

# Chapter 1

## Transformers

*Draft chapter from Torralba, Isola, Freeman (this is a computer vision textbook, hence the emphasis on CNNs; note that transformers are also very related to GNNs)*

**Transformers** are a recent family of architectures that generalize and expand the ideas behind CNNs. The term for this family of architectures was coined by [Vaswani et al. 2017], where they were applied to language modeling. Our treatment in this chapter more closely follows the “Vision Transformers” introduced in [Dosovitskiy et al. 2021].

Like CNNs, transformers factorize the signal processing problem into stages that involve independent and identically processed chunks. However, they also include layers that mix information across the chunks, called “attention layers”, so that the full pipeline can model joint dependences between the chunks.

### 1.1 A limitation of CNNs: independence between far apart patches

CNNs are built around the idea of *locality*: different local regions of an image can safely be processed independently. This is what allows us to use filters with small kernels. However, very often, there is global information that needs to be shared across all receptive fields in an image. Convolutional layers are not well-suited to *globalizing* information since the only way they can do so is by either increasing the kernel size of their filters, or by stacking layers to increase the receptive field of neurons on deeper layers.

How can we efficiently pass messages across large spatial distances? We already have seen one option: just use a fully-connected layer, so that every output neuron after this layer takes input from every neuron on the layer before. However, fully-connected layers have a ton of parameters –  $N^2$  if their input and output are  $N$ -dimensional vectors – and it can take an exorbitant amount of time and data to fit all those parameters. Can we come up with a more efficient strategy?

### 1.2 The idea of attention

Attention is a strategy for processing global information efficiently, focusing just on the parts of the signal that are most salient to the task at hand. The idea can be motivated by attention in human perception. When we look at a scene, our eyes flick around and we “attend to” certain elements that stand out, rather than taking in the whole scene at once [Wolfe 2000]. If we are asked a question about the color of a car in the scene, we will move our eyes to look at the car, rather than just staring passively. Can we give neural nets the same ability?

In neural nets, attention follows the same intuitive idea. A set of neurons on layer  $l + 1$  may *attend* to a set of neurons on layer  $l$ , in order to decide what their response should be. If we “ask” that set of neurons to report the color of any cars in the input image, then they

Transformers were originally introduced in the field of natural language processing, where they were used to model language – sequences of characters and words. As a result, some texts present transformers as an alternative to RNNs for sequence modeling, but in fact transformer layers are *parallel* processing machines, like convolutional layers, rather than a sequential machine, like recurrent layers. Therefore, we will present transformers as more related to CNNs than RNNs.

should direct their attention to the neurons on the layer before that represent the color of the car. We will soon see how this is done, in full detail, but first we need to introduce a new data structure and a new way of thinking about neural processing.

### 1.3 A new data type: tokens

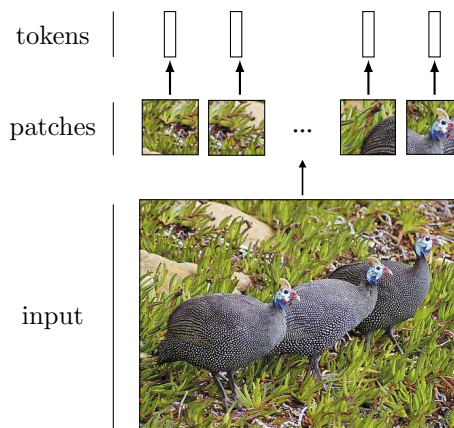
We discussed that the main data structures in deep learning are different kinds of groups of neurons – channels, tensors, batches, etc. Now we will introduce another fundamental data structure, **tokens**. A “token” is another kind of group of neurons, but there are particular ways we will operate over tokens that are different from how we operated over channels, batches, and the other groupings we saw before. Specifically, we will think of tokens as *encapsulated* groups of information; we will define operators over tokens, and these operators will be our only interface for accessing and modifying the internal contents of tokens. From a programming languages perspective, you can think of tokens as a new data *type*.

In this chapter we will only consider token whose internal content is a vector of neurons. We will call this vector the token’s **code** vector; the code for a token  $t$  will be labeled as  $t.z$ .

Transformers consist of two main operations over tokens: 1) *mixing* tokens via a weighted sum, and 2) *modifying* each individual token via a nonlinear transformation. These operations are analogous to the two workhorses of regular neural nets: the linear layer and the pointwise nonlinearity. Before we get to that, though, how do we turn data into tokens in the first place?

#### 1.3.1 Tokenizing data

The first step to working with tokens is to *tokenize* the raw input data. Once we have done this, all subsequent layers will operate over tokens, until the output layer, which will make some decision or prediction as a function of the final set of tokens. How can we tokenize an input image? Well, how did we “neuronize” an image for processing in a vanilla neural net? We simply represented each *pixel* in the image with a neuron (or three neurons, if it’s a color image). To tokenize an image, we may simply represent each *patch of pixels* in the image with a token. The token vector is the vectorized patch (stacking the three color channels one after the other). With each patch represented by a token, the full image corresponds to an array of tokens. Here’s what it looks like to tokenize an image of guineafowl in this way:



#### 1.3.2 Mixing tokens

Once we have converted our data to tokens, we now need to define operations for transforming these tokens and eventually making decisions based on them. The first key operation we will define is how to take *linear combinations of tokens*.

Although we are only considering vector-valued tokens in this chapter, it’s easy to imagine tokens that are any kind of structured group. We just need to define how basic operators, like summation, operate over these groups (and, ideally, in a differentiable manner).

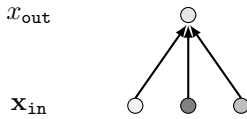
A linear combination of tokens is not the same as a fully connected layer in a neural net. Instead of taking a weighted sum of scalar neurons, it takes a weighted sum of token code vectors:

$$x_{\text{out}} = \sum_{i=1}^N w_i x_{\text{in}_i} \quad \triangleleft \text{linear comb of neurons} \quad (1.1)$$

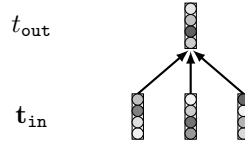
$$t_{\text{out}} \cdot \mathbf{z} = \sum_{i=1}^N w_i t_{\text{in}_i} \cdot \mathbf{z} \quad \triangleleft \text{linear comb of tokens} \quad (1.2)$$

As notational convenience, in this chapter we define  $\mathbf{x}_{\text{in}} = [x_1, \dots, x_N]$  and  $\mathbf{t}_{\text{in}} = [t_1, \dots, t_N]$ .

**Linear combination of neurons**



**Linear combination of tokens**



Operations over tokens can be defined just like operations over neurons except that the tokens are vector-valued while the neurons are scalar-valued. Most layers we have seen can be defined for tokens in an analogous way to how they were defined for neurons, like we saw with the token linear combination.

For example, we define an fc-layer over token codes as a mapping from  $N_1$  input tokens to  $N_2$  output tokens, parameterized by a matrix  $\mathbf{W} \in \mathbb{R}^{N_2 \times N_1}$  (and, optionally, a set of biases  $\mathbf{b} \in \mathbb{R}^{N_2 \times M}$  (for token's with  $M$ -dimensional code vectors).

For vector-valued tokens, these layers can be written compactly by defining  $\mathbf{Z}_{\text{in}} \in \mathbb{R}^{N_1 \times M}$  and  $\mathbf{Z}_{\text{out}} \in \mathbb{R}^{N_2 \times M}$  as matrices whose rows are  $M$ -dimensional token code vectors (transposed since by convention we use column vectors in this book):

$$\mathbf{Z}_{\text{in}} = \begin{pmatrix} t_1 \cdot \mathbf{z}^T \\ \vdots \\ t_N \cdot \mathbf{z}^T \end{pmatrix} \quad (1.3)$$

Then the fc-layer over token codes can be written as:

$$\mathbf{Z}_{\text{out}} = \mathbf{W}\mathbf{Z}_{\text{in}} + \mathbf{b} \quad \triangleleft \text{fc-layer over token codes} \quad (1.4)$$

Notice how the structure is analogous to fc-layers over neurons, except that the elements of the input are vector-valued. We can proceed in this fashion and make analogous token layers for any neuron-layer. For example, we could define a convolution layer over tokens as just like a convolution over neurons except that each weighted sum is a linear combination of tokens rather than a linear combination of neurons. Let's write this out for the simple case of 1D convolution with a single filter  $\mathbf{w}$  over a 1D array of token code vectors:

$$t_{\text{out}} \cdot \mathbf{z} = \mathbf{w} \star t_{\text{in}} \cdot \mathbf{z} \quad \triangleleft \text{conv over token codes} \quad (1.5)$$

$$t_{\text{out}}[i] \cdot \mathbf{z} = \sum_{k=-N}^N \mathbf{w}[k] t_{\text{in}}[i - k] \cdot \mathbf{z} \quad (1.6)$$

This layer is not currently popular but maybe it will be in the future. For any neural layer you come across, you may want to consider: what if I make it a token layer instead?

**1.3.3 Modifying tokens**

Linear combinations only let us linearly mix and recombine tokens, and stacking linear functions can only result in another linear function. In standard neural nets, we ran into

This notation is compact and turns working with tokens into an exercise in matrix algebra. However, the notation here is also somewhat limiting, as it only applies to vector-valued tokens. What if we want tokens that are tensor-valued, or tokens whose codes are elements of an abstract group such as  $\text{SO}(3)$ ? There is not yet standard notation for working with tokens like this. As you read this chapter try to think about how the operations we define for standard vector-valued tokens could be instead defined for other kinds of tokens.

the same problem with fully-connected and convolutional layers, which, on their own, are incapable of modeling nonlinear functions. To get around this limitation, we added *pointwise nonlinearities* to our neural nets. These are functions that apply a nonlinear transformation to each neuron *individually*, independently from all other neurons. Analogously, for “token networks” we will also introduce “pointwise” operators – these are functions that apply a nonlinear transformation to each token individually, independently from all other tokens. Given a nonlinear function  $F_\theta : \mathbb{R}^N \rightarrow \mathbb{R}^N$ , a token-wise nonlinearity layer can be expressed as:

$$\mathbf{t}_{\text{out}} = [F_\theta(t_1.\mathbf{z}), \dots, F_\theta(t_N.\mathbf{z})] \quad \triangleleft \text{per-token nonlinearity} \quad (1.7)$$

Notice that this operation is generalization of the pointwise nonlinearity in regular neural nets; a **relu** layer is the special case where  $N = 1$  and  $F_\theta = \text{relu}$ :

$$\mathbf{x}_{\text{out}} = [\text{relu}(x_1), \dots, \text{relu}(x_N)] \quad \triangleleft \text{per-neuron nonlinearity (relu layer)} \quad (1.8)$$

$F_\theta$  may be any nonlinear function but some choices will work better than others. One popular choice is for  $F_\theta$  to be an MLP (multi-layer perceptron; see chapter ??). In this case,  $F_\theta$  has learnable parameters  $\theta$  which are the weights and biases of the MLP. This reveals an important difference between pointwise operations in regular neural nets and in token nets: **relus**, and most other neuron-wise nonlinearities, have no learnable parameters, whereas  $F_\theta$  typically does. This is one of the interesting things about working with tokens, the pointwise operations become expressive and parameter-rich.

**CNNs in disguise** Pointwise operations apply the same operation independently and identically to all elements of an input array. Where have we seen that before? That’s right, convolution! In Chapter ?? we emphasized that the key idea of CNNs is to break up an input signal into chunks and process each chunk independently and identically. While a single convolutional layer is linear, a full CNN is a “pointwise” nonlinear function over patches of the input signal – and that’s precisely what a per-token MLP is, just with patches of spatial size  $1 \times 1$ .

So, for any per-token MLP, there is an equivalent CNN, which only uses kernels of size  $1 \times 1$ . Moreover, the first step in a token-net – tokenizing the input by vectorizing  $k \times k$  patches – can also be represented as a convolutional layer: in this case there are  $N$  filters of size  $k \times k$ , each of which picks out a single pixel in the image patch, to create  $N$  output channels that correspond to the vectorized patch. Really, the only *new* thing in token nets (and transformers, as we will see) is the attention layer. Otherwise, transformers are just CNNs in disguise.

As an exercise, we write out below two equivalent views of a per-token MLP, first as a pointwise nonlinearity over tokens and second as a CNN over neurons. The MLP is of the form **linear-relu-linear**, and the input is a 1D tensor of tokens  $\mathbf{t}_{\text{in}}$ , which can equivalently be represented as a 2D tensor of neurons  $\mathbf{X}_{\text{in}}$  whose rows are the token codes:

$$\mathbf{t}_{\text{out}} = [F_\theta(t_1.\mathbf{z}), \dots, F_\theta(t_N.\mathbf{z})] \quad (1.9)$$

Per-token MLP over  $\mathbf{t}_{\text{in}}$ :

$$\mathbf{a} = [\mathbf{W}_1 t_1.\mathbf{z} + \mathbf{b}_1, \dots, \mathbf{W}_1 t_N.\mathbf{z} + \mathbf{b}_1] \quad \triangleleft \text{per-token linear} \quad (1.10)$$

$$\mathbf{h} = [\text{relu}(\mathbf{a}_1), \dots, \text{relu}(\mathbf{a}_N)] \quad \triangleleft \text{relu} \quad (1.11)$$

$$\mathbf{t}_{\text{out}} = [\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2, \dots, \mathbf{W}_2 \mathbf{h}_N + \mathbf{b}_2] \quad \triangleleft \text{per-token linear} \quad (1.12)$$

Equivalent CNN over  $\mathbf{X}_{\text{in}}$ :

$$\mathbf{a}[:, k] = \sum_c \mathbf{W}_1[c, k] \star \mathbf{X}_{\text{in}} + \mathbf{b}_1[k] \quad \forall k \quad \triangleleft \text{conv} \quad (1.13)$$

$$\mathbf{h} = [\text{relu}(\mathbf{h}[1, :]), \dots, \text{relu}(\mathbf{h}[N, :])] \quad \triangleleft \text{relu} \quad (1.14)$$

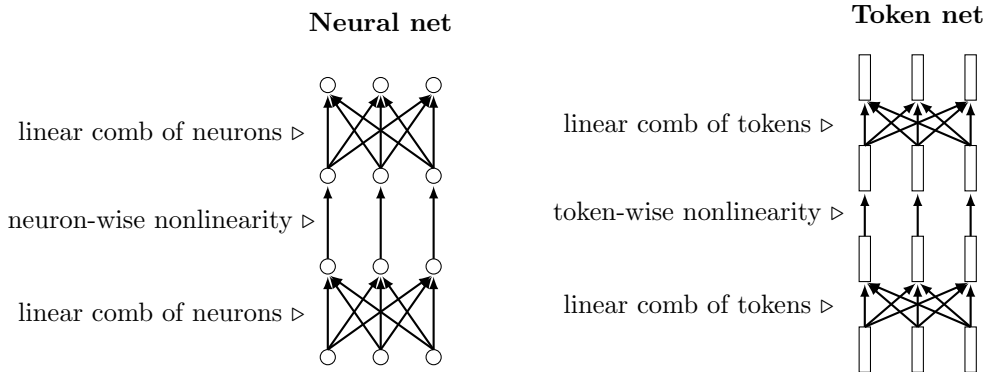
$$\mathbf{X}_{\text{out}}[:, k] = \sum_c \mathbf{W}_2[c, k] \star \mathbf{h} + \mathbf{b}_2[k] \quad \forall k \quad \triangleleft \text{conv} \quad (1.15)$$

This is such a fundamentally useful idea that it shows up in many different fields under different names. One general name for it is **factorizing** a problem into smaller pieces.

## 1.4 Token nets

We will use the term **token nets** to refer to computation graphs that use tokens as the primary nodes, rather than neurons. Token nets are just like neural nets, alternating between layers that mix nodes in linear combinations (e.g., fully-connected linear layers, convolutional layers, etc) and layers that apply a pointwise nonlinearity to each node (e.g., relus, per-token MLPs). Of course, since tokens are simply groups of neurons, every token net is itself also a neural net, just viewed differently – it is a net of sub-nets. Below we show a standard neural net and a token net side by side, to emphasize the similarities in their operations:

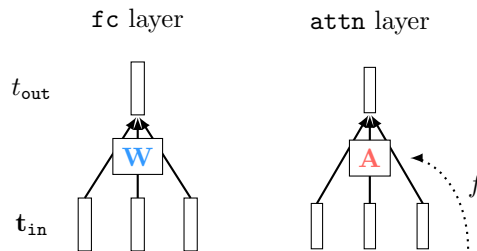
Note that the terminology in this chapter is not standard. The term “token nets”, and the token layer definitions we have given, are our own invention.



The arrows here represent any functional dependency between the nodes (note that different arrows represent different types of functions).

## 1.5 The attention layer

**Attention layers** define a special kind of linear combination of tokens. Rather than parameterizing the linear combination with a matrix of free parameters  $\mathbf{W}$ , attention layers use a different matrix, which we call the attention matrix  $\mathbf{A}$ . The important difference between  $\mathbf{A}$  and  $\mathbf{W}$  is that  $\mathbf{A}$  is *data-dependent*, that is, the values of  $\mathbf{A}$  are a function the data input to the network. In the diagram below, we indicate the data-dependency with the function labeled  $f$ , and we color the attention matrix red to indicate that it is constructed from *transformed data* rather than being free parameters (for which we use the color blue):



Here we make the connection between attention and fc layers. You can also make the connection between attention and pooling layers. From that perspective attention is a kind of *dyanmic pooling*: it’s mean pooling but with a weighted average where the weights are dynamically decided based on the input data.

The equation for an attention layer is the same as for a linear layer except that the weights are a function of some other data (left unspecified for now but we will see concrete examples below):

$$\mathbf{A} = f(\dots) \quad \triangleleft \text{attention} \tag{1.16}$$

$$\mathbf{Z}_{\text{out}} = \mathbf{A}\mathbf{Z}_{\text{in}} \tag{1.17}$$

The key question, of course, is “what exactly is  $f$ ”? What inputs does  $f$  depend on and what is  $f$ ’s mathematical form? Before writing out the exact equations, we will start with the intuition:  $f$  is a function that determines how much “attention” to apply to each token

in  $\mathbf{t}_{in}$ ; since this layer is just a weighted combination of tokens  $f$  is simply determining the weights in this combination.  $f$  can depend on any number of input signals that tell the net what to pay attention to.

As a concrete example, consider that we want to be able to ask questions about different objects in an image, such as “what color is the bird’s head?” Then we can use attention to direct the model to focus on just the object in question – the bird’s head in this example.  $f$  would take as input the text query, and would produce as output weights  $\mathbf{A}$  that are high for the  $\mathbf{t}_{in}$  tokens that correspond to any bird head’s and are low for all other  $\mathbf{t}_{in}$  tokens. If we train such a system to answer questions about color, then the token codes might end up representing the color of the object in their receptive field; after all, this would be a solution that would solve our problem (it would minimize the loss and correctly answer the question). Other solutions might be possible, but we will focus on this intuitive solution.

What’s neat here is that attention gives us a way to make the layer dynamically change its behavior in response to different input questions; asking different questions results in different answers:

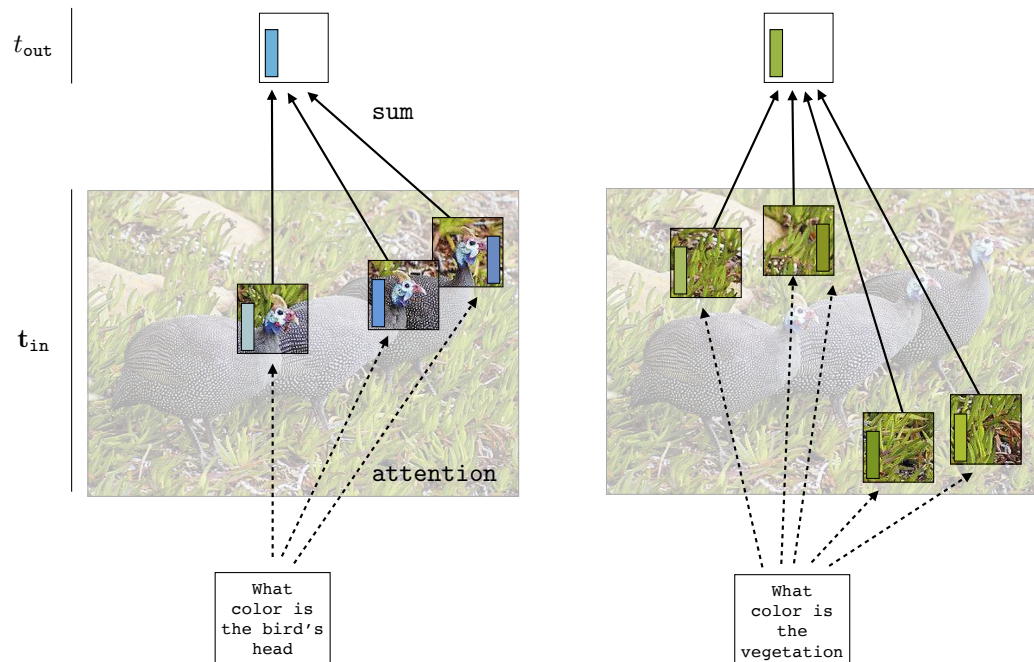


Figure 1.1: How attention can be allocated across different regions (tokens) in an image. The token codes are indicated as the colored rectangles within each token.  $t_{out}$  is a weighted sum over tokens in  $\mathbf{t}_{in}$ , weighted by attention. Only the tokens that contribute most to this sum are visualized here. On the left, the tokens corresponding to the birds’ heads are attended to, whereas on the right, tokens in the background are attended to.

Keeping this intuitive picture in mind, we will now turn to the equations that define  $f$ . We will focus on the particular version of  $f$  that appears in transformers, which is called **query-key-value attention**.

### 1.5.1 Query-Key-Value attention

Transformers use a particular kind of attention based on the idea of keys, queries, and values. In query-key-value attention, each token is associated with a **query** vector, a **key** vector, and a **value** vector. Just like the token’s code vector, we can think of these vectors as additional members of the structure  $t$ . We define these vectors as linear transformations of the token’s

The idea of keys, queries, values comes from databases, where a database cell holds a *value*, which is retrieved when a *query* matches the cell’s *key*. Tokens are like database cells and attention is like retrieving information from the database of tokens.

code vector. For a token with code vector  $\mathbf{z}$ , we have:

$$\mathbf{q} = t.\text{query}() = \mathbf{W}_q \mathbf{z} \quad \triangleleft \text{query} \tag{1.18}$$

$$\mathbf{k} = t.\text{key}() = \mathbf{W}_k \mathbf{z} \quad \triangleleft \text{key} \tag{1.19}$$

$$\mathbf{v} = t.\text{value}() = \mathbf{W}_v \mathbf{z} \quad \triangleleft \text{value} \tag{1.20}$$

The queries, keys, and values of  $\mathbf{t}_{\text{in}}$  can compactly be written as matrices:

$$\mathbf{Q}_{\text{in}} = \begin{pmatrix} \mathbf{q}_1^T \\ \vdots \\ \mathbf{q}_N^T \end{pmatrix} \quad \mathbf{K}_{\text{in}} = \begin{pmatrix} \mathbf{k}_1^T \\ \vdots \\ \mathbf{k}_N^T \end{pmatrix} \quad \mathbf{V}_{\text{in}} = \begin{pmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_N^T \end{pmatrix} \tag{1.21}$$

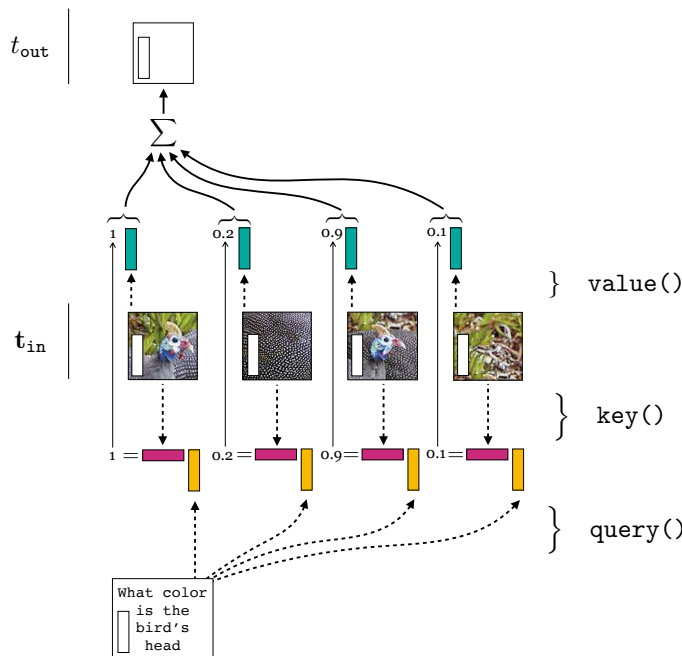
In transformers, all inputs to the net are tokenized, so the textual question “What color is the bird’s head” will also be represented as a token. This token will submit its query vector,  $\mathbf{q}_{\text{question}}$  to be matched against the keys of the tokens that represent different patches in the image; the similarity between the query and the key determines the amount of attention weight that query will apply to the token with that key. The most common measure of similarity between a query  $\mathbf{q}$  and a key  $\mathbf{v}$  is the dot product  $\mathbf{q}^T \mathbf{v}$ . Querying each token in  $\mathbf{t}_{\text{in}}$  in this way gives us a vector of similarities. We then normalize this vector using the softmax function to give us our attention weights  $\mathbf{A}$ , and finally, rather than applying  $\mathbf{A}$  over token codes directly (i.e. taking a weighted sum over tokens), we take a weighted sum over token values to obtain  $\mathbf{Z}_{\text{out}}$ :

$$\mathbf{s} = [\mathbf{q}_{\text{question}}^T \mathbf{k}_1, \dots, \mathbf{q}_{\text{question}}^T \mathbf{k}_N] \tag{1.22}$$

$$\mathbf{A} = \text{softmax}(\mathbf{s}) \tag{1.23}$$

$$\mathbf{Z}_{\text{out}} = \mathbf{A} \mathbf{V}_{\text{in}} \tag{1.24}$$

Figure 1.2 visualizes these steps:



Question to think about: could you use other differentiable functions for `query()`, `key()`, and `value()`? Would that be useful?

We do not cover them in this book but methods from natural language processing can be used to transform text into a token, or a sequence of tokens.

We use the following color scheme here and later in this chapter:

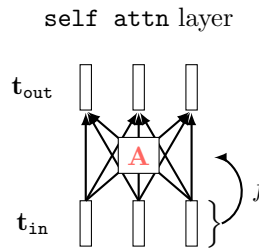


Figure 1.2: Mechanics of an attention layer. Queries from the question match keys from the tokens representing bird heads; value vectors of these two tokens then contribute the most to the sum that yields  $t_{\text{out}}$ 's code vector. (Softmax omitted in this example.)

### 1.5.2 Self-attention

As we have now seen, attention is a general-purpose way of dynamically pooling information in one set of tokens based on queries from a different set of tokens. The next question we will consider is: “which tokens should be doing the querying and which should we be matching against?” In the example from the last section, the answer was intuitive because we had a textual question that was asking about content in a visual image, so naturally the text gives the query and we match against tokens that represent the image. But can we come up with a more generic architecture where we don’t have to hand design which tokens interact in which ways?

**Self-attention** is just such an architecture. The idea is that on a self-attention layer, *all* tokens submit queries, and for each of these queries, we take a weighted sum over *all* tokens in that layer. If  $\mathbf{t}_{in}$  is length  $N$  then we have  $N$  queries,  $N$  weighted sums, and  $N$  output tokens to form  $\mathbf{t}_{out}$ :



The equations for self-attention can be written in an especially compact form:

$$\mathbf{Q}_{in} = \mathbf{Z}_{in} \mathbf{W}_q \quad \triangleleft \quad \text{query matrix} \quad (1.25)$$

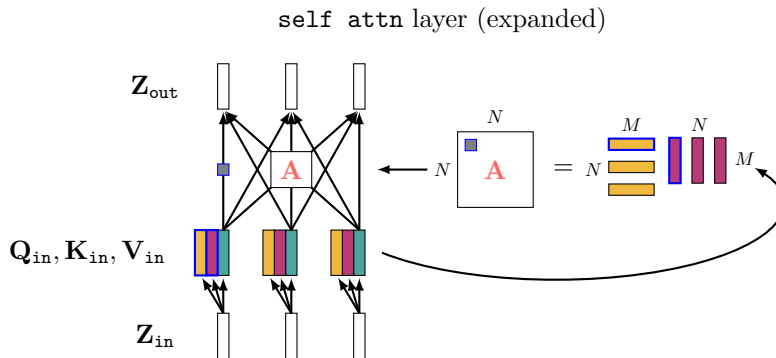
$$\mathbf{K}_{in} = \mathbf{Z}_{in} \mathbf{W}_k \quad \triangleleft \quad \text{key matrix} \quad (1.26)$$

$$\mathbf{V}_{in} = \mathbf{Z}_{in} \mathbf{W}_v \quad \triangleleft \quad \text{value matrix} \quad (1.27)$$

$$\mathbf{A} = f(\mathbf{t}_{in}) = \text{softmax}\left(\frac{\mathbf{Q}_{in} \mathbf{K}_{in}^T}{\sqrt{d}}\right) \quad \triangleleft \quad \text{attention matrix} \quad (1.28)$$

$$\mathbf{Z}_{out} = \mathbf{A} \mathbf{V}_{in} \quad (1.29)$$

$d$  is dimensionality of the query/key vector (since we take a dot product between query and key their dimensionalities must match). In expanded detail, here are the full mechanics of an attention layer:



The nodes outlined in blue correspond to each other; they represent one query being matched against one key to result a scalar similarity value, in the gray box, which acts as a weight in the weighted sum computed by  $\mathbf{A}$ .

This fully defines a self-attention layer, which is the kind of attention layer used in transformers. Before we move on though, let’s think through the intuition of what self-attention might be doing.

Consider that we are processing the Guineafowl image and our task is semantic segmentation (label each patch with an object class). First, we tokenize the image so that each



patch is represented by a token. Now we have a token,  $t_1$ , that represents the patch of pixels around of the birds' heads. We wish to update this token via one layer of self-attention. Since the goal of the network is to classify patches, it would make sense to update  $t_1$  to get a better semantic representation of what's going on in that patch. One way to do this would be to attend to the tokens representing the other bird heads, and use them to refine  $t_1$ . The intuition is that it's easier to recognize an object given three views of it (the three tokens representing bird heads). The refinement operation is just to sum over the token code vectors, which has the effect of reducing “noise” that is not shared between the three tokens and amplifying the commonalities between them. Figure 1.3 illustrates this scenario.

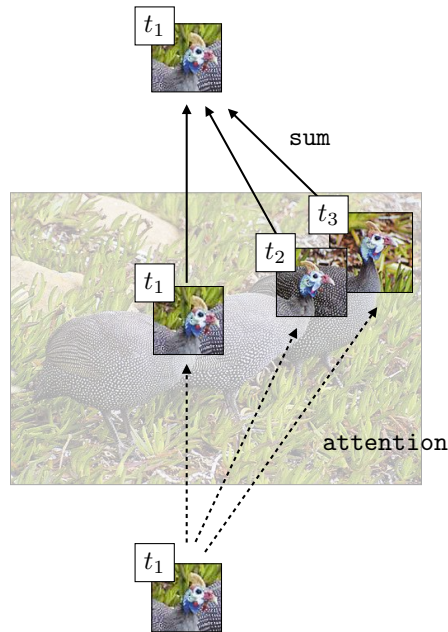


Figure 1.3: One way self-attention could be used to aggregate information across all patches containing the same object, and thereby arrive at a better representation of this object.

This is just one way self-attention could be used by the network. How it is actually used will be determined by the training data and task. What really happens might deviate from our intuitive story: tokens on hidden layers do not necessarily represent spatially localized patches of pixel. While the initial tokenization converts patches to pixels, after this point attention layers can mix information across spatially distant tokens –  $t_{\text{out}1}$  does not necessarily represent the same spatial region in the image as  $t_{\text{in}1}$ .

In fact, attention layers are **permutation equivariant**:

$$\text{attn}(\text{permute}(\mathbf{t}_{\text{in}})) = \text{permute}(\text{attn}(\mathbf{t}_{\text{in}})) \quad (1.30)$$

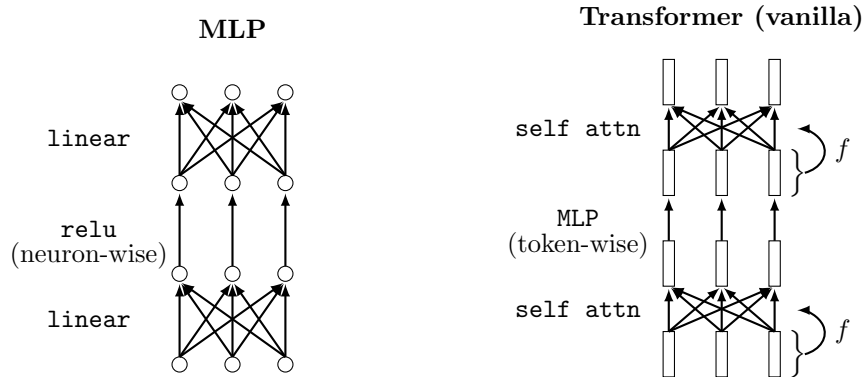
where **permute** is a permutation of the indices of the vector  $\mathbf{t}_{\text{in}}$ . This means that if you scramble (i.e. permute) the patches in the input image then apply attention, the output will be unchanged up to a permutation of the original output. It is often useful to understand layers in terms of their invariances and equivariances. Convolutional layers are translation equivariant but not necessarily permutation equivariant whereas attention layers are both translation equivariant *and* permutation equivariant (since translation is a special kind of permutation, any permutation equivariant layer is also translation equivariant). Note, however, that 1x1 convolutions are a special case of convolution that is in fact permutation equivariant, because they operate pointwise. Other layers can be catalogued similarly: global average pooling layers are permutation *invariant*, relu layers are permutation equivariant, per-token MLP layers are also permutation equivariant (but w.r.t. vectors of tokens

rather than vectors of neurons), and so on. As we will see below, transformers only use layers that are permutation equivariant, so the entire transformer architecture is permutation equivariant over its input tokens.

A generally good strategy is to select layers that reflect the symmetries in your data domain or task: in object detection, translation equivariance makes sense because, roughly, a bird is a bird no matter where it appears in an image. Permutation equivariance might also make sense, for that same reason, but only to an extent: if you break up an image into small patches and scramble them then this could disrupt spatial layout that is important for recognition. We will see in Section 1.7 how transformers use something called positional codes to re-insert useful information about spatial layout.

## 1.6 The full transformer architecture

A full transformer architecture is a stack of self-attention layers interleaved with token-wise nonlinearities. These two steps are analogous to linear layers interleaved with neuron-wise nonlinearities in an MLP:



Beyond this basic template, there are many variations that can be added, resulting in different particular architectures within the transformer family. Some common additions are normalization layers and residual connections.

### 1.6.1 Multihead self-attention

Additionally, it is common to use **multihead self-attention**, or **MSH**, which simply consists of running  $k$  attention layers in parallel, applied to the same input  $\mathbf{t}_{\text{in}}$ , then concatenating all the outputs, and finally projecting back to the original dimensionality of  $\mathbf{t}_{\text{in}}$ :

$$\mathbf{Z} = \begin{pmatrix} \text{attn}_1(\mathbf{t}_{\text{in}}) \cdot \mathbf{z}^T \\ \vdots \\ \text{attn}_k(\mathbf{t}_{\text{in}}) \cdot \mathbf{z}^T \end{pmatrix} \quad (1.31)$$

$$\mathbf{t}_{\text{out}} \cdot \mathbf{z} = \mathbf{WZ} \quad \triangleleft \quad \mathbf{W} \in \mathbb{R}^{M_2 \times kM_1} \quad (1.32)$$

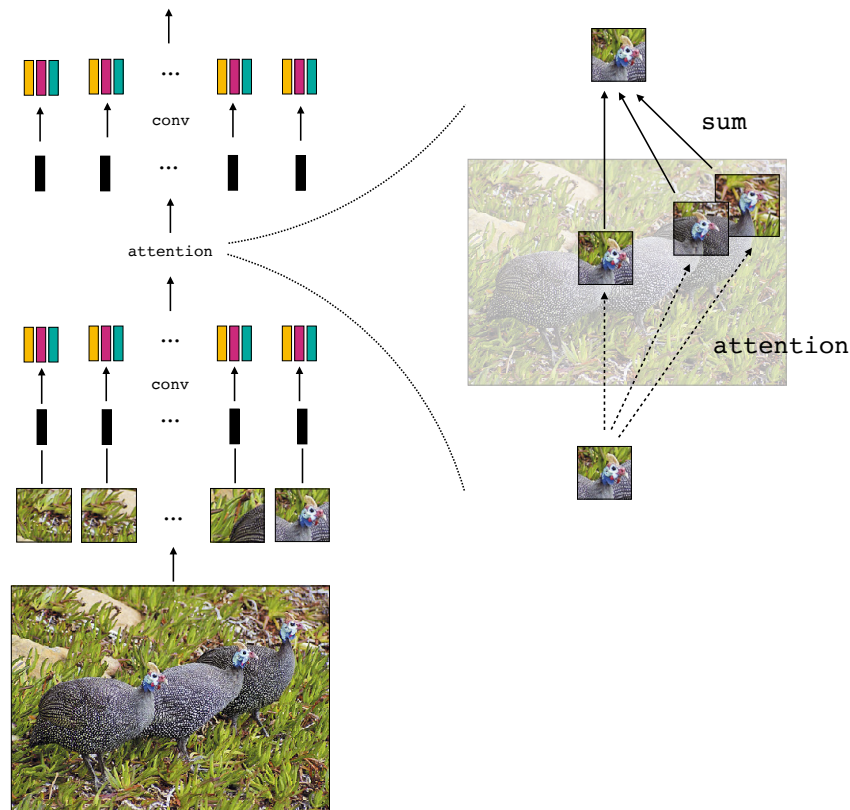
$\mathbf{W}$  are learnable parameters of this layer (in addition to the query, key, and value projections parameters for each of the  $k$  attention heads),  $M_1$  is the dimensionality of the value vectors and  $M_2$  is the dimensionality of the code vectors of the output ([Dosovitskiy et al. 2021] recommends setting  $kM_1 = M_2$ ).

### 1.6.2 Input and output modules

The transformer also has an input and output module. The input module is the tokenization layer that converts the input signal into a set of tokens. The output module converts the

transformed tokens into a target prediction or decision. The input and output modules are specific to the type of input signal and the type of output task.

Here is what the transformer looks like as a whole: Notice that most of the operations



are familiar from the regular CNNs from Chapter ???. As discussed in Section 1.3.3, the per-token MLP layers are equivalent to CNNs with  $1 \times 1$  spatial kernels. By a similar argument, it can be shown that the query, key, and value functions are equivalent to  $1 \times 1$  convs over token code vectors (we leave this as an exercise to the reader). Normalization layers, softmax layers, and residual connections also appear in CNNs. The main novelty of transformers is the self-attention layer. These layers look like fc-layers but are importantly different in two ways:

1. They operate over tokens rather than neurons.
2. The transformation parameters are a function of the input data.

Therefore, we can see that transformers are intimately connected to both MLPs and CNNs, but differ from both in important ways.

## 1.7 Positional encodings

Another idea associated with transformers is **positional encoding**. Operations over tokens in a transformer are permutation equivariant, which means that we can shuffle the positions of the tokens and nothing substantial changes (the only change is that the outputs get permuted). A consequence is that tokens do not naturally encode their position within the representation of the signal. Sometimes we may wish to retain positional knowledge, for example, knowing that a token is a representation of the top region of an image can help us identify that the token is likely to represent sky. Positional encoding concatenates a code representing *position within the signal* onto each token. If the signal is an image, then the positional code should represent the x and y coordinate. However, it need not represent these coordinates as scalars – more commonly we use a periodic representation of position,

where the coordinates are encoded as the vector of values a set of sinusoidal waves take on at each position:

$$\mathbf{p}_x = [\sin(x), \sin(x/B), \sin(x/B^2), \dots, \sin(x/B^P)] \quad (1.33)$$

$$\mathbf{p}_y = [\sin(y), \sin(y/B), \sin(y/B^2), \dots, \sin(y/B^P)] \quad (1.34)$$


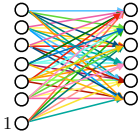
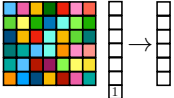

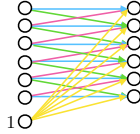
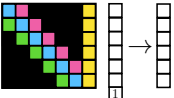

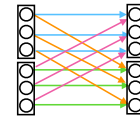

where  $x$  and  $y$  are the coordinates of the token.

While positional encoding is useful and common in transformers, it is not specific to this architecture. The same kind of encodings can be useful for CNNs as well, as a way to make convolutional filters that are conditioned on position, thereby applying a different weighted sum at each location in the image [Liu et al. 2018]. Positional encodings also appear in many graphics applications, e.g., [Mildenhall et al. 2020].

## 1.8 Comparing fc, conv, and attn

Many layers in deep nets are special kinds of affine transformations. Three we have seen so far are fc layers, conv layers, and self-attention layers. All these layers are alike in that their forward pass can be written as  $\mathbf{X}_{\text{out}} = \mathbf{W}\mathbf{X}_{\text{in}} + \mathbf{b}$  for some matrix  $\mathbf{W}$  and some vector  $\mathbf{b}$ . In conv and attn layers,  $\mathbf{W}$  and  $\mathbf{b}$  are determined as some function of the input  $\mathbf{X}_{\text{in}}$ . In conv layers this function is very simple: just make a Toeplitz matrix that repeats the convolutional kernel(s) to match the dimensionality of  $\mathbf{X}_{\text{in}}$ . In self-attention layers the function that determines  $\mathbf{W}$  is a bit more involved, as we saw above, and typically we don't use biases  $\mathbf{b}$ .

Each of these layers can be represented as a matrix, and examining the structure in these matrices can be a useful way to understand their similarities and differences. The matrix for an fc layer is full rank, whereas the matrices for conv and self-attention layers have low-rank structure, but different kinds of low-rank structure. Below we show what these matrices look like, and also catalogue some of the other important properties of each of these layers:

	Wiring graph	Matrix	Properties
			Fixed input dim $N^2$ learnable parameters
			Variable input dim $k$ learnable parameters ( $k = \text{kernel size}$ ) $\text{conv}(\text{translate}(\mathbf{x}_{\text{in}})) = \text{translate}(\text{conv}(\mathbf{x}_{\text{in}}))$
			Variable input dim $qkv$ learnable parameters (# params in query(), key(), and value()) $\text{attn}(\text{permute}(\mathbf{t}_{\text{in}})) = \text{permute}(\text{attn}(\mathbf{t}_{\text{in}}))$



# Bibliography

- A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *ICLR*, 2021.
- R. Liu, J. Lehman, P. Molino, F. P. Such, E. Frank, A. Sergeev, and J. Yosinski. An intriguing failing of convolutional neural networks and the coordconv solution. *arXiv preprint arXiv:1807.03247*, 2018.
- B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*, pages 405–421. Springer, 2020.
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- J. M. Wolfe. Visual attention. *Seeing*, pages 335–386, 2000.