CHAPTER 22

Transformer Encoders for Images

Built into convolutional encoders is the idea that it is more important for a pixel to interact with its nearer neighbors than with distant neighbors. Call a feature that depends on a small neighborhood a short range feature. It is certainly easier for a convolutional encoder to produce short and medium range features than it is for it to produce long range features. There is a limit to how big the filters can be, so for features to involve well-separated pixels requires a deep encoder. Experience suggests that even if the receptive field of some feature in the network is very large, the sensitivity of the feature to distant pixels falls off quite fast. This general property is sometimes referred to as an *inductive bias*.

Inductive bias is not necessarily a bad thing. Imagine the features you wish to construct really should be short or medium range features. In this case, using an architecture that has difficulty building anything else is likely to be helpful. You will be using parameters efficiently. But if most of the features should be short or medium range and some should be very long range, the performance of your method will be limited. Such performance limits appear to have manifested for convolutional encoders, though being certain they are there is extremely difficult.

This chapter describes a new procedure to encode an image which has deep roots in the ideas of Chapter 10. The encoder represents an image as a collection of patches, and builds a representation out of patch relations. The encoder originates in the *transformer* used to encode and transform word sequences in natural language processing. A *vision transformer* or *ViT* can be used in slightly different forms for image to image mapping and for image classification.

Vision transformers are now very widely used. They integrate well with language representations, so they simplify building methods to map pictures to captions or text prompts to images. Experience shows vision transformers produce superior encodings of images for most tasks. However, immense amounts of data are required to train a vision transformer that works well. This is likely because the inductive biases in a convolutional encoder and in a transformer are different. Vision transformers also are very demanding of computation and change the nature of the computations required compared to convolutional encoders, so the performance gains may not be justified

22.1 VISION TRANSFORMER ENCODERS

The original transformer is an architecture that encodes a sequence of *tokens* into another sequence of latent representations. Tokens can be pretty much anything that can be represented by a fixed length vector. The first tokens were word embeddings – representations of words as vectors. The latent representations are then decoded into an output sequence. The decoder can be constructed so that the output sequence is *auto-regressive* – the $i+1$'th generated token is a function of the

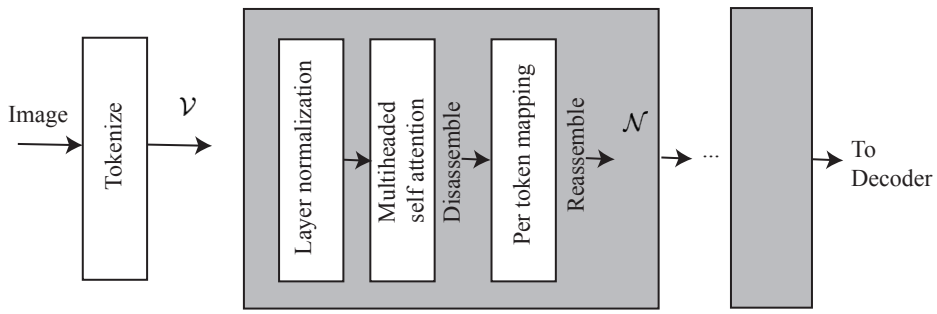

FIGURE 22.1: *A vision transformer (very often, ViT) accepts an image, tokenizes it producing a matrix whose rows are tokens (Section 22.1.4), then passes these tokens through a set of transformer layers (the* **gray** *boxes), then on to a decoder. The transformer layers consist of a layer normalization (Section* **??***), multiheaded self-attention (Section* **??***), and then a per-token mapping (Section* **??***). The mapped tokens are then reassembled into a matrix of tokens, and passed on.*

input and all or some earlier tokens. This property is extremely helpful if you intend to generate long sequences of words (paragraphs, books, etc) because it means you do not need to look at future words when you make the current word.

Figure 22.1 shows the overall structure of a vision transformer. The image is mapped into a set of N_t tokens. These tokens pass through a series of transformer layers, and then on to a decoder. The key feature of a transformer layer is attention. The layer turns a set of tokens into another set of tokens, *where each token in the resulting sequence is affected by every input token.* The output of the encoder will be a set of N_t tokens. There are various ways to decode this result, depending on the application.

22.1.1 Attention

The key operation in a transformer maps a token sequence to a token sequence. A token sequence is represented by a matrix. Each image will be tokenized into N_t tokens $\mathbf{v}_1 \dots \mathbf{v}_{N_t}$. It turns out to be helpful to have an extra, learnable, token $\mathbf{v}_0$. Stack the $\mathbf{v}_i$ into a matrix

$$\mathcal{V} = \begin{bmatrix} \mathbf{v}_0^T \\ \mathbf{v}_1^T \\ \dots \mathbf{v}_{N_t}^T \end{bmatrix}$$

and use this matrix to represent the tokens.

There are various important details (below), but at the heart of the mapping is *attention.* This is a procedure to re-estimate the embedding of each token taking each other token into account. Each token is replaced by a convex combination of all the tokens. The weights are constructed using a softmax function.

Definition: 22.1 *The softmax function*

The function that maps a vector $\mathbf{g}$ to a normalized vector $\mathbf{n}$ by

$$n_i = \frac{e^{g_i}}{\sum_u e^{g_u}}$$

is often referred to as the *softmax* function. Notice that all components of $\mathbf{n}$ are non-negative, and $\mathbf{1}^T\mathbf{n} = 1$.

Definition: 22.2 *Softmax for matrices*

Write $\text{softmax}(\mathcal{G})$ for the function that maps a matrix $\mathcal{G}$ to a matrix $\mathcal{W}$ where

$$w_{ij} = \frac{e^{g_{ij}}}{\sum_u e^{g_{iu}}}.$$

In this case, $\mathcal{W}\mathbf{1} = \mathbf{1}$. This means each row represents a set of *convex weights* (all non-negative, and sum to one; $\mathbf{1}$ is a vector of all ones).

Each token $\mathbf{v}_i$ has a *query* $\mathbf{q}_i$ and a *key* $\mathbf{k}_i$ associated with it. Where these might come from will be revealed below; for the moment, accept that they are there. These vectors have the same dimension. Attention uses these tokens to compute a matrix of weights to reestimate the tokens. Arrange the $\mathbf{k}_i$ and the $\mathbf{q}_i$ into matrices

$$\mathcal{Q} = \begin{bmatrix} \mathbf{q}_0^T \\ \mathbf{q}_1^T \\ \dots \mathbf{q}_{N_t}^T \end{bmatrix} \text{ and } \mathcal{K} = \begin{bmatrix} \mathbf{k}_0^T \\ \mathbf{k}_1^T \\ \dots \mathbf{k}_{N_t}^T \end{bmatrix}.$$

Now consider

$$\text{softmax}(\frac{\mathcal{Q}\mathcal{K}^T}{\sqrt{d_k}}).$$

This is a $(N_t + 1) \times (N_t + 1)$ matrix whose rows sum to one. So

$$\mathcal{N} = \text{softmax}(\frac{\mathcal{Q}\mathcal{K}^T}{\sqrt{d_k}})\mathcal{V} = \mathcal{W}\mathcal{V}$$

maps each token to a convex combination of tokens, where the weights are determined by the query and key corresponding to each token. This mechanism allows token representations to affect one another in a way that is determined by the keys and the queries. Tokens whose key vector are similar to the query vector of token i will have larger weights in the i'th row of $\mathcal{W}$, and so will tend to affect $\mathbf{n}_i$ more strongly.

The term $\sqrt{d_k}$ arises from experience. If a component of $\mathcal{Q}\mathcal{K}^T$ is very much larger than the others, the corresponding weight will be very close to 1. More

important, the gradient of the softmax at that component will be very small, which creates problems for learning. Experience shows that scaling by $\sqrt{d_k}$ is helpful, likely because this ensures that the largest weight isn't too large, and that the gradients of the softmax behave well.

Procedure: 22.1 *Attention*

Given an $(N_t + 1) \times e_{\mathrm{t}}$ matrix of tokens $\mathcal{V}$, an $(N_t + 1) \times d_k$ matrix of queries $\mathcal{Q}$ corresponding to those tokens, and a $(N_t + 1) \times d_k$ matrix of keys $\mathcal{K}$ corresponding to those tokens, attention computes a new set of tokens

$$\text{Attention}(\mathcal{Q}, \mathcal{K}, \mathcal{V}) = \text{softmax}(\frac{\mathcal{S}}{\sqrt{d_k}})\mathcal{V}.$$

where $\mathcal{S} = \mathcal{Q}\mathcal{K}^T$.

22.1.2 Self-Attention

But where do the keys and queries come from? in natural language models, they may come from other sequences. In ViT's, they come from the tokens in $\mathcal{V}$. You apply a learned linear mapping to each token to obtain a key, and another learned linear mapping to obtain a query. The mapping from token to key is the same for each token, and the mapping from token to query is the same for each token. Choose two $d_k \times e_{\mathrm{t}}$ matrices $\mathcal{M}_q$ and $\mathcal{M}_k$ (which will be learned), and apply these to the original tokens to get

$$\mathcal{Q} = \mathcal{V}\mathcal{M}_q^T \text{ and } \mathcal{K} = \mathcal{V}\mathcal{M}_k^T$$

and follow the box, below.

Procedure: 22.2 *Self Attention*

Choose two $d_k \times e_{\mathrm{t}}$ matrices $\mathcal{M}_q$ and $\mathcal{M}_k$ which will be learned. Self-attention maps $\mathcal{V}$ to

$$\text{Self Attention}(\mathcal{M}_q, \mathcal{M}_k, \mathcal{V}) = \text{Attention}(\mathcal{V}\mathcal{M}_q^T, \mathcal{V}\mathcal{M}_k^T, \mathcal{V}).$$

22.1.3 Multi-head Self-Attention

Self-attention maps tokens that are similar to tokens that are closer together, but does not allow two tokens to be similar in one respect, but different in another. As an example, think of a black cat and a black cadillac. If you attend to colors, they are similar; if you attend to categories, they are different. Multi-head self-attention deals with this problem by projecting tokens to a number of lower dimensional spaces; applying attention within those spaces; then reassembling the results.

Procedure: 22.3 *Multi-headed Self-attention*

Choose the dimension for the lower dimensional keys and values, d_c, and the dimension for the lower dimensional tokens, d_l, and the number of different lower dimensional spaces, s. Write $\mathcal{W}_{q,u}$ for a learnable $d_c \times e_{\text{t}}$ matrix that will map tokens to the u'th lower dimensional space to form queries, $\mathcal{W}_{k,u}$ for a learnable $d_c \times e_{\text{t}}$ matrix that will map tokens to the u'th lower dimensional space to form keys, $\mathcal{W}_{k,u}$ for a learnable $d_l \times e_{\text{t}}$ matrix that will map tokens to the u'th lower dimensional space lower dimensional tokens, and $\mathcal{W}_A$ for a $e_{\text{t}} \times (sd_i)$ learnable matrix that will map reassembled partial tokens to the original dimension. For an input stack of tokens $\mathcal{V}$, compute

$$\mathcal{A}_u = \text{Self Attention}(\mathcal{W}_{q,u}, \mathcal{W}_{k,u}, \mathcal{V}\mathcal{W}_u^T)$$

then form

$$\text{Multi-head Self-Attention}(\mathcal{V}, \mathcal{W}_{q,1}, \ldots, \mathcal{W}_s, \mathcal{W}_A)) = [\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_s]\, \mathcal{W}_A^T.$$

You should check that multi-headed self-attention produces a $N_t \times e_{\text{tokenl}}$ matrix.

22.1.4 Turning an Image into Tokens

It is possible to build transformers that accept images of variable size, but everything gets more complicated. I assume that images have fixed size here. All images have height H and width W that are fixed and are divisible by the patch size p. This choice is convenient for exposition and is quite usual. It is possible to build ViT's that accept variable size images, but doing so involves a certain amount of extra work. Typical numbers are $H = 224$, $W = 224$ and $p = 16$.

Now cut the image into non-overlapping square patches of fixed size $p \times p$. Arrange these patches into a sequence using, for example, a scan conversion order (top left patch goes first, top second left next, and so on till bottom right). You must use the same convention for each image. Now straighten each patch into a vector $\mathbf{x}_i$ for the i'th patch in the sequence. This will be $3p^2 \times 1$, the 3 coming from the color channels. Embed the vector corresponding to each patch using a learned linear operation, using the same operation for each patch and each image. You can think of this as a matrix $\mathcal{M}$, though it is usual to use a convolution with a learned kernel. This linear operation maps the patch vector into a (possibly higher dimensional) space to obtain $\mathbf{u}_i = \mathcal{M}\mathbf{x}_i$. You could use the sequence

$$\{\mathbf{u}_1, \ldots, \mathbf{u}_N\}$$

to represent the image.

Procedure: 22.4 *Tokenizing images: setup*

Choose

- an image size $H \times W$ and a token size p such that H and W are divisible by p;
- a convention for ordering the patches (typically, top left goes first, and so on to bottom right)
- and the embedding dimension for the patch e_{patch} (typically, **** and larger than ($d_{\text{patch}} = 3 \times p \times p$); for convenience, choose e_{patch} even).

Write $k = (H/p)(W/p)$ for the number of patches.

Procedure: 22.5 *Tokenizing images: embedding patches*

Learned parameters are a linear operation $\mathcal{M}$ for embedding patches. Options for $\mathcal{M}$ include: using the identity; using a convolution, where the kernel is learned; or using a learned matrix. Cut the image into $N_t = (H/p)(W/p)$ non-overlapping $p \times p$ patches. Straighten each patch into a vector $\mathbf{x}_i$, where i is the number of the patch given the ordering. Write

$$\mathbf{v}_i = \mathcal{M}\mathbf{x}_i$$

The image patches are then represented by:

$$\{\mathbf{v}_1, \mathbf{v}_1, \ldots, \mathbf{v}_{N_t}\}.$$

Experience shows it is helpful that a token "knows" where it is in the sequence. You can achieve this by constructing a *positional encoding* or *positional embedding*. There are several ways to construct a positional embedding. One approach – which you can call *additive positional embedding* – obtains a vector $\mathbf{p}_i$ the same size as the patch vector, and adds that vector to the patch vector. There is one $\mathbf{p}_i$ per location in the sequence (so i runs from $1 \ldots N$). Because different locations have different $\mathbf{p}_i$, the transformer has information about where a patch appears in a sequence. There are two natural ways to obtain such vectors.

Procedure: 22.6 *Fixed positional embedding*

Choose $\mathbf{p}_i$ to be constant vectors. Ideally, different components of the vector represent the position in the sequence at different levels of precision. A good choice is to use sinusoids []. Assume the token dimension e_{embed} is even. Then the $2t$'th and $2t+1$'th components of the i'th token are:

$$\begin{aligned} p_{i,2t} &= \sin\left(\frac{i}{1000^{(\frac{2t}{d})}}\right) \\ p_{i,2t+1} &= \cos\left(\frac{i}{1000^{(\frac{2t}{d})}}\right). \end{aligned}$$

Notice that the wavelengths of the sinusoids in i get longer and longer as t goes up.

Procedure: 22.7 *Learned positional embedding*

Let $\mathbf{p}_i$ be a learnable parameter.

A further very useful trick comes from natural language processing. Each input sequence has an extra, learnable token prepended to it. At the input, this *readout token* bears no relationship to the image – it is just a learned vector. The encoder can be trained so that, at that token's location in the output, there is a useful summary of the image.

Procedure: 22.8 *Tokenizing images: forming the token sequence*

Learnable parameters are $\mathbf{v}_0$, the readout token and optionally the positional embedding. Take the image patch vectors as above, add the appropriate positional encoding for the location, then prepend a readout token to form

$$\{\mathbf{v}_0, \mathbf{v}_1 + \mathbf{p}_1, \ldots, \mathbf{v}_{N_t} + \mathbf{p}_{N_t}\}.$$

22.1.5 From MHSA to a Transformer Layer

A transformer layer consists of two sub-layers. The first normalizes the input tokens, then applies multi-headed self-attention. The second then transforms the representation of each token to a new form, and renormalizes.

The first sublayer: if you look closely at self-attention, you will notice two things. Self-attention pulls tokens towards one another. This means repeated

application of self-attention may lead to a case where all tokens are essentially the same, which is unlikely to be a good representation of an image. A procedure called *layer normalization* discourages this effect. Layer normalization maps a token $\mathbf{v}_i$ to $\text{LN}(\mathbf{v}_i)$ by scaling and translating the feature values within the token $\mathbf{v}_i$ so that the mean is zero and the standard deviation is one.

Procedure: 22.9 *Layer Normalization*

Write

$$\mathbf{z}_i = (\mathbf{v}_i - \frac{(\mathbf{v}_i^T \mathbf{1})}{e_{\text{t}}} \mathbf{1})$$

then

$$\text{LN}(\mathbf{v}_i) = \frac{1}{\sqrt{\mathbf{z}_i^T \mathbf{z}_i}} \mathbf{z}_i$$

and

$$\text{LN}(\mathcal{V}) = \begin{bmatrix} \text{LN}(\mathbf{v}_1)^T \\ \dots \\ \text{LN}(\mathbf{v}_k)^T \end{bmatrix}.$$

It is now usual to normalize tokens *before* passing them to the attention operation. The gradient of an output token with respect to some input tokens might be very small (for example, one token might be very different from others). This property could make it hard to get good descent from a stack of self-attention layers (recheck Section 36.3 if you find this remark mysterious). This is easily cured with a residual connection (compare Section 36.3), leading to the box below.

Procedure: 22.10 *Transformer layer: first sub-layer*

For $\mathcal{V}$ an input stack of tokens, compute

$$\begin{aligned} \mathcal{U} &= \text{LN}(\mathcal{V}) \\ \mathcal{A} &= \mathcal{U} + \text{Multi-head Self-Attention}(\mathcal{U}, \mathcal{W}_{q,1}, \dots, \mathcal{W}_s, \mathcal{W}_A). \end{aligned}$$

Here $\mathcal{A}$ is the set of tokens leaving the first layer.)

The second sub-layer: The first sub-layer produces a set of tokens. The second sub-layer normalizes these tokens, then maps them with a learned mapping applied to each token individually. The normalization is the layer normalization above. Mapping uses a *GELU* non-linearity in place of a ReLU, where

$$\text{GELU}(x) = (\frac{x}{2})(1 + \text{erf}(\frac{x}{\sqrt{2}})$$

where $\text{erf}(u)$ is the *standard error function*,

$$\text{erf}(u) = \frac{2}{\sqrt{\pi}} \int_0^u e^{-t^2} dt.$$

Exploiting the considerable body of knowledge about fast approximations to the error function yields

$$\text{GELU}(x) \approx x\text{sigmoid}(1.702x)$$

where

$$\text{sigmoid}(u) = \left[\frac{e^u}{1+e^u}\right]$$

which looks an awful lot like a ReLU, with a small bump in the negative x domain (Figure **??**). Like a ReLU, a GELU can be applied to a vector or matrix or anything componentwise. All this yields the contents of the box, below.

Procedure: 22.11 *Transformer layer: second sub-layer*

Given a stack of tokens $\mathcal{V}$, a learnable $e_{\text{interior}} \times e_{\text{t}}$ matrix $\mathcal{W}_u$ and another learnable $e_{\text{t}} \times e_{\text{interior}}$ matrix $\mathcal{W}_d$, compute another stack of tokens $\mathcal{N}$ where:

$$\mathcal{N} = \begin{bmatrix} \mathbf{n}_0^T \\ \mathbf{n}_1^T \\ \dots \\ \mathbf{n}_{N_t} \end{bmatrix}$$

and

$$\mathbf{n}_i = \mathcal{W}_d \text{GELU}(\mathcal{W}_u \mathbf{v}_i).$$

22.2 DECODING VISION TRANSFORMERS

How you decode a ViT depends somewhat on what you want to do. If you want to classify images, it is simple and natural to take the $\mathbf{n}_0$ token from the last layer – sometimes known as a *readout token* – and simply pass this to a linear classifier. This option is commonly used to pretrain encoders as well. The readout token has seen contributions from all patches in the image, so is a reasonable choice for this purpose. If you want to map an image to an image-like thing, you need a feature map registered to the image. Obtaining one is relatively straightforward (Section 22.2.1). Finally, for some purposes you will want to decode into a sequence of tokens. For example, if you want to map images to sentences, you could map to a sequence of tokens then decode the tokens into phrases. As another example, you could decode a sequence of tokens into detector reports (an object of this category is at that location). You might consider just regarding the output of the ViT as a sequence of tokens – because it is one – but as Section **??**, there are reasons to build a decoder which will itself consist of transformer layers.

22.2.1 Dense Decoding

ViT's use a fixed number of image patches, and this makes it relatively straightforward to decode the transformer state into an image-like representation. I describe the procedures of [**?**]. This takes matrices of tokens output by various layers in

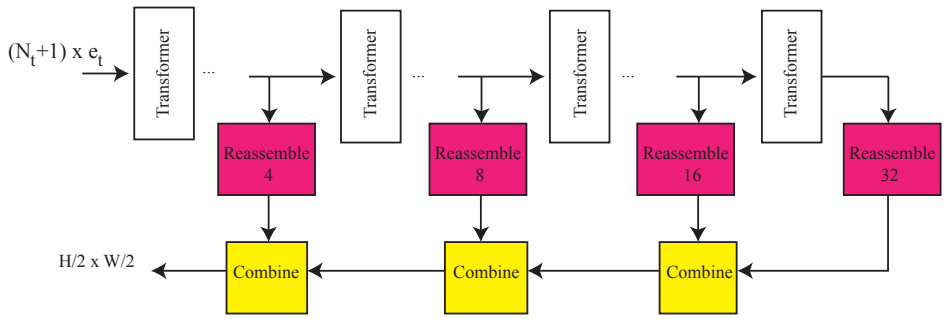

FIGURE 22.2: *Decoding a vision transformer representation into an image-like thing involves extracting blocks of tokens passing between various transformer layers; adjusting these blocks into image registered maps (the* **pink** *reassemble boxes); and combining these blocks with blocks coming down from earlier layers (the* **yellow** *combine boxes). Reassemble boxes have a parameter (write s), which is the scale at which they reconstruct the image. Further details on reassemble and combine boxes in Figure 22.3 and in the text. If you track the parameters through the drawing, you will see that this decode finally produces a block that is $(H/2) \times (W/2) \times D$, which is easily decoded into an image-like thing. Notice how this block contains information from early transformer layers as well as from later layers; there is good experimental evidence that this structure is helpful [].*

the encoder, reassembles them into something registered to the image, then combines them (Figure 22.2). Choose two feature sizes D and D_r. The result is a $H/2 \times W/2 \times D$ feature map, which is easily decoded into (say) a depth map, etc.

The reassemble box consists of a "read" step, an "assemble" step, a "project" step and a "resample" step. The box takes two parameters: s which represents the scale of the reassembled feature map compared to the original image, and D is the desired feature dimension. This box must take a token matrix of fixed size $(N_t + 1) \times e_{\mathrm{t}}$, and adjust this into a block that is $(H/s) \times (W/s) \times D_r$.

The "read" and "assemble" steps: Each transformer layer produces a set of $(N_t + 1) = (H/p) \times (W/p) + 1$ tokens of dimension e_{t}. All but one – the readout token – are associated with image patches. The readout token presents a mild problem. If you ignore it, you can rearrange the remaining tokens into something registered to the image. It turns out that you will generally get better results if you take each non-readout token, append the readout token to it, then apply a linear layer followed by a GELU to get a vector the same size as the original token. Once the readout token is dealt with, arrange the $(H/p) \times (W/p)$ tokens into a $(H/p) \times (W/p) \times e_{\mathrm{t}}$ feature map by placing each token in the location of the original patch.

The "project" and "resample" steps: Now apply a linear operator – a 1×1 convolution – to project the feature map to $(H/p) \times (W/p) \times e_{\mathrm{t}}$ to a $(H/p) \times (W/p) \times D_r$ dimensional feature map. If $s < p$, you will need to resize the feature map, possibly using interpolation. Otherwise, use a strided 3×3 convolution

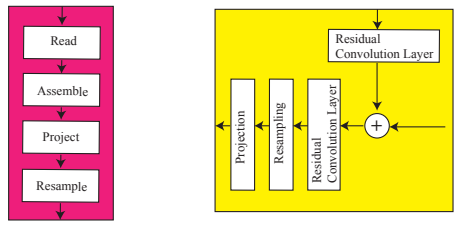

FIGURE 22.3: *Some more detail on the colored boxes of Figure 22.2. The reassemble (***pink***) box adjusts the matrix of tokens to a more convenient size (read); rearranges these tokens back into an image-like thing (assemble); reduces the dimension of that result (project); then finally resamples to achieve the required size (resample). The combine (***yellow***) box passes the block from above through a residual convolutional layer, then adds it to the block from the right; passes the result through another residual convolutional layer; resamples it to the required size; then finally projects the result to have the required feature dimension.*

to map this to a $(H/s) \times (H/s) \times D_r$ dimensional feature map.

The combine box consists of two residual convolution layers, resampling and projection. These boxes accept a block that is $(H/s) \times (W/s) \times D$ from the **right** and a block that is $(H/s) \times (W/s) \times D_r$ from **above**, and produce a block that is $(2H/s) \times (2W/s) \times D$. layers are the encoder blocks of Figure 18.3. The results of these layers are added. Finally, you upsample the resulting feature map by a factor of 2, and project as required with a 1×1 convolution to get the feature dimension right.

If you pay close attention to this account, you will notice that you should be able to build a transformer encoder and dense decoder that could accept a wide range of image sizes. The transformer doesn't really care how many tokens it must handle, though some care is required with the positional encoding. Very minor adjustments to the decoder will produce something that works. The details can be found in [] or at `https://github.com/isl-org/DPT`.

22.2.2 Decoding into Tokens

Assume you have a transformer encoder, and you wish to produce a decode that is a sequence. It is natural to use a set of transformer layers to decode because transformers are sequence to sequence mappings. In Section 22.1.1, a token was reestimated as a weighted sum of all tokens. The weights were computed by comparing that token's query with the keys associated with other tokens. Self-attention obtains the queries and keys from the tokens themselves. It isn't particularly interesting to use self-attention in all the decoder layers because the result would just be a slightly bigger encoder. But you don't have to do this.

In one alternative, arrange the first transformer layer in the decoder to accept the encoder output and use self-attention to compute queries and keys. Now the second transformer layer uses the encoder output as values and to form keys, but the output of the first decoder layer to form queries. The third uses the encoder

output as values and to form keys, but the output of the second decoder layer to form queries. You can think about this *encoder-decoder attention* as producing a very highly adapted form of the encoder output. Each transformer layer adjusts the communication between decoder tokens to produce something more like the desired output.

In a second alternative – which I believe to be unique to object detection applications – arrange the first transformer layer in the decoder to accept the encoder output to form keys and values, and supply a learnable set of tokens with positional encoding to form queries. The details are in Section **??**. You can think about this procedure as using the object queries to produce a very highly adapted form of the encoder output. Each transformer layer adjusts the communication between decoder tokens to produce something more like the desired output.