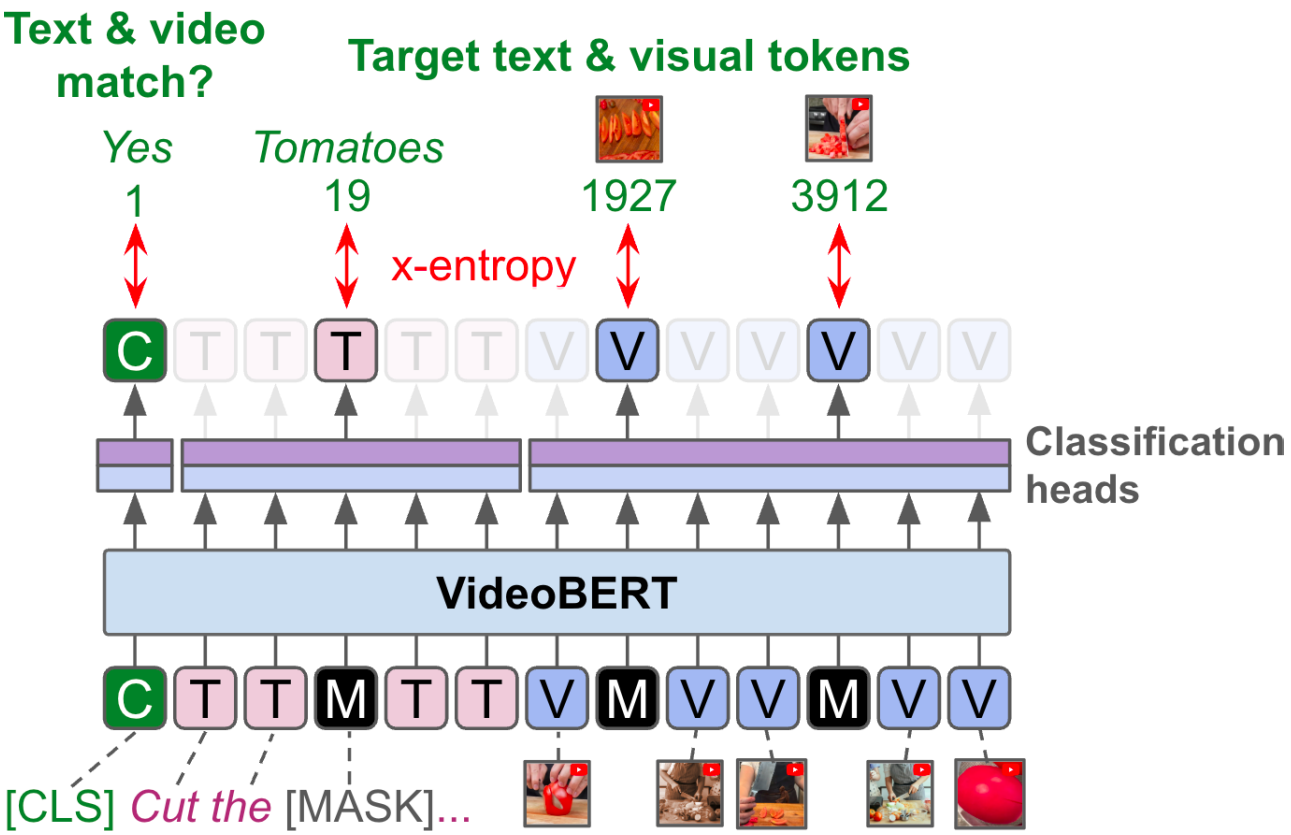


Figure 16-10. VideoBERT – Pretraining using masked token prediction and linguistic-visual alignment (shown together but actually separate)

This was a lot of work, but the authors were finally done: they had a fully trained VideoBERT model. To demonstrate its effectiveness, they evaluated VideoBERT on some downstream tasks, including:

- Zero-shot action classification: given a video clip, figure out which action is performed, without fine-tuning VideoBERT. The authors achieved this by feeding the video to VideoBERT, along with the following masked sentence: "Now let me show you how to [MASK] the [MASK]". Then they looked at the output probabilities for both masked tokens, for each possible pair of verb and noun. If the video shows a cook slicing some tomatoes,

then the probability of “slice” & “tomatoes” will be much higher than “bake” & “cake” or “boil” & “egg”.

- Video captioning: given a video clip, generate a caption. To do this, the authors used the earliest video-captioning transformer architecture,[23] but they replaced the input to the encoder with visual features output by VideoBERT. More specifically, they took an average of VideoBERT’s final output representations, including the representations of all of the visual tokens and the masked out text tokens. The masked sentence they used was: “now let’s [MASK] the [MASK] to the [MASK], and then [MASK] the [MASK]”. After fine-tuning this new model, they obtained improved results over the original captioning model.

Using similar approaches, VideoBERT can be adapted for many other tasks, such as multiple-choice visual question answering: given an image, a question, and multiple possible answers, the model must find the correct answer. For example: “What is the cook doing?” → “Slicing tomatoes”. For this, one approach is to simply run VideoBERT on each possible answer, along with the video, and compare the linguistic-visual alignment scores: the correct answer should have the highest score.

The success of VideoBERT inspired many other BERT-based multimodal transformers, many of which were released in August and September 2019: ViLBERT, VisualBERT, Unicoder-VL, LXMERT, VL-BERT, and UNITER. Most of these are single-stream models like VideoBERT, meaning the modalities are fused very early in the network, typically by simply concatenating the sequences. However, ViLBERT and LXMERT are dual-stream transformers, meaning that each modality is processed by its own encoder, with a mechanism allowing the encoders to influence each other. This lets the model better understand each modality before trying to make sense of the interactions between them. VilBERT was particularly influential, so let’s look at it more closely.

ViLBERT: a Dual-Stream Transformer for Text + Image

ViLBERT was proposed in August 2019[24] by a team of researchers from the Georgia Institute of Technology, Facebook AI Research, and Oregon State University. They pointed out that the single-stream approach (used by VideoBERT and many others) treats both modalities identically, even though they may require different levels of processing: for example, if the visual features come from a deep CNN, then we already have good high-level visual features, whereas the text will need much more processing before the model has access to high-level text features. Moreover, the researchers hypothesized that “image regions may have weaker relations than words in a sentence”.[25] Lastly, BERT was initially pretrained using text only, so forcing it to process other modalities may give suboptimal results and even damage its weights during multimodal training.

So the authors chose a dual-stream approach instead: each modality goes through its own encoder, and in the upper layers the two encoders are connected and exchange information through a new bidirectional cross-attention mechanism called *co-attention* (see Figure 16-11). Specifically, in each pair of connected encoder layers, the MHA query of one encoder is used as the MHA key/value by the other encoder.

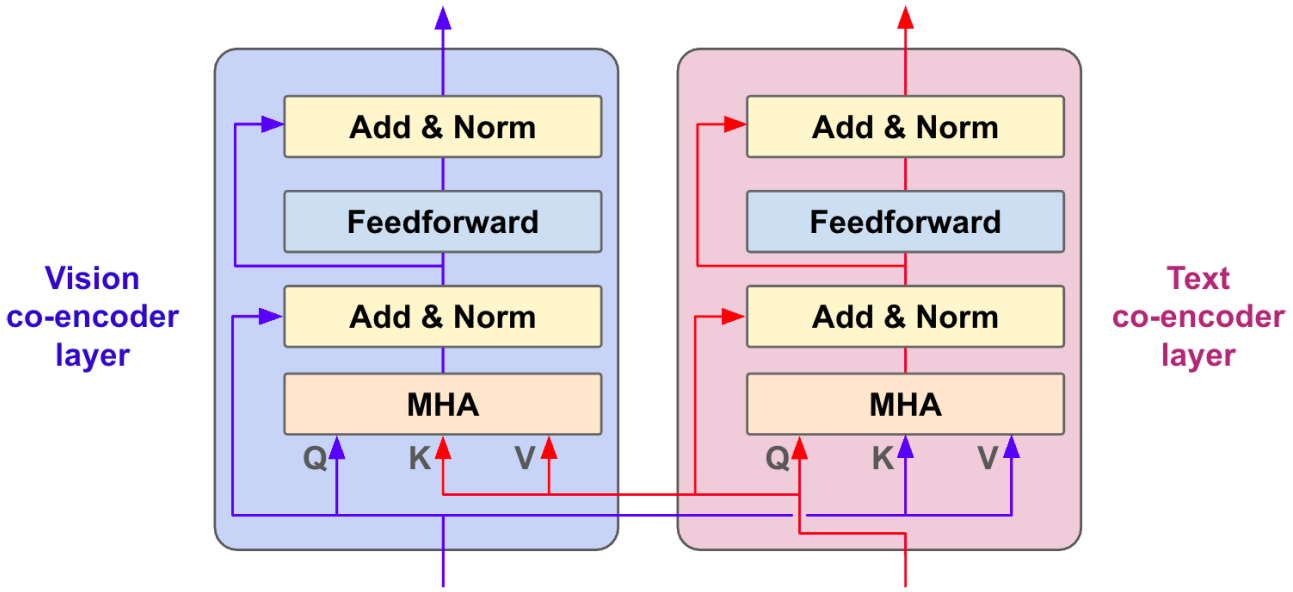


Figure 16-11. Two encoder layers connected through co-attention: the MHA query of one encoder is used as the MHA key/value by the other encoder

The lower layers of the text encoder are initialized with BERT's weights (the authors used BERT base, which has 12 layers), and 6 co-attention layers sit on top (see the lower-right quadrant of Figure 16-12). The visual features are produced by a pretrained and frozen Faster R-CNN model, and it is assumed that these features are sufficiently high level so no further processing is needed, therefore the visual encoder is exclusively composed of 6 co-attention layers, paired up with the text encoder's 6 co-attention layers (see the lower-left quadrant of the figure). The Faster R-CNN model's outputs go through a mean pooling layer for each detected region, so we get one feature vector per region, and low-confidence regions are dropped: each image ends up represented by 10 to 36 vectors.

Since regions don't have a natural order like words do, the visual encoder does not use positional encoding. Instead, it uses spatial encodings which are computed like this: each region's bounding box is encoded as a 5D vector containing the normalized upper-left and lower-right coordinates, and the ratio of the image covered by the bounding box. Then this 5D vector is linearly projected up to the same dimensionality as the visual vector, and simply added to it.

Lastly, a special [IMG] token is prepended to the visual sequence: it serves the same purpose as the class token (i.e., to produce a representation of the whole sequence), but instead of being a trainable embedding, it's computed as the average of the feature vectors (before spatial encoding), plus the spatial encoding for a bounding box covering the whole image.

Now on to training! Similar to VideoBERT, the authors used masked token prediction and linguistic-visual alignment:

- For masked token prediction, the authors used regular BERT-like MLM for the text encoder. However, for the visual encoder, since ViLBERT does not use a fixed-size visual vocabulary (there's no clustering step), the model is trained to predict the class distribution that the CNN predicts for the given image region (this is a soft target). The authors chose this task rather than predicting raw pixels because the regions can be quite large and there's typically not enough information in the surrounding regions and in the text to reconstruct the

masked region correctly: it's better to aim for a higher-level target.

- For linguistic-visual alignment, the model takes the outputs of the [IMG] and [CLS] tokens, then computes their item-wise product and passes the result to a binary classification head that must predict whether the text and image match. Multiplication is preferred over addition because it amplifies features that are strong in both representations (a bit like a logical AND gate), so it better captures alignment.

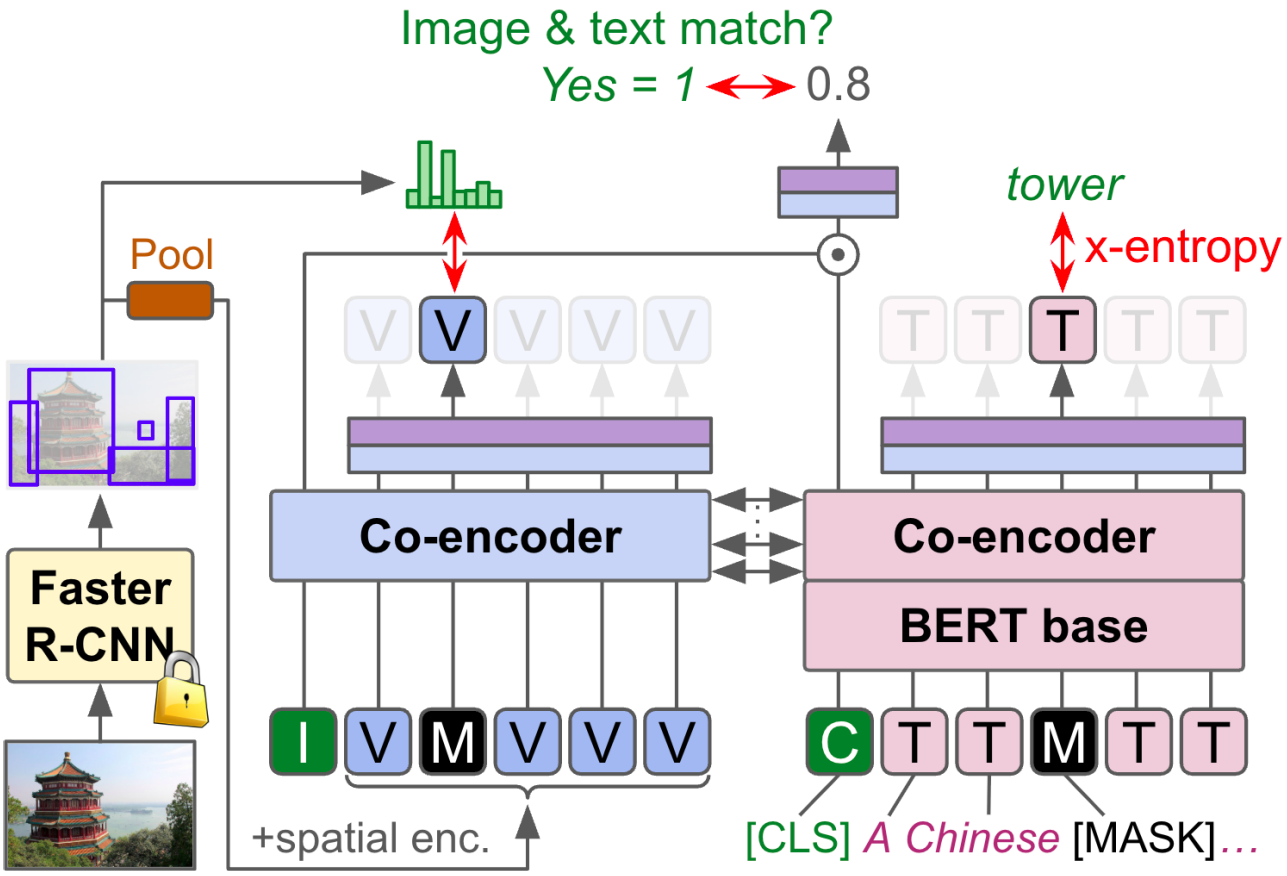


Figure 16-12. ViLBert pretraining using masked token prediction and linguistic-visual alignment (again, shown together but actually separate)

And that's it. This model significantly beat the state-of-the-art for several downstream tasks, including image grounding, caption-based image retrieval (even zero-shot), visual question answering, and *visual commonsense reasoning* (VCR) which involves answering a multiple-choice question about an image (like VQA), then selecting the appropriate justification. For example, given an image of a waiter serving some pancakes at a table, along with the question "Why is person #4 pointing at person #1", the model must choose the correct answer "He is telling person #3 that person #1 ordered the pancakes", then it must choose the justification "Person #3 is serving food and she might not know whose order is whose".

ViLBERT had a strong influence on the field of multimodal machine learning thanks to its dual stream architecture, the invention of co-attention, and its excellent results on many downstream tasks: it was another great demonstration of the power of large-scale self-supervised pretraining using transformers. The next major milestone came in 2021, and it approached the problem very differently, using contrastive pretraining: meet CLIP.

CLIP: a Dual-Encoder Text + Image Model Trained With Contrastive Pretraining

OpenAI's January 2021 release of CLIP[26] was a major breakthrough, not just for its astounding capabilities, but also because of its surprisingly straightforward approach based on *contrastive learning*: the model learns to encode text and images into vector representations that are similar when the text and image match, and dissimilar when they don't match.

Once trained, the model can be used for many tasks, particularly zero-shot image classification. For example, CLIP can be used as an insect classifier, without any additional training: just start by feeding all the possible class names to CLIP, such as "cricket", "ladybug", "spider", and so on, to get one vector representation for each class name. Then, whenever you want to classify an image, feed it to CLIP to get a vector representation, and find the most similar class name representation using cosine similarity. This usually works even better if the text resembles typical image captions found on the web, since this is what CLIP was trained on, for example "This is a photo of a ladybug." instead of just "ladybug". A bit of prompt engineering can help (i.e., experimenting with various prompt templates).

The good news is that CLIP is fully open source[27], several pretrained models are available on the Hugging Face Hub, and the Transformers library provides a convenient pipeline for zero-shot image classification:

```
from transformers import pipeline

model_id = "openai/clip-vit-base-patch32"
clip_pipeline = pipeline(task="zero-shot-image-classification", model=model_id,
                         device_map="auto", torch_dtype="auto")
candidate_labels = ["cricket", "ladybug", "spider"]
image_url = "https://homl.info/ladybug"  # a photo of a ladybug on a dandelion
results = clip_pipeline(image_url, candidate_labels=candidate_labels,
                        hypothesis_template="This is a photo of a {}.")
```

Note that we provided a prompt template, so the model will actually encode "This is a photo of a ladybug.", not just "ladybug" (if you don't provide any template, the pipeline actually defaults to "This is a photo of {}."). Now let's look at the results, which are sorted by score:

```
[{'score': 0.9972853660583496, 'label': 'ladybug'},
 {'score': 0.001651176018640399, 'label': 'spider'},
 {'score': 0.0010634493082761765, 'label': 'cricket'}]
```

Great! CLIP predicts ladybug with over 99.7% confidence. Now if you want a flower classifier instead, just replace the candidate labels with names of flowers. If you include "dandelion" in the list and classify the same image, the model should choose "dandelion" with high confidence (ignoring the ladybug). Impressive!

So how does this magic work? Well, CLIP's architecture is based on a regular text encoder and a regular vision encoder, no co-attention or anything fancy (see Figure 16-13). You can actually use pretty much any text and vision encoders you want, as long as they can produce a vector representation of the text or image. The authors experimented with various encoders, includ

representation of the text or image. The authors experimented with various encoders, including several ResNet and ViT models for vision, and a GPT-2-like model for text, all trained from scratch. What's that I hear you say, GPT-2 is not an encoder? That's true, it's a decoder-only model, but we're not pretraining it for next token prediction, so the last token's output is free to be used as a representation of the entire input sequence, which is what CLIP does. You may wonder why we're not using a regular text encoder like BERT? Well, we could, but OpenAI created GPT—Alex Radford is the lead author of both GPT and CLIP—so that's most likely why GPT-2 was chosen: the authors simply had more experience with this model and a good training infrastructure already in place. Using a causal encoder also makes it possible to cache the intermediate state of the model when multiple texts start in the same way, for example "This is a photo of a".

Also note that a pooling layer is added on top of the vision encoder to ensure it outputs a single vector for the whole image, instead of feature maps. Moreover, a linear layer is added on top of each encoder to project the final representation into the same output space (i.e., with the same number of dimensions). So given a batch of m image-caption pairs, we get m vector representations for the images and m vector representations for the captions, and all vectors have the same number of dimensions. Figure 16-13 shows $m = 4$, but the authors used a shockingly large batch size of $m = 2^{15} = 32{,}768$ during training.

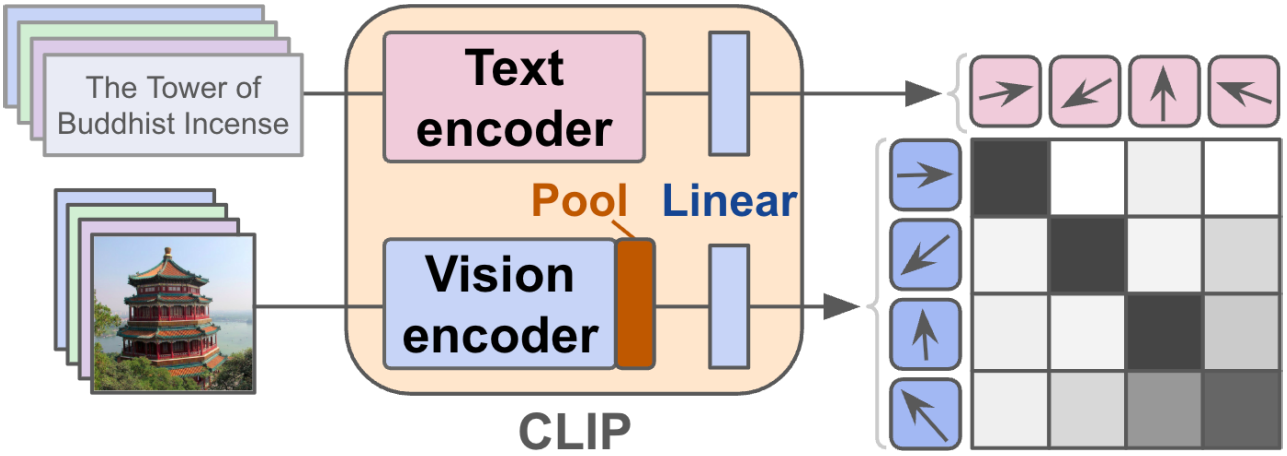


Figure 16-13. CLIP: a batch of image-caption pairs is encoded as vectors, then matching pairs are pulled closer while mismatched pairs are pushed away

The model was then pretrained on a large dataset of 400 million image-caption pairs scraped from the internet, using a contrastive loss[28] that pulls together the representations of matching pairs, while also pushing apart representations of mismatched pairs. Here's how it works:

- All vectors are first ℓ_2 normalized, meaning they are rescaled to unit vectors: we only care about their orientation, not their length.
- Next, we compute the cosine similarity of the image representation and the text representation for every possible image-caption pair. The result is an $m \times m$ matrix containing numbers between –1 for opposite vectors, and +1 for identical vectors. In Figure 16-13, this matrix is represented by the 4 × 4 grid (black is +1, white is –1). Each column measures how much each image in the batch matches a given caption in the same batch, while each row measures how much each caption matches a given image.
- Since the i^{th} image corresponds to the i^{th} caption, we want the main diagonal of this matrix to contain similarity scores close to +1, while all other scores should be close to 0. Why not

close to –1? Well, if an image and a text are totally unrelated, we can think of their representations as two random vectors. Recall that two random high-dimensional vectors are highly likely to be close to orthogonal (as discussed in Chapter 7) so their cosine similarity will be close to 0, not –1. In other words, it makes sense to assume that the text and image representations of a mismatched pair are unrelated (score close to 0), not opposite (score close to –1).

- In the i^{th} row, we know that the matching caption is in the i^{th} column, so we want the model to produce a high similarity score in that column, and a low score elsewhere. This resembles a classification task where the target class is the i^{th} class. Indeed, we can treat each similarity score as class logit and simply compute the cross-entropy loss for that row with i as the target. We can follow the exact same rationale for each column. If we compute the cross-entropy loss for each row and each column (using class i as the target for the i^{th} row and the i^{th} column), and evaluate the mean, we get the final loss.
- There's just one extra technical detail: the similarity scores range between –1 and +1, which is unlikely to be the ideal logit scale for the task, so CLIP divides all the similarity scores by a trainable temperature (a scalar) before computing the loss.

WARNING

This loss requires a large batch size to ensure the model sees enough negative examples to contrast with the positive examples, or else it could overfit details in the positive examples. CLIP's success is due in part to the gigantic batch size that the authors were able to implement.

The authors evaluated CLIP on many image classification datasets, and for roughly 60% of these, it performed better without any extra training (i.e., zero-shot) than a *linear probe* trained on ResNet-50 features (that's a linear classifier trained on features output by a pretrained and frozen ResNet-50 model), including on ImageNet, despite the fact that the ResNet-50 model was actually pretrained on ImageNet. CLIP is particularly strong on datasets with few examples per class, with pictures of everyday scenes (i.e., the kind of pictures you find on the web). In fact, CLIP even beat the state-of-the-art on the Stanford Cars dataset, ahead of the best ViTs specifically trained on this dataset, because pictures of cars are very common on the Web and the dataset doesn't have many examples per class. However, CLIP doesn't perform as well on domain-specific images, such as satellite or medical images.

Importantly, the visual features output by CLIP are also highly robust to perturbations, making them excellent for downstream tasks, such as image retrieval: if you store images in a vector database, indexing them by their CLIP-encoded visual features, you can then search for them given either a text query or an image query. For this, just run the query through CLIP to get a vector representation, then search the database for images with a similar representation.

To get the text and visual features using the Transformers library, you must run the CLIP model directly, without going through a pipeline:

```
import PIL
import urllib.request
```

```
from transformers import CLIPProcessor, CLIPModel

clip_processor = CLIPProcessor.from_pretrained(model_id)
clip_model = CLIPModel.from_pretrained(model_id)
image = PIL.Image.open(urllib.request.urlopen(image_url)).convert("RGB")
captions = [f"This is a photo of a {label}." for label in candidate_labels]
inputs = clip_processor(text=captions, images=[image], return_tensors="pt",
                        padding=True)
with torch.no_grad():
    outputs = clip_model(**inputs)

text_features = outputs.text_embeds    # shape [3, 512]  # 3 captions
image_features = outputs.image_embeds  # shape [1, 512]  # 1 image (ladybug)
```

TIP

If you need to encode the images and text separately, you can use the CLIP model's `get_image_features()` and `get_text_features()` methods. You must first tokenize the text using a `CLIPTokenizer` and process the images using a `CLIPImageProcessor`, and the resulting features are not ℓ_2 normalized, so you must divide them by `features.norm(dim=1, keepdim=True)` (see the notebook for a code example).

The features are already ℓ_2 normalized, so if you want to compute similarity scores, a single matrix multiplication is all you need:

```
>>> similarities = image_features @ text_features.T  # shape [1, 3]
>>> similarities
tensor([[0.2337, 0.3021, 0.2381]])
```

This works because matrix multiplication computes the dot products of every row vector in the first matrix with every column vector in the second, and each dot product is equal to the cosine of the angle between the vectors multiplied by the norms of the vectors. Since the vectors have been ℓ_2 normalized in this case, the norms are equal to 1, so the result is just the cosine of the angle, which is the similarity score we're after. As you can see, the most similar representation is the second one, for the ladybug class. If you prefer estimated probabilities rather than similarity scores, you must first rescale the similarities using the model's learned temperature, then pass the result through the softmax function (it's nice to see that we get the same result as the pipeline):

```
>>> temperature = clip_model.logit_scale.detach().exp()
>>> rescaled_similarities = similarities * temperature
>>> probabilities = torch.nn.functional.softmax(rescaled_similarities , dim=1)
>>> probabilities
tensor([[0.0011, 0.9973, 0.0017]])
```

CLIP wasn't the only surprise OpenAI had in stock in 2021: just the following month, they an

CLIP wasn't the only surprise OpenAI had in stock in 2021. Just the following month, they announced DALL·E, which can generate impressive images given a text description. Let's discuss it now.

DALL·E: Generating Images from Text Prompts

OpenAI DALL·E[29], released in February 2021, is a model capable of generating images based on text prompts such as "an armchair in the shape of an avocado". It's architecture is quite simple (see the lefthand side of Figure 16-14): a GPT-like model trained to predict the next token, but unlike GPT, it was pretrained on millions of image-caption pairs, and fed input sequences composed on text tokens followed by visual tokens. At inference time, you only feed it the text tokens, and the model then generates the visual tokens, one at a time, until you get the full image. The visual tokens are generated by a dVAE model which takes an image and outputs a sequence of tokens from a fixed vocabulary. Sadly, the model was never released to the public, but the paper was detailed enough, so some open source replications are available, such as DALL·E mini, also known as Craiyon.

One year later, in April 2022, OpenAI released DALL·E 2,[30] able to generate even higher quality images. Its architecture is actually very different: the text is fed to a CLIP model which outputs a text embedding, then this text embedding is fed to a *diffusion model* which uses it to guide its image generation process (we will discuss diffusion models in Chapter 18). The model is not open source, but it's available through a paid API, and via some products such as Microsoft Designer, Bing Image Creator, Canva, ChatGPT, and more.

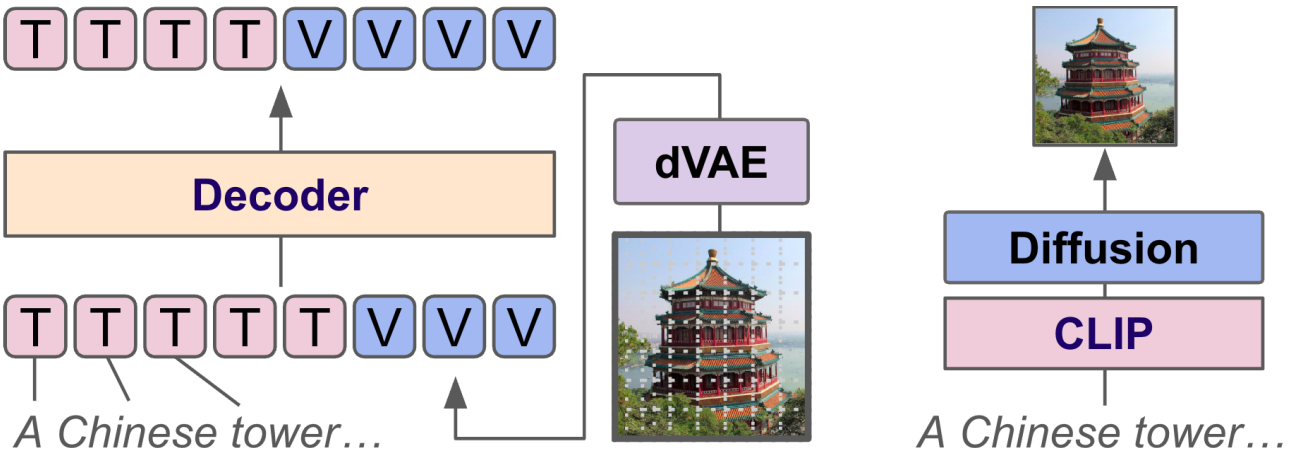


Figure 16-14. DALL·E (left) and DALL·E 2 (right)

DALL·E 3 was released in October 2023. Sadly, by then OpenAI had fully shifted away from their initial openness: there was no peer-reviewed paper, no code, no weights, no data. Like the previous version, it's available through an API and via some products. We know it's diffusion-based, it doesn't use CLIP, and it's tightly integrated with GPT-4, which rewrites the prompt before generating the image. It works impressively well: it outputs stunning images which match the prompts much more precisely than previous versions. The difference is particularly striking for *compositional prompts* (e.g., "A fluffy white cat sitting on a red velvet cushion, with a vase of sunflowers behind it, bathed in golden hour light. The cat is looking directly at the viewer."): DALL·E 1 and 2 would generally follow only one or two elements of such prompts, whereas DALL·E 3 follows instructions much more closely. The image quality, real-

ism, artistic style and consistency, are astounding. Lastly, DALL·E 3 also integrates some moderation capabilities.

The next landmark in our multimodal journey came one month after the first DALL·E model: the Perceiver.

Perceiver: Bridging High-Resolution Modalities with Latent Spaces

Every transformer so far has required chopping the inputs into meaningful tokens. In the case of text, tokens represent words or subwords. In the case of ViTs, they represent 16 × 16 pixel patches. In VideoBERT, it's short 1.5s clips. In audio transformers, it's short audio clips. If we fed individual characters, pixels, or audio frames, directly into a transformer, the input sequence would be extremely long, and we would run into the quadratic attention problem. Also, we would lose important inductive biases: for example, by chopping an image into patches, we enforce a strong inductive bias toward proximity (i.e., nearby pixels are assumed to be more strongly correlated than distant pixels).

However, such tokenization is modality-specific, which makes it harder to deal with new modalities or mix them in the model. Moreover, inductive biases are great when you don't have a lot of training data (assuming the biases are correct), but if your dataset is large, you will often get better performance by using unbiased models, with very few implicit assumptions: sure, the model will have to figure out on its own that nearby pixels are generally related, but on the other hand it will be flexible enough to discover patterns that might otherwise go unnoticed.

This is why DeepMind introduced the *Perceiver*,[31] in March 2021. This architecture is capable of directly handling any modality at the lowest level: characters, pixels, audio frames, and more. Moreover, it does so with a modality-agnostic design, so the same model can handle different modalities. The Perceiver architecture is shown in Figure 16-15.

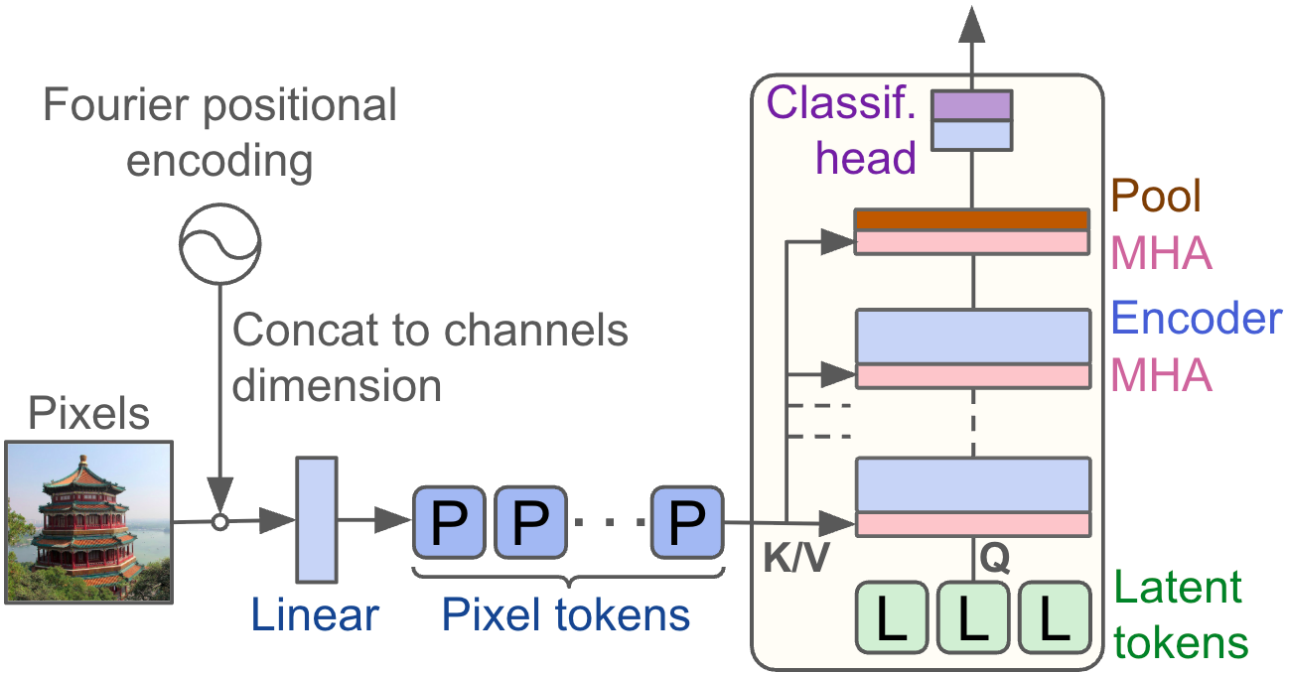


Figure 16-15. Perceiver architecture: inputs are ingested through cross-attention layers while the main input is a sequence of

learned latent tokens

Let's walk through this architecture:

- The input is first chopped into its smallest constituents. In this example, the input is an image, so it is chopped into individual pixels: we now have a sequence of 3D vectors (red, green, blue).
- Positional encodings are concatenated to these feature vectors: Perceiver uses Fourier positional encodings, which are very similar to the sinusoidal positional encodings of the original Transformer, except they encode all of the input's dimensions. Since an image is 2D, each pixel's horizontal and vertical coordinates are encoded: for example, if a pixel is located at coordinates x and y (normalized between –1 and 1), then the positional encoding vector will include x and y, followed by sin(π_fx_), sin(π_fy_), cos(π_fx_), cos(π_fy_) repeated K times (typically 6) with the frequency f starting at 1 and going up to $\mu / 2$ (spaced equally), where μ is the target resolution (e.g., if the image is 224 × 224 pixels, then $\mu = 224$).[32] The dimensionality of the positional encoding vector is d(2_K_ + 1), where d is the number of input dimensions (i.e., 1 for audio, 2 for images, 3 for videos, etc.).
- The pixel tokens now have 3 + 2 × (2 × 6 + 1) = 29 dimensions. We then pass them through a linear layer to project them to the Perceiver's dimensionality (e.g., 512).
- The Perceiver's architecture itself is composed of repeated processing blocks (e.g., 8), where each block is composed of a single cross-attention multi-head attention layer (MHA) followed by a regular transformer encoder (e.g., with 6 encoder layers). The final block is composed of a single cross-attention MHA layer and an average pooling layer to reduce the input sequence into a single vector, which is then fed to a classification head (i.e., linear + softmax).
- The pixel tokens are fed to the Perceiver exclusively through the MHA layers, and they play the role of the keys and values. In other words, the Perceiver attends to the pixel tokens through cross-attention only.
- Crucically, the Perceiver's main input is a fairly short sequence of *latent tokens* (e.g., 512). These tokens are similar to an RNN's hidden state: an initial sequence (learned during training) is fed to the Perceiver, and it gradually gets updated as the model learns more and more about the pixel tokens via cross-attention. Since it's a short sequence, it doesn't suffer much from the quadratic attention problem: this is called the *latent bottleneck trick*, and is the key to the success of the Perceiver.
- The authors experimented sharing weights across processing blocks (excluding the first cross-attention layer), and they got good results. When the processing blocks share the same weights, the Perceiver is effectively a recurrent neural network, and the latent tokens really are its hidden state.

NOTE

As we saw in Chapter 7, the manifold hypothesis states that most real world data lives near a low-dimensional manifold, much like a rolled piece of paper lives in 3D but is essentially a 2D object. This 2D space is latent (i.e., hidden, potential) until we unroll the paper. Similarly, the Perceiver's goal is to "unroll" its high-dimensional inputs so the model can work in the latent space, using low-dimensional representations.

Importantly, this architecture can efficiently process high-resolution inputs. For example, a 224 × 224 image has 50,176 pixels, so if we tried to feed such a long sequence of pixel tokens directly to a regular encoder, each self-attention layer would have to compute $50{,}176^2 \approx 2.5$ billion attention scores! But since the Perceiver only attends to the pixel tokens through cross-attention, it just needs to compute 50,176 times the number of latent tokens. Even for the biggest Perceiver variant, that's just a total of $50{,}176 \times 512 \approx 25.7$ million attention scores, which is roughly 100 times less compute.

NOTE

Thanks to the latent bottleneck, the Perceiver scales linearly with the number of pixel token, instead of quadratically.

The authors trained the Perceiver using regular supervised learning on various classification tasks across several modalities, including image-only (ImageNet), audio + video (AudioSet),[33] or point clouds (ModelNet40),[34] all using the same model architecture: they got competitive results, in some cases even reaching the state-of-the-art.

The videos in the AudioSet dataset were downsampled to 224 × 224 pixels at 25 frames per second (fps), with a 48 kHz audio sample rate. You could theoretically feed each pixel and each audio frame individually to the Perceiver, but this would be a bit extreme, as each 10s video would be represented as a sequence of $224 \times 224 \times 25 \times 10 \approx 12.5$ million pixel tokens, and 48,000 × 10 = 480,000 audio tokens.

So the authors had to compromise: they trained on 32-frame clips (at 25 fps, that's 1.28s each, instead of 10s) and they chopped the video into 2 × 8 × 8 patches (i.e., 2 frames × 8 × 8 pixels), resulting in 224 × 224 × 32 / (2 × 8 × 8) = 12,544 video tokens of 128 RGB pixels each (plus the position encoding). They also chopped the audio into clips of 128 frames each, resulting in 480 audio tokens. They also tried converting the audio to a mel spectrogram (which resulted in 4,800 audio tokens): using a spectrogram instead of raw audio is a standard practice in audio processing, but it made very little difference to the model's performance, which shows that the Perceiver is able to extract useful features from the raw data without any help.

Then they simply concatenated the video and audio token sequences (after positional encoding), and also concatenated a modality embedding to help the model distinguish the modalities.

One limitation of the Perceiver architecture is that it was only designed for multimodal classification. That said, instead of averaging the latent tokens and feeding them to a classification head, we could try to use them for other downstream tasks. Of course, the DeepMind researchers thought of that, and just a few months later they published the Perceiver IO architecture.

Perceiver IO: a Flexible Output Mechanism for the Perceiver

DeepMind released Perceiver IO in July 2021:[35] it can perform classification tasks like the Perceiver, but also many other tasks such as masked language modeling (MLM) better than BERT, *optical flow* (i.e., predicting where each pixel will move in the next video frame), actually beating the state of the art, and even playing StarCraft II.

The model is identical to Perceiver up to the output latent tokens, but the pooling layer and the classification head are replaced by a very flexible output mechanism (see Figure 16-16):

- A new cross-attention layer is added, which acts as a decoder by attending to the output latent tokens and produceing the final output representations. These output representations can then go through a task-specific head, or even multiple heads if we're doing multi-task learning.
- The number and nature of the output tokens is task-specific:
  - For classification, we only need one output vector, which we can feed to a classification head. Therefore, we need one output query token, which can just be a learned embedding.
  - For masked language modeling, we can use one output query token per masked token, and add a classification head on top of the output representations (i.e., linear + softmax) to get one estimated token probabilities for each masked token. To help the model locate each masked token, the output query tokens are learnable positional embeddings based on the masked token's position. For example, given the masked sentence "The dog [MASK] the [MASK]", the masked tokens are located at positions #2 and #4, so we use the positional embedding #2 as the first output query token, and #4 as the second output query token. This same approach works for any other modality: just predict the masked tokens. It can also be extended to multiple modalities at once, typically by adding a modality embedding to the output query token before feeding it to the output cross-attention layer.
  - For optical flow, the authors actually used one output token per pixel, using the same pixel representations both as the inputs to the Perceiver and as the output query tokens. This representation includes a Fourier positional encoding.

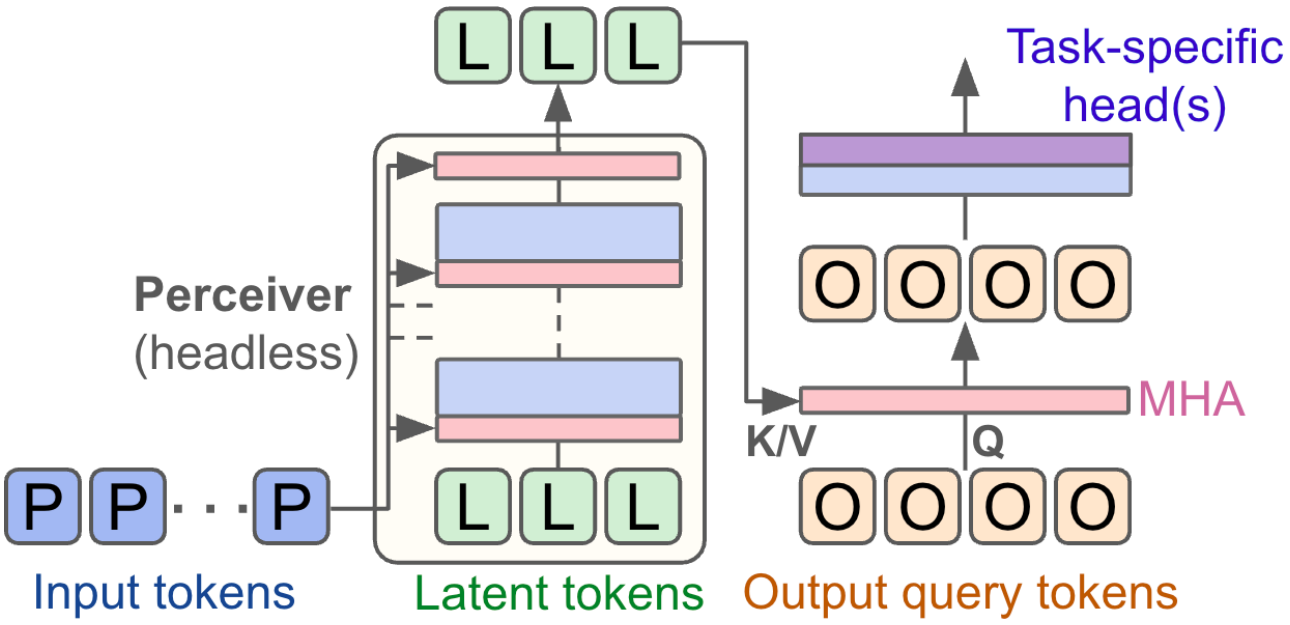


Figure 16-16. Perceiver IO architecture: one output query token per desired output token is fed to a cross-attention layer which

attends to the Perceiver's output latent tokens

NOTE

Because the output query tokens only ever attend to the latent tokens, the Perceiver IO can handle a very large number of output query tokens. The latent bottleneck allows the model to scale linearly for both the inputs and outputs.

The Perceiver IO is a bidirectional architecture, there's no causal masking, so it's not well suited for autoregressive tasks. In particular, it cannot efficiently perform next token prediction, so it's not well suited for text generation tasks such as image captioning. Sure, you could feed it an image and some text with a mask token at the end, and make it predict which token was masked, then start over to get the next token, and so on, but it would be horribly inefficient compared to a causal model (which can cache the previous state).

For this reason, Google and DeepMind researchers released the Perceiver AR architecture in February 2022 to address this limitation (AR stands for autoregressive). The model works very much like the Perceiver, except the last tokens of the input sequence are used as the latent tokens, the model is causal over these latent tokens, and it is trained using next token prediction. Perceiver AR didn't quite have the same impact as Perceiver and Perceiver IO, but it got excellent results on very long input sequences, thanks to its linear scaling capability.

But DeepMind wasn't done with multimodal ML: they soon released yet another amazing multimodal model, partly based on the Perceiver: Flamingo.

Flamingo: Open-Ended Visual Dialogue

DeepMind's Flamingo paper, published in April 2022, introduced a visual-language model (VLM) that can take arbitrary sequences of text and images as input and generate coherent free-form text. Most importantly, its few-shot performance is excellent on a wide variety of tasks.

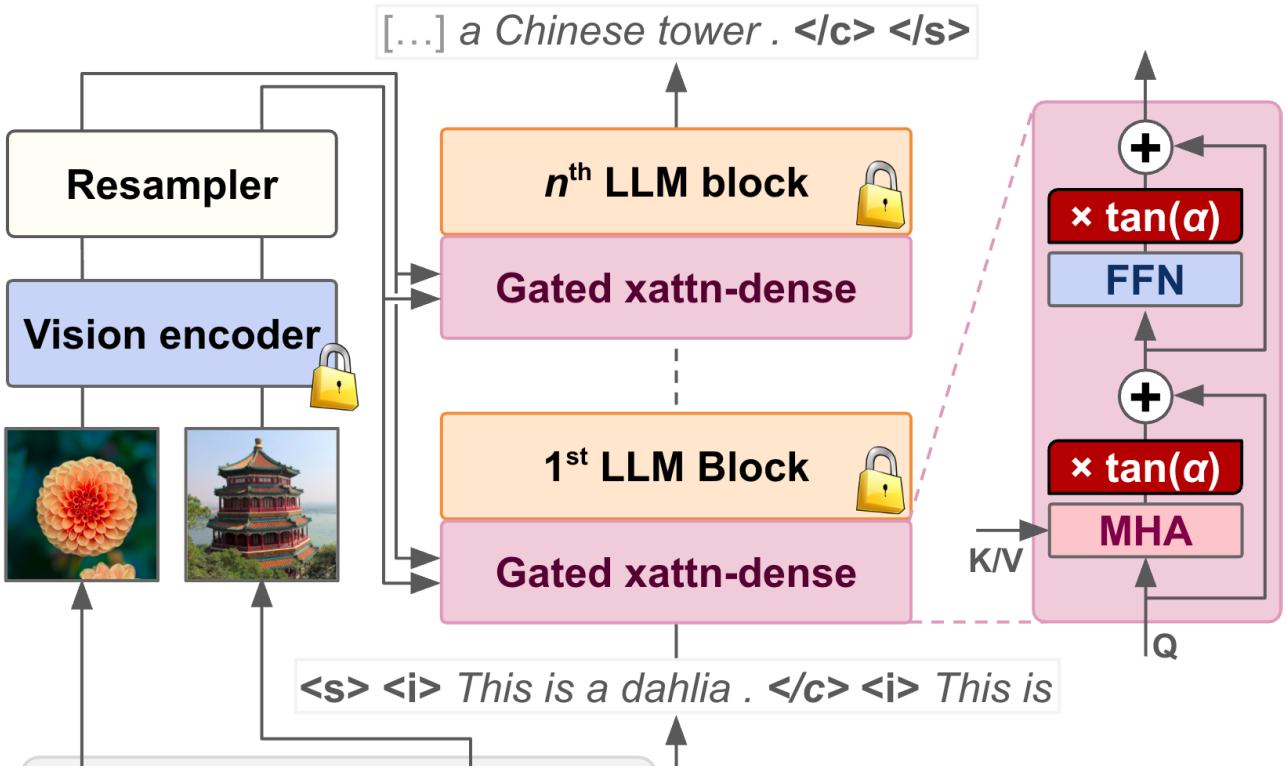
For example, suppose you want to build a model that takes a picture and outputs a poem about that image: no need to train a new model, you can just feed a few examples to Flamingo, add the new image at the end, and it will happily generate a poem about this new image. If you want it to detect license plate numbers on car photos, just give it a few photos along with the corresponding license plate numbers (as text), then add a new car photo, and Flamingo will output its license plate number. You can just as easily use Flamingo for image captioning. Or visual question answering. Or you can ask it to compare two images. In fact, you can even give the model several frames from a video and ask it to describe the action. It's an incredibly versatile and powerful model out-of-the-box, without any fine-tuning.

Let's look at Flamingo's architecture (see Figure 16-17):

- Instead of starting from scratch, Flamingo is based on two large pretrained models, which

are both frozen: a vision model and a decoder-only language model. The authors used Chinchilla and CLIP respectively, but many other powerful models would work fine too.

- Each input image is fed to the vision model, and the outputs go through a Perceiver model, called a *Resampler*, which produces a sequence of latent token representations: this ensures that every image gets represented as a fairly short sequence of latent representations (typically much shorter than the output of the vision model). This works around the quadratic attention problem.
- The sequences output by the Resampler are fed as the keys/values to many *gated xattn-dense* modules, which are inserted before every block in the frozen LLM:
  - Each gated xattn-dense module is composed of a masked multi-head attention layer followed by a feedforward module, with a skip connection each, just like the cross-attention half of a vanilla Transformer's decoder layer.
  - However, both the masked MHA layer and the feedforward module are followed by a *tanh gate*: these gates multiply their input by tanh(α), where α is a learnable scalar parameter initialized to 0 (one per gate). Since tan(0) = 0, training starts with all gates closed, so the inputs can only flow through the skip connections, and the gated xattn-dense modules have no impact on the LLM. But as training progresses, the model gradually learns to open the gates, allowing the gated modules to influence the LLM's outputs.
  - In the gated xattn-dense module, each text token can only attend to visual tokens from the closest image located before it: visual tokens from all other images are masked. For example, the last text token ("is") can only attend to the Chinese tower photo, it cannot directly attend to the flower photo. However, since previous text tokens have information about the flower photo, the last token does have indirect access to the flower photo via the frozen LLM's self-attention layers.
- The text is tokenized as the LLM expects (e.g., Chinchilla expects start-of-sequence and end-of-sequence tokens, which I noted <s> and </s>), but a couple new special tokens are added: each image-text chunk ends with an end-of-chunk token (which I noted </c>), and each image is replaced with an image token (which I noted <i>). Both are represented using trainable embeddings.



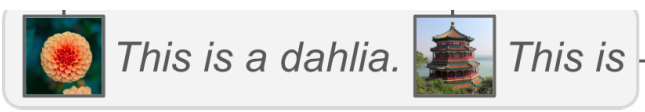


Figure 16-17. Flamingo takes any sequence of text and images and outputs coherent free-form text

The bad news is that DeepMind did not release Flamingo to the public. The good news is that open source replications and variants are available:

- OpenFlamingo, created by the MLFoundations team, which is part of the non-profit organization LAION. It is fully open source and available on the Hugging Face Hub (e.g., openflamingo/OpenFlamingo-9B-vitl-mpt7b, based on a CLIP ViT-L/14 vision encoder and a MPT-7B LLM).
- IDEFICS by Hugging Face, trained on a huge dataset named OBELICS,[36] composed of 141 million interleaved text-image documents gathered from Common Crawl (including 350 million images, and 115 billion text tokens). Both IDEFICS and OBELICS are available on the hub (e.g., Idefics3-8B-Llama3 and OBELICS by HuggingFaceM4). The architecture includes a few improvements over Flamingo, for example you can more easily swap in different LLMs or vision encoders. IDEFICS itself is open source, but the models it is based on may have licensing limitations. In particular, IDEFICS 1 and 3 are based on Llama, which has some limitations for commercial use, while IDEFICS 2 is based on Mistral, which is fully open source.
- AudioFlamingo by Nvidia, which is very similar to Flamingo but handles audio instead of images.
- Other variants are available, such as domain-specific models like Med-Flamingo, an OpenFlamingo model trained on medical documents.

The last multimodal architecture we will discuss is BLIP by Salesforce. Its second version, BLIP-2, also successfully reuses two large pretrained models—a vision model and an LLM—to create a VLM that can ingest both images and text and generate free-form text. Let's see how.

BLIP and BLIP-2

The original BLIP model is an excellent visual-language model released by Salesforce in January 2022.[37] Its architecture is a *mixture of encoder-decoder* (MED) composed of a text-only encoder, a vision-only encoder, an image-grounded text encoder, and an image-grounded text decoder, sharing many layers. This flexible architecture made it possible to train the model simultaneously on three distinct objectives: *image-text matching* (ITM), an *image-text contrastive* (ITC) loss to align image and text representations (similar to CLIP), and finally language modeling (LM) where the model must try to generate the caption using next token prediction.

Another important reason for BLIP's success is the fact that it was pretrained on a very large and clean dataset. To build this dataset, the authors simultaneously trained a *captioning module* to generate synthetic captions for images, and a *filtering module* to remove noisy data. This approach, named *CapFilt*, removed poor quality captions from the original web-scraped dataset, and added many new high-quality synthetic captions. After this bootstrapping stage, the authors trained the final model on the large and clean dataset they had just built. It's a two-stage process, hence the name BLIP: *bootstrapping language-image pretraining*.

One year later, in January 2023, Salesforce released BLIP-2[38], which is based on the same core ideas but greatly improves the model's performance by reusing two large pretrained models, one vision model and one language model, both frozen. BLIP-2 even outperformed Flamingo with a much smaller model.

Training is split in two stages. BLIP-2's architecture during the first stage is shown in Figure 16-18:

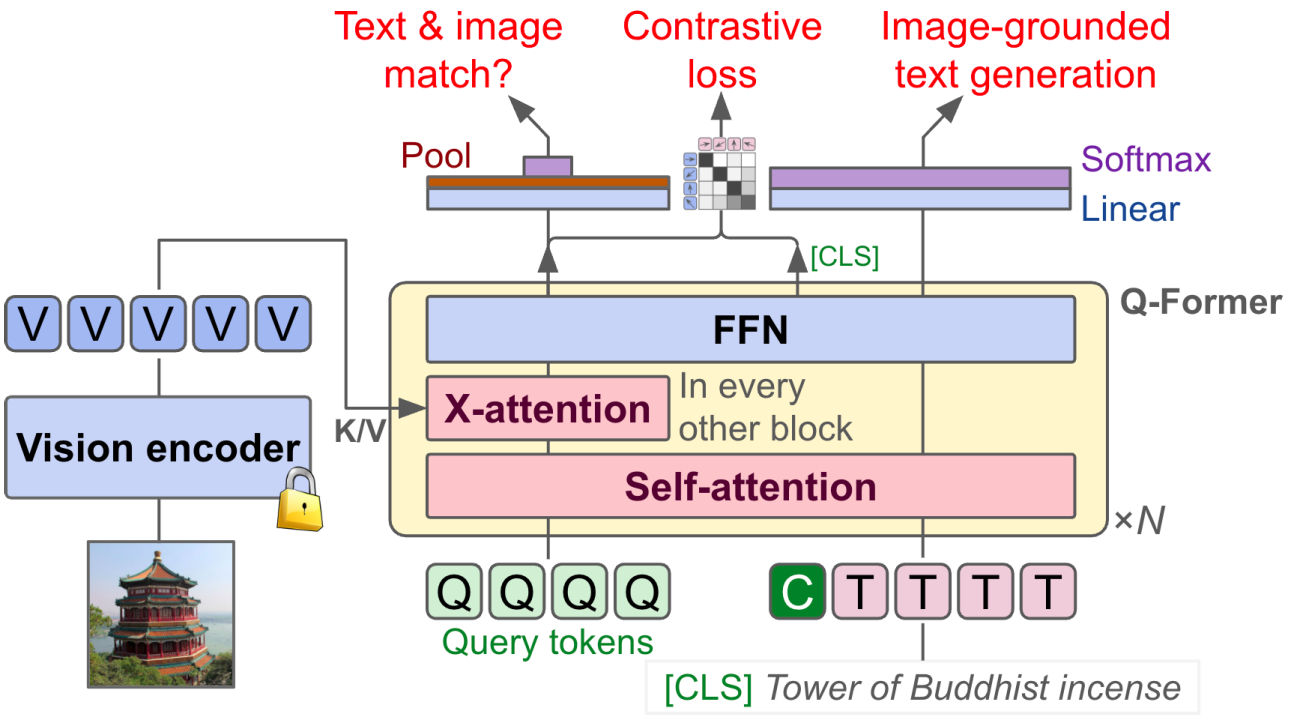


Figure 16-18. BLIP-2 Pretraining – Stage 1: training the Q-Former

- The central component is called the *Q-Former* (querying transformer). Its architecture is the same as BERT-base, and in fact it's even initialized using BERT-base's pretrained weights, but it also has some extra cross-attention layers that let it attend to visual tokens produced by the pretrained visual encoder. The cross-attention layers are inserted in every other encoder layer, between the self-attention layer and the feedforward module, and they are initialized randomly.
- The Q-Former processes three sequences: a sequence of text tokens (using BERT tokenization and token embeddings), a sequence of visual tokens produced by the pretrained vision encoder, and lastly a sequence of trainable Perceiver-style latent tokens. In BLIP-2, the latent tokens are called *query tokens* because their output representations will later be used to query the pretrained LLM.
- The Q-Former is trained with the same three objectives as BLIP: ITM, ITC, and LM. For each objective, a different mask is used:
  - For ITM, query tokens and text tokens can attend to each other. In other words, the output representations for the query tokens represent text-grounded visual features, and the output representations for the text tokens represent image-grounded text features. The query token outputs go through a linear layer which produces 2 logits per query token (image-text match or mismatch?), and the model computes the mean logits across all query tokens, then computes the binary cross-entropy.
  - For ITC, query tokens and text tokens cannot attend to each other. In other words, the Q-

Former's outputs represent visual-only features and text-only features. For each possible image/caption pair in the batch, the model computes the max similarity between the query token outputs and the class token output. We get a matrix of max similarities, and the loss pushes the values toward +1 on the main diagonal, and pushes the other values toward 0, much like CLIP.

– For LM, text tokens can only attend previous tokens (i.e., we use a causal mask), but they can attend all query tokens. However, query tokens cannot attend any text token. In other words, the query token outputs represent visual-only features, while text token outputs represent image-grounded causal text features. The model is trained using next token prediction: each text token's output goes through a classification head which must predict the next token in the caption.

You may be surprised that the Q-Former is used to encode text (for ITM and ITC) and also to generate text (for LM). Since the Q-Former is initialized using the weights of a pretrained BERT-base model, it's pretty good at text encoding right from the start of training, but it initially doesn't know that it has to predict the next token for the LM task. Luckily, it can learn fairly fast since it's not starting from scratch, it has good BERT features to work with. However, we need to tell it whether we want it to encode the text or predict the next token. For this, we replace the class token with a *decode token* during LM.[39]

TIP

To produce negative examples for ITM, one strategy is to randomly pick a caption in the same batch, excluding the image's true caption. However, this makes the task too easy, so the model doesn't learn much. Instead, the authors used a *hard negative mining* strategy, where difficult captions are more likely to be sampled. For example, given a photo of a chimpanzee, the caption "A gorilla" is more likely to be sampled than "A spacecraft". To find difficult captions, the algorithm uses the similarity scores from the ITC task.

Once stage 1 is finished, the Q-Former is already a powerful model that can encode images and text into the same space, so a photo of a chimpanzee produces a very similar output representation as the caption "A photo of a chimpanzee". But it's even better than that: the query token outputs were trained to be most helpful for next token prediction.

So it's time for the second stage of training (see Figure 16-19):

- We keep the vision transformer and the Q-Former, but we drop the rest and we add a new linear layer, initialized randomly, on top of the Q-Former.
- For each image/caption pair, the Q-Former attends to the visual features produced by the pretrained vision encoder, and the outputs go through the linear layer to produce a sequence of visual query tokens.
- The visual query tokens and the text token representations are concatenated and fed to the (frozen) pretrained LLM. We train BLIP-2 to predict the next caption token.

[...] *The Tower of Buddhist incense*

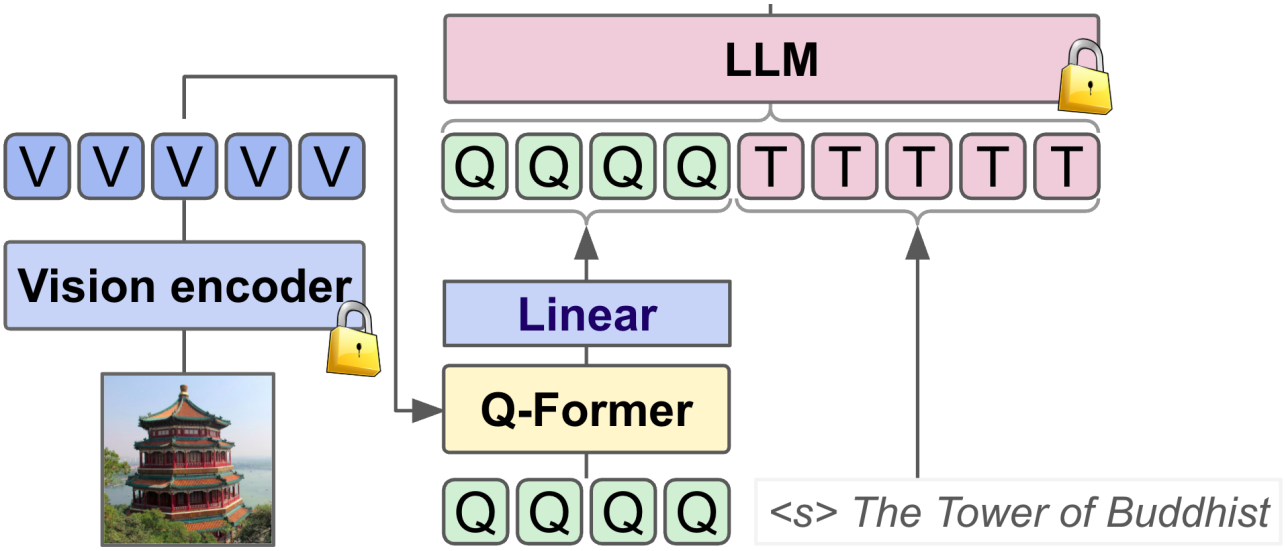


Figure 16-19. BLIP-2 Pretraining – Stage 2: training the linear layer to map the query tokens to the LLM's input space

During stage 2, the model learns to properly map the visual query tokens to the LLM's input space. Once trained, the model can be used like in stage 2, generating visual-grounded text.

Let's use BLIP-2 to generation a caption for an image:

```
from transformers import Blip2Processor, Blip2ForConditionalGeneration

model_id = "Salesforce/blip2-opt-2.7b"
blip2_processor = Blip2Processor.from_pretrained(model_id)
blip2_model = Blip2ForConditionalGeneration.from_pretrained(
    model_id, device_map=device, torch_dtype=torch.float16)

image_url = "http://images.cocodataset.org/val2017/000000039769.jpg"  # two cats
image = Image.open(urllib.request.urlopen(image_url))
inputs = blip2_processor(images=image, return_tensors="pt")
inputs = inputs.to(device, dtype=torch.float16)
with torch.no_grad():
    generated_ids = blip2_model.generate(**inputs)

generated_text = blip2_processor.batch_decode(generated_ids)
```

What did BLIP-2 see?

```
>>> generated_text
['<image><image><image><image>[...]<image></s>two cats laying on a couch\n']
```

It's a good description of the photo, but it would nicer without the special tokens, so let's get rid of them when decoding the model's output:

```
>>> generated_text = blip2_processor.batch_decode(generated_ids,
...                                               skip_special_tokens=True)
...
>>> generated_text
```

```
>>>
['two cats laying on a couch\n']
```

Perfect!

TIP

Also check out InstructBLIP, a BLIP-2 model with vision-language instruction tuning.

Other Multimodal Models

We've covered quite a few multimodal models, with very different architectures and pretraining techniques, but of course there are many others. Here is a quick overview of some of the most notable ones, in chronological order:

- LayoutLM (Microsoft, Dec 2019): document understanding based on text, vision, and document layout. Version 3 was released in April 2022.
- GLIP (Microsoft, Dec 2021): visual grounding and object detection. GLIP-2 was released in 2022.
- Stable Diffusion (Stability AI, Dec 2021): a powerful text-to-image model.
- OFA (Microsoft, Feb 2022): unified (one for all) vision-language pretraining framework handling various vision-language tasks.
- CoCa (Google, May 2022): pretrained using contrastive and captioning objectives. It influenced later models like PaLI-X and Flamingo-2.
- PaLI (Google, Sep 2022): multilingual multimodal models for vision-language tasks like VQA and captioning, with strong zero-shot performance. The next versions, PaLI-X and PaLI-3, were released in 2023.
- Kosmos-1 (Microsoft, Feb 2023): a multimodal LLM with strong support for visual grounding. Kosmos-2 and Kosmos-2.5 came out in 2023.
- PaLM-E (Google, Mar 2023): extends Google's PaLM series with visual inputs and embodied sensor data. A decoder-only LLM generates text commands like "grab the hammer", which are interpreted and executed by a robot via a downstream system.
- LLaVA (H. Liu et al., Apr 2023): among the best open source vision-language chat models.
- MiniGPT-4 (KAUST, Apr 2023): a lightweight and open multimodal chat model based on BLIP-2 and Vicuna.
- ImageBind (Meta, May 2023): a CLIP-style model extended to 6 modalities (image, text, audio, IMU[40], depth, thermal).
- RT-2 (DeepMind, Jul 2023): a vision-language model also capable of robotic control, trained on a large scale instruction-following dataset.
- SeamlessM4T (Meta, Aug 2023): a single model that can perform speech-to-text, speech-to-speech, text-to-speech, and text-to-text translation across close to 100 languages.
- Fuyu (Adept AI, Oct 2023): processes interleaved image and text in real time with a unified transformer.
- EMO (Alibaba, Feb 2024): takes an image of a person, plus an audio recording of someone

- EMO (Alibaba, Feb 2024): takes an image of a person, plus an audio recording of someone speaking or singing, and the model generates a video of that person, matching the audio. EMO-2 was released in January 2025.
- Yi-VL (01.AI, Mar 2024): strong BLIP-2-style vision-language models.
- GLaMM (H. Rasheed et al., Jun 2024): a visual dialogue model which generates text responses mixed with object segmentation masks.

TIP

I've created homl.info short links for all the models discussed in this chapter, just use the lowercase name without hyphens, for example: *https://homl.info/yivl*.

There are also several commercial multimodal models whose detailed architectures were not disclosed, such as GPT-4.1 and Sora by OpenAI, Gemini 2.5 Pro by Google, and Claude 4 Opus by Anthropic. To access these models, you first need to create an account and get a subscription (or use the free tier), then you can either use the provided apps (e.g., Google AI Studio, *https://aistudio.google.com*), or query the model via an API. For example, below is a short code example showing how to query Gemini 2.5 Pro via the API. You first need to get an API key in Google AI Studio, then you can use any secret management method you prefer to store it and load it in your code (e.g., if you are using Colab, I recommend you use Colab's secret manager, as we saw in Chapter 15).

```
from google import genai

gemini_api_key = [...]  # load from Colab secrets, or from a file, or hardcode
gemini_client = genai.Client(api_key=gemini_api_key)
cats_photo = gemini_client.files.upload(file="my_cats_photo.jpg")
question = "What animal and how many? Format: [animal, number]"
response = gemini_client.models.generate_content(
    model="gemini-2.5-flash",  # or "gemini-2.5-pro"
    contents=[cats_photo, question])
print(response.text)  # prints: "[cat, 2]"
```

This code uses the `google-genai` library, which is already installed on Colab. It also assumes that a file named `my_cats_photo.jpg` is present in the same directory as the notebook.

This wraps up this chapter; I hope you enjoyed it. Transformers can now see, hear, touch, and more! If you'd like to go further, Chapter 17 introduces some of the main advanced transformer techniques, designed to scale and speed up transformers. After that, we will switch to an entirely different topic: representation learning and generative ML using autoencoders, generative adversarial networks (GANs), and diffusion models.

Exercises

1. Can you describe the original ViT's architecture? Why does it matter?

2. What tasks are regular ViTs (meaning non-hierarchical) best used for? What are their limitations?
3. What is the main innovation in DeiT? Is this idea generalizable to other architectures?
4. What are some examples of hierarchical ViTs? What kind of tasks are they go for?
5. How do PVTs and Swin Transformers reduce the computational cost of processing high resolution images?
6. How does DINO work? What changed in DINOv2? When would you want to use DINOv2?
7. What is the objective of the JEPA architecture? How does it work?
8. What is a multimodal model? Can you give five examples of multimodal tasks?
9. Explain what the fusion and alignment problems are in multimodal learning? Why are transformers well suited to tackle them?
10. Can you write a one-line summary of the main ideas in VideoBERT, ViLBERT, CLIP, Dall-E, Perceiver IO, Flamingo, and BLIP-2?
11. If you are using a Perceiver IO model and you double the length of the inputs and the outputs, approximately how much more computation will be required?
12. Try fine-tuning a pretrained ViT model on the Food 101 dataset (`torchvision.datasets.Food101`). What accuracy can you reach? How about using a CLIP model, zero-shot?
13. Create a simple search engine for your own photos: first, write a function that uses a CLIP model to embed all of your photos and saves the resulting vectors. Next, write a function that takes a search query (text or image), embeds it using CLIP, then finds the most similar photo embeddings and displays the corresponding photos. You can manually implement the similarity search algorithm, or a dedicated library such as the FAISS library or even a full-blown vector database.
14. Use BLIP-2 to automatically caption all of your photos.

[1] Kelvin Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.

[2] This is a part of Figure 3 from the paper. It is reproduced with the kind authorization of the authors.

[3] Marco Tulio Ribeiro et al., "'Why Should I Trust You?': Explaining the Predictions of Any Classifier", *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

[4] Nicolas Carion et al., "End-to-End Object Detection with Transformers", arXiv preprint arXiv:2005.12872 (2020).

[5] Alexey Dosovitskiy et al., "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale", arXiv preprint arXiv:2010.11929 (2020).

[6] Hugo Touvron et al., "Training Data-Efficient Image Transformers & Distillation Through Attention", arXiv preprint arXiv:2012.12877 (2020).

[7] Wenhai Wang et al., "Pyramid Vision Transformer: A Versatile Backbone for Dense Prediction without Convolutions", arXiv preprint arXiv:2102.12122 (2021).

[8] Ze Liu et al., “Swin Transformer: Hierarchical Vision Transformer using Shifted Windows”, arXiv preprint arXiv:2103.14030 (2021)

[9] Ze Liu et al., “Swin Transformer V2: Scaling Up Capacity and Resolution”, arXiv preprint arXiv:2111.09883 (2021).

[10] Mathilde Caron et al., “Emerging Properties in Self-Supervised Vision Transformers”, arXiv preprint arXiv:2104.14294 (2021).

[11] These images are extracted from Figure 3 of the DINO paper, and are reproduced with the kind authorization of the authors.

[12] This is the righthand part of figure 3 from the DINO paper, reproduced with the kind authorization of the authors.

[13] Yangtao Wang et al., “TokenCut: Segmenting Objects in Images and Videos with Self-supervised Transformer and Normalized Cut”, arXiv preprint arXiv:2209.00383 (2022).

[14] “DINOv2: Learning Robust Visual Features without Supervision”, arXiv preprint arXiv:2304.07193 (2023).

[15] Xiaohua Zhai et al., “Scaling Vision Transformers”, arXiv preprint arXiv:2106.04560 (2021).

[16] Hangbo Bao et al., “BEiT: BERT Pre-Training of Image Transformers”, arXiv preprint arXiv:2106.08254 (2021).

[17] Kaiming He et al., “Masked Autoencoders Are Scalable Vision Learners”, arXiv preprint arXiv:2111.06377 (2021).

[18] Mitchell Wortsman et al., “Model Soups: Averaging Weights of Multiple Fine-tuned Models Improves Accuracy Without Increasing Inference Time”, arXiv preprint arXiv:2203.05482 (2022).

[19] Yuxin Fang et al., “EVA: Exploring the Limits of Masked Visual Representation Learning at Scale”, arXiv preprint arXiv:2211.07636 (2022).

[20] “Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture”, arXiv preprint arXiv:2301.08243 (2023).

[21] Yann LeCun, “A Path Towards Autonomous Machine Intelligence” (2022).

[22] Chen Sun et al., “VideoBERT: A Joint Model for Video and Language Representation Learning”, arXiv preprint arXiv:1904.01766 (2019).

[23] L. Zhou, Y. Zhou et al., “End-to-End Dense Video Captioning with Masked Transformer”, *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018).

[24] Jiasen Lu et al., “ViLBERT: Pretraining Task-Agnostic Visiolinguistic Representations for Vision-and-Language Tasks”, Advances in neural information processing systems 32 (2019).

[25] Jize Cao et al. later provided some empirical evidence supporting this claim in their paper “Behind the Scene: Revealing the Secrets of Pre-trained Vision-and-Language Models”: in particular, they found that

more attention heads focus on the text modality than on the visual modality.

[26] Alec Radford et al., “Learning Transferable Visual Models From Natural Language Supervision”, arXiv preprint arXiv:2103.00020 (2021).

[27] The training code and data were not released by OpenAI, but Gabriel Ilharco et al. created OpenCLIP, which is a flexible open source replication of CLIP with the full training code and data.

[28] This contrastive loss was first introduced as the *multiclass n-pair loss* in a 2016 paper by Kihyuk Sohn, then used for contrastive representation learning and renamed to *InfoNCE* (information noise-contrastive estimation) in a 2018 paper by Aaron van den Oord et al.

[29] Aditya Ramesh et al., “Zero-Shot Text-to-Image Generation”, arXiv preprint arXiv:2102.12092 (2021).

[30] Aditya Ramesh et al., “Hierarchical Text-Conditional Image Generation with CLIP Latents”, arXiv preprint arXiv:2204.06125 (2022).

[31] Andrew Jaegle et al., “Perceiver: General Perception with Iterative Attention”, arXiv preprint arXiv:2103.03206 (2021).

[32] If Δ is the spacing between samples, then the Nyquist–Shannon sampling theorem tells us that the maximum frequency we can measure is $f = 1 / 2\Delta$. This is why f stops at $\mu / 2$ rather than μ: sampling at a higher resolution would not add any information, and it might introduce aliasing artifacts.

[33] AudioSet contains over 2 million video segments of 10s each, sorted into over 500 classes.

[34] ModelNet40 is a synthetic dataset of 3D point clouds of various shapes such as airplanes or cars. A common source of point clouds in real life is LiDAR sensors.

[35] Andrew Jaegle et al., “Perceiver IO: a General Architecture for Structured Inputs & Outputs”, arXiv preprint arXiv:2107.14795 (2021).

[36] In the French comic series Astérix, Obélix is a big and friendly Gaul, and Idéfix is his clever little dog.

[37] Junnan Li et al., “BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation”, arXiv preprint arXiv:2201.12086 (2022)

[38] Junnan Li et al., “BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models”, arXiv preprint arXiv:2301.12597 (2023).

[39] The idea of training a single model capable of both encoding and generating text was introduced in 2019 by Microsoft researchers Li Dong et al. with their UniLM model.

[40] Most modern smartphones contain an inertial measurement unit (IMU) sensor: it measures acceleration, angular velocity, and often the magnetic field strength.