26 Transformers

26.1 Introduction

Transformers are a recent family of architectures that generalize and expand the ideas behind convolutional neural nets (CNNs). The term for this family of architectures was coined by [487], where they were applied to language modeling. Our treatment in this chapter more closely follows the **vision transformers** (**ViTs**) that were introduced in [109].

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 dependencies between the chunks.

Transformers were originally introduced in the field of natural language processing, where they were used to model language, that is, sequences of characters and words. As a result, some texts present transformers as an alternative to recurrent neural nets (RNNs) for sequence modeling, but in fact transformer layers are *parallel* processing machines, like convolutional layers, rather than sequential machines, like recurrent layers.

26.2 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 stacking layers to increase the receptive field of neurons on deeper layers. Figure 26.1 shows the inability of a shallow CNN to compare two input nodes (x_1 and x_7) that are spatially too far apart:

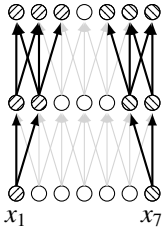

Figure 26.1: Consider a 2-layer CNN with kernel size 3, tasked to compare x_1 and x_7. It can't do it: there are no neurons that are connected to both x_1 and x_7.
Hatch marks indicate which neurons are connected to x_1 and x_7 respectively.

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?

26.3 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 [508]. 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 should direct their attention to the neurons on the previous layer 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.

26.4 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, and so on. 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 tokens whose internal content is a vector of neurons. A single token will therefore be represented by a column vector $\mathbf{t} \in \mathbb{R}^{d \times 1}$, which is also sometimes called the token's **code vector**.

26.4.1 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), or a lower-dimensional projection of the vectorized patch. With each patch represented by a token, the full image corresponds to an array of tokens. Figure 26.2 shows what it looks like to tokenize a safari image in this way.

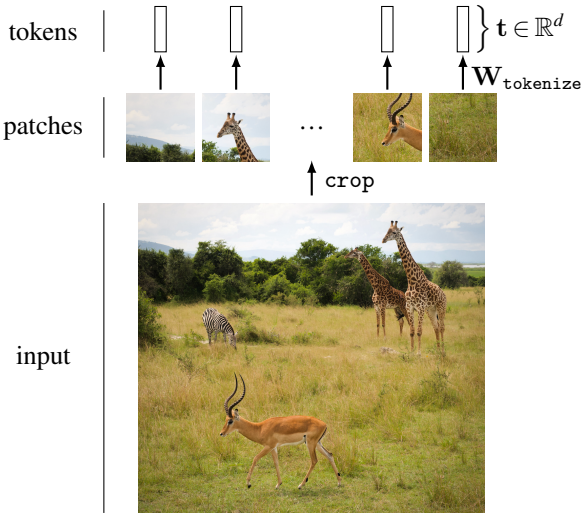

Figure 26.2: Tokenization: converting an image to a set of vectors. $\mathbf{W}_{\texttt{tokenize}}$ is a learnable linear projection from the dimensionality of the vectorized crops to d dimensions. This is just one of many possible ways to tokenize an image.

26.4.2 Data Structures and Notation for Working with Tokens

A sequence of tokens will be denoted by a matrix $\mathbf{T} \in \mathbb{R}^{N \times d}$, in which each token in the sequence, $\mathbf{t}_1, \ldots, \mathbf{t}_N$, is transposed to become a row of the matrix:

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}_1^\mathsf{T} \\ \vdots \\ \mathbf{t}_N^\mathsf{T} \end{bmatrix} \tag{26.1}$$

As we will see, transformers are invariant to permutations of the input sequence, so, as far as transformers are concerned, groups of tokens should be thought of a *sets* rather than ordered sequences.

Graphically, $\mathbf{T}$ is constructed from $\mathbf{t}_1, \ldots, \mathbf{t}_N$ like this (figure 26.3):

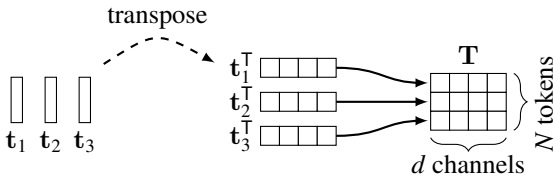

Figure 26.3: In this chapter, we will represent a set of tokens as a matrix whose rows are the token vectors.

The idea of this notation is that *tokens are to transformers as neurons are to neural nets*. Neural net layers operate over arrays of neurons; for example, an MLP takes as input a column vector $\mathbf{x}$, whose rows are scalar neurons. Transformers operate over arrays of tokens. A matrix $\mathbf{T}$ is just a convenient representation of 1D array of vector-value tokens.

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.

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).

26.4.3 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 operation we will define is how to take a *linear combination of tokens*.

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 vector-valued tokens (figure 26.4):

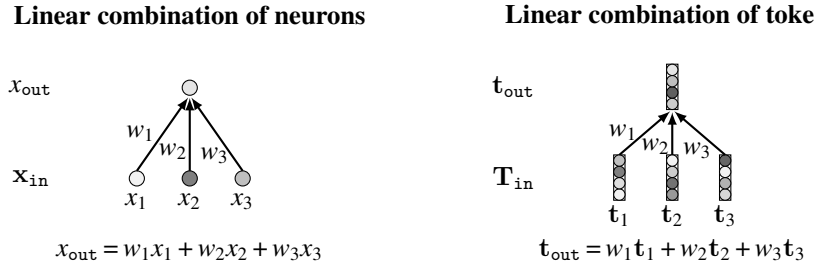

Figure 26.4: Linear combination of neurons versus tokens.

Notation: we use $\mathbf{T}_{\text{out}}$ when the output is multiple tokens and $\mathbf{t}_{\text{out}}$ when it is a single token.

The general form of these equations for multiple input and output neurons/tokens is:

$$x_{\text{out}}[i] = \sum_{j=1}^{N} w_{ij} x_{\text{in}}[j] \tag{26.2}$$

$$\mathbf{x}_{\text{out}} = \mathbf{W}\mathbf{x}_{\text{in}} \quad \triangleleft \text{ linear combination of neurons} \tag{26.3}$$

$$\mathbf{T}_{\text{out}}[i,:] = \sum_{j=1}^{N} w_{ij} \mathbf{T}_{\text{in}}[j,:] \tag{26.4}$$

$$\mathbf{T}_{\text{out}} = \mathbf{W}\mathbf{T}_{\text{in}} \quad \triangleleft \text{ linear combination of tokens} \tag{26.5}$$

Our notation here, which represents a set of tokens as a matrix $\mathbf{T}$, transforms working with tokens into an exercise in matrix algebra. However, this notation 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 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.

As can be seen above, 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 encountered in previous chapters can be defined for tokens in an analogous way to how they were defined for neurons.

For example, we can define a fully connected layer (fc layer) over tokens 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, by a set of token biases $\mathbf{b} \in \mathbb{R}^{N_2 \times d}$):

$$\mathbf{T}_{\text{out}} = \mathbf{W}\mathbf{T}_{\text{in}} + \mathbf{b} \quad \triangleleft \text{ fc layer over tokens} \tag{26.6}$$

26.4.4 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 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 networks of tokens we will introduce *tokenwise* operators; these are functions that apply a nonlinear transformation to each *token* individually, independently from all other tokens. Given a nonlinear function $F_\theta : \mathbb{R}^N \to \mathbb{R}^N$, a tokenwise nonlinearity layer, taking input $\mathbf{T}_{\texttt{in}}$, can be expressed as:

$$\mathbf{T}_{\texttt{out}} = \begin{bmatrix} F_\theta(\mathbf{T}_{\texttt{in}}[0,:]) \\ \vdots \\ F_\theta(\mathbf{T}_{\texttt{in}}[N-1,:]) \end{bmatrix} \quad \triangleleft \quad \text{per-token nonlinearity} \tag{26.7}$$

Notice that this operation is generalization of the pointwise nonlinearity in regular neural nets; a relu layer is the special case where $F_\theta = \texttt{relu}$ and the layer operates over a set of neuron inputs (scalars) rather than token inputs (vectors):

$$\mathbf{x}_{\texttt{out}} = \begin{bmatrix} \texttt{relu}(x_{\texttt{in}}[0]) \\ \vdots \\ \texttt{relu}(x_{\texttt{in}}[N-1]) \end{bmatrix} \quad \triangleleft \quad \text{per-neuron nonlinearity (}\texttt{relu}\text{)} \tag{26.8}$$

The F_θ may be any nonlinear function but some choices will work better than others. One popular choice is for F_θ to be a multilayer perceptron (MLP); see chapter 12. In this case, F_θ has learnable parameters θ, 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 neuronwise nonlinearities, have no learnable parameters, whereas F_θ typically does. This is one of the interesting things about working with tokens, the pointwise operations become expressive and parameter-rich.

26.5 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 subnets. In figure 26.5, 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 some of the definitions we have given, are our own invention.

26.6 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

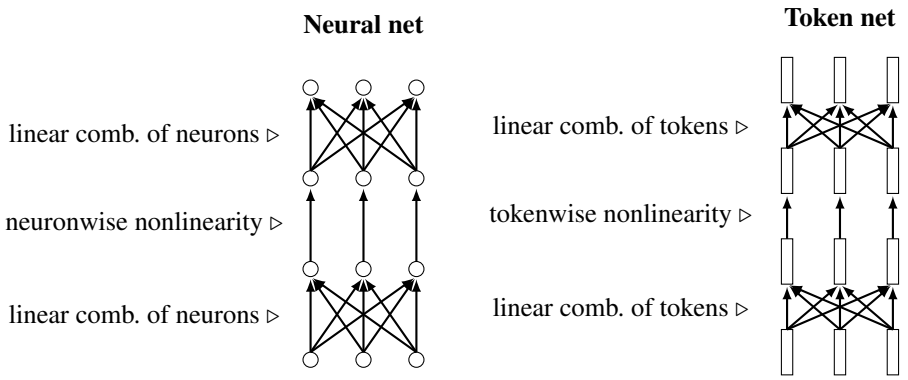

Figure 26.5: Neural nets versus token nets. The arrows here represent any functional dependency between the nodes (note that different arrows represent different types of functions).

Here, we describe attention as fc layers with data-dependent weights. We could have instead described attention as a kind of **dynamic pooling**, which is mean pooling but using a weighted average where the weights are dynamically decided based on the input data.

to the network. In addition, $\mathbf{A}$ typically only contains non-negative values, consistent with thinking of it as a matrix that allocates how much (non-negative) attention we pay to each input token. In the diagram below (figure 26.6), 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):

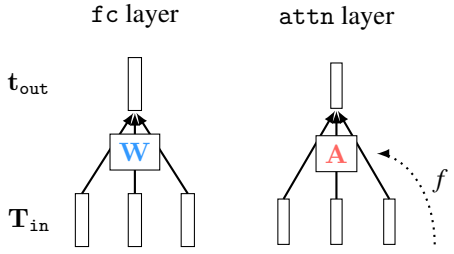

Figure 26.6: Fully-connected layers versus attention layers.

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 subsequently):

$$\mathbf{A} = f(\ldots) \qquad \triangleleft \text{ attention} \tag{26.9}$$

$$\mathbf{T}_{\texttt{out}} = \mathbf{A}\mathbf{T}_{\texttt{in}} \tag{26.10}$$

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 $\mathbf{T}_{\texttt{in}}$; because this layer is just a weighted combination of tokens, f is simply determining the weights in this combination. The 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 our safari example image, such as how many animals are in the photo. Then

one strategy would be to attend to each token that represents an animal's head, and then just count them up. The f would take as input the text query, and would produce as output weights $\mathbf{A}$ that are high for the $\mathbf{T}_{\texttt{in}}$ tokens that correspond to any animal's head and are low for all other $\mathbf{T}_{\texttt{in}}$ tokens. If we train such a system to answer questions about counting animals, then the token code vectors might naturally end up encoding a feature that represents the number of animal heads 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, which we illustrate in figure 26.7.

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, as is visualized below in figure 26.7.

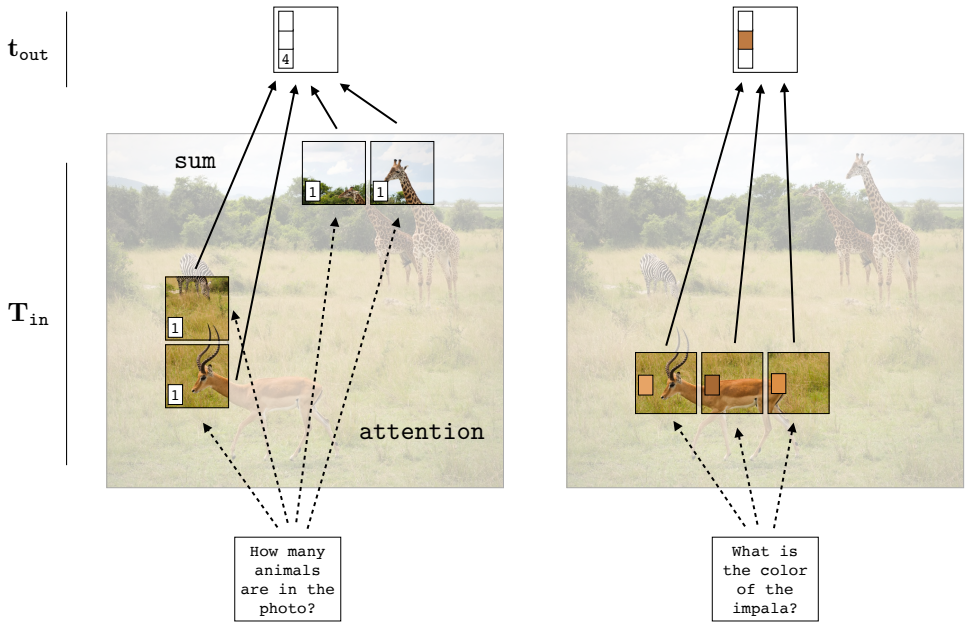

Figure 26.7: How attention can be allocated across different regions (tokens) in an image. The token code vectors consist of multiple dimensions and each can encode a different attribute of the token. To the left we show a dimension that encodes number of animal heads. To the right we show a different dimension that encodes color (or this could be three dimensions, coding RGB). The output token is a weighted sum over all the tokens attended to.

Let's walk through the logic of figure 26.7. Here we are imagining a token representation that can answer two different kinds of questions, one about number and the other about color. The representation we have come up with (which learning could have arrived at) is to encode in one dimension of the token vector a constant of value 1, which will be used for counting up the number of attended tokens. In another set of dimensions we have the average RGB color of the patch the token represents. Note that tokens only directly represent image patches at the input to the network, right after the tokenization step; at deeper layers of the network, the tokens may be more abstract in what they represent. Each text query elicits a different allocation of attention, and we will get to exactly how that process works later. For now just consider that the text query assigns a scalar weight to each

token depending on how well that token's content matches the query's content. The output token, $\mathbf{t}_{\texttt{out}}$, is the sum of all the tokens weighted by the attention scalars. This scheme will arrive at a reasonable answer to the questions if the text query "How many animals are in this photo" gives attention weight 1 to just the tokens representing animal heads and the text query "What is the color of the impala" gives weight $\frac{1}{3}$ just to the impala tokens. Then the output vector in the former case contains the correct answer 4 in the dimension that represents number of attended tokens, and contains the RGB values for brownish in the dimensions that represent average patch color.

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

26.6.1 Query-Key-Value Attention

The idea of queries, keys, and 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.

Transformers use a particular kind of attention based on the idea of queries, keys, and values. In query-key-value attention, each token is associated with a **query** vector, a **key** vector, and a **value** vector.

We define these vectors as linear transformations of the token's code vector, projecting to query/key/value vectors of length m. For a token $\mathbf{t}$, we have:

$$\mathbf{q} = \mathbf{W}_q \mathbf{t} \qquad \triangleleft \text{ query} \tag{26.11}$$

$$\mathbf{k} = \mathbf{W}_k \mathbf{t} \qquad \triangleleft \text{ key} \tag{26.12}$$

$$\mathbf{v} = \mathbf{W}_v \mathbf{t} \qquad \triangleleft \text{ value} \tag{26.13}$$

Here is a question to think about: Could you use other differentiable functions to compute the query, value, and key? 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 into a sequence of tokens.

In transformers, all inputs to the net are tokenized, so the textual question "How many animals are in the photo?" will also be represented as a token. This token will submit its query vector, $\mathbf{q}_{\texttt{question}}$ 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 the query will apply to the token with that key. The most common measure of similarity between a query $\mathbf{q}$ and a key $\mathbf{k}$ is the dot product $\mathbf{q}^\mathsf{T}\mathbf{k}$. Querying each token in $\mathbf{T}_{\texttt{in}}$ in this way gives us a vector of similarities:

$$\mathbf{s} = [s_1, \ldots, s_N]^\mathsf{T} = [\mathbf{q}_{\texttt{question}}^\mathsf{T}\mathbf{k}_1, \ldots, \mathbf{q}_{\texttt{question}}^\mathsf{T}\mathbf{k}_N]^\mathsf{T} \tag{26.14}$$

We then normalize the vector $\mathbf{s}$ using the softmax function to give us our attention weights $\mathbf{a} \in \mathbb{R}^{N\times 1}$, 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 value vectors, to obtain $\mathbf{T}_{\texttt{out}}$:

$$\mathbf{a} = \texttt{softmax}(\mathbf{s}) \tag{26.15}$$

$\mathbf{v}_1$ is the value vector for $\mathbf{t}_1 = \mathbf{T}_{\texttt{in}}[0, :]$, and so forth.

$$\mathbf{T}_{\texttt{out}} = \begin{bmatrix} a_1\mathbf{v}_1^\mathsf{T} \\ \vdots \\ a_N\mathbf{v}_N^\mathsf{T} \end{bmatrix} \tag{26.16}$$

Figure 26.8 visualizes these steps.

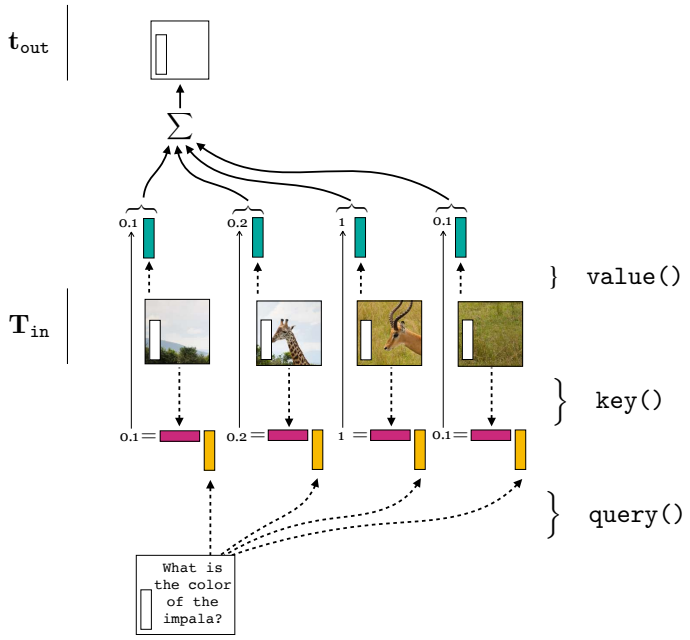

Figure 26.8: Mechanics of an attention layer. Queries from the question match keys from the tokens representing the impala; value vectors of the impala tokens then contribute the most to the sum that yields $\mathbf{t}_{\texttt{out}}$'s code vector. (Softmax omitted in this example.)

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

26.6.2 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 $\mathbf{T}_{\texttt{in}}$ is a set of N input tokens, then we have N queries, N weighted sums, and N output tokens to form $\mathbf{T}_{\texttt{out}}$. This is visualized below in figure 26.9.

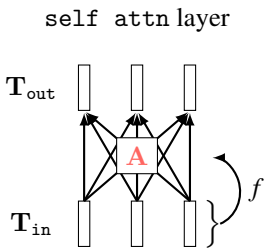

Figure 26.9: A self-attention layer.

To compute the query, key, and value for a set of input tokens, $\mathbf{T}_{\texttt{in}}$, we apply the same linear transformations to each token in the set, resulting in matrices $\mathbf{Q}_{\texttt{in}}, \mathbf{K}_{\texttt{in}} \in \mathbb{R}^{N \times m}$ and $\mathbf{V}_{\texttt{in}} \in \mathbb{R}^{N \times d}$, where each row is the query/key/value for each token:

Note that the query and key vectors must have the same dimensionality, m, because we take a dot product between them. Conversely, the value vectors must match the dimensionality of the token code vectors, d, because these are summed up to produce the new token code vectors.

$$\mathbf{Q}_{\texttt{in}} = \begin{bmatrix} \mathbf{q}_1^\mathsf{T} \\ \vdots \\ \mathbf{q}_N^\mathsf{T} \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_q \mathbf{t}_1)^\mathsf{T} \\ \vdots \\ (\mathbf{W}_q \mathbf{t}_N)^\mathsf{T} \end{bmatrix} = \mathbf{T}_{\texttt{in}} \mathbf{W}_q^\mathsf{T} \quad \triangleleft \quad \text{query matrix} \tag{26.17}$$

$$\mathbf{K}_{\texttt{in}} = \begin{bmatrix} \mathbf{k}_1^\mathsf{T} \\ \vdots \\ \mathbf{k}_N^\mathsf{T} \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_k \mathbf{t}_1)^\mathsf{T} \\ \vdots \\ (\mathbf{W}_k \mathbf{t}_N)^\mathsf{T} \end{bmatrix} = \mathbf{T}_{\texttt{in}} \mathbf{W}_k^\mathsf{T} \quad \triangleleft \quad \text{key matrix} \tag{26.18}$$

$$\mathbf{V}_{\texttt{in}} = \begin{bmatrix} \mathbf{v}_1^\mathsf{T} \\ \vdots \\ \mathbf{v}_N^\mathsf{T} \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_v \mathbf{t}_1)^\mathsf{T} \\ \vdots \\ (\mathbf{W}_v \mathbf{t}_N)^\mathsf{T} \end{bmatrix} = \mathbf{T}_{\texttt{in}} \mathbf{W}_v^\mathsf{T} \quad \triangleleft \quad \text{value matrix} \tag{26.19}$$

Finally, we have the attention equation:

$$\mathbf{A} = f(\mathbf{T}_{\texttt{in}}) = \texttt{softmax}\left(\frac{\mathbf{Q}_{\texttt{in}} \mathbf{K}_{\texttt{in}}^\mathsf{T}}{\sqrt{m}}\right) \quad \triangleleft \quad \text{attention matrix} \tag{26.20}$$

$$\mathbf{T}_{\texttt{out}} = \mathbf{A} \mathbf{V}_{\texttt{in}} \tag{26.21}$$

where the softmax is taken within each row (i.e., over the vector of matches for each separate query vector, like in equation (26.14)). In expanded detail, here are the full mechanics of a self-attention layer (figure 26.10):

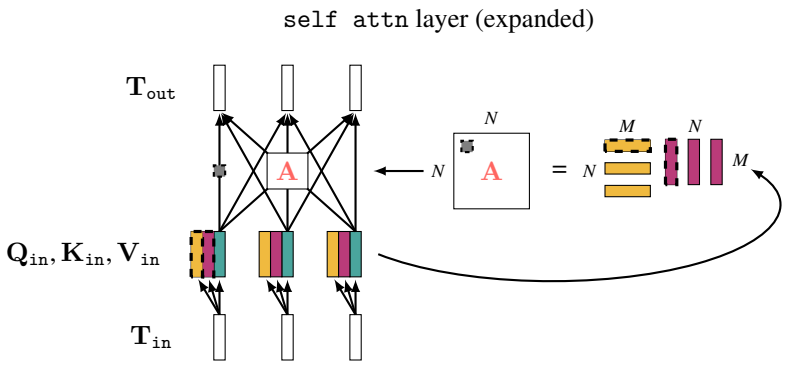

Figure 26.10: Self-attention layer expanded. The nodes with the dashed outline correspond to each other; they represent one query being matched against one key to result in a scalar similarity value, in the gray box, which acts as a weight in the weighted sum computed by $\mathbf{A}$.

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 safari image, and our task is semantic segmentation (label each patch with an object class). Figure 26.11 illustrates this scenario. We start by tokenizing the image so that each patch is represented by a token. Now we have a token,

$\mathbf{t}_2$, that represents the patch of pixels around the torso of the impala. 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 $\mathbf{t}_2$ 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 other patches of the impala, and use them to refine $\mathbf{t}_2$ into a more abstracted token vector, capturing the label *impala*. The intuition is that it's easier to recognize a patch given the context of other relevant patches around it. 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 attended impala patches, which amplifies the commonality between them – the label *impala*. More sophisticated refinements could be achieved via multiple layers of self-attention. Further, the impala patch query could also retrieve information from the giraffe and zebra patches, as those patches provide additional context that could be informative (the animal in the query is more likely to be an impala if it is found near giraffes and zebras, since all those animals tend to congregate together in the same biome).

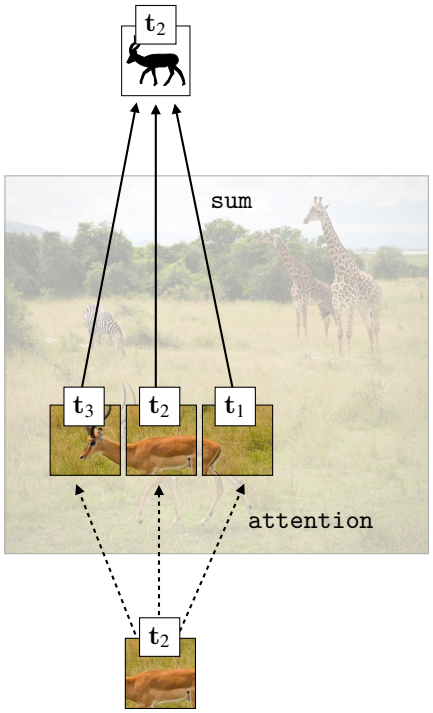

Figure 26.11: One way self-attention could be used to aggregate information across all patches containing the same object, and thereby arrive at a better representation of the object in $\mathbf{t}_2$, the query patch.

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 pixels. While the initial tokenization layer creates tokens out of local image patches, after this point attention layers can mix information across spatially distant tokens;

note that $\mathbf{T}_{\texttt{out}}[0,:]$ does not necessarily represent the same spatial region in the image as $\mathbf{T}_{\texttt{in}}[0,:]$.

Figure 26.12 gives an example of what self-attention maps can look like on the safari image. In this example, we are simply using patch color as the query and key features. Each attention map shows one row of $\mathbf{A}$ reshaped into the size of the input image.

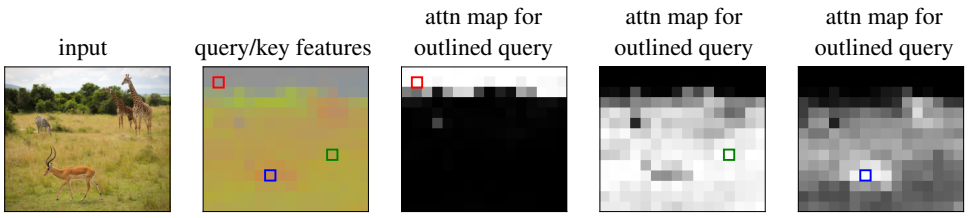

Figure 26.12: Example of self-attention maps where each token is an image patch and the query and key vectors are both set to the mean color of the patch, normalized to be a unit vector.

26.6.3 Multihead Self-Attention

Despite their power, self-attention layers are still limited in that they only have one set of query/key/value projection matrices (namely, $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$). These matrices define the notion of similarity that is used to match queries to keys. In particular, the similarity between two tokens i and j is measured as:

$$s_{ij} = \mathbf{q}_i^\top \mathbf{k}_j \tag{26.22}$$

$$= (\mathbf{W}_q \mathbf{t}_i)^\top \mathbf{W}_k \mathbf{t}_j \tag{26.23}$$

$$= \mathbf{t}_i^\top \mathbf{W}_q^\top \mathbf{W}_k \mathbf{t}_j \tag{26.24}$$

$$= \mathbf{t}_i^\top \mathbf{S} \mathbf{t}_j \tag{26.25}$$

What this shows is that $\mathbf{W}_q$ and $\mathbf{W}_k$ define some matrix $\mathbf{S} = \mathbf{W}_q^\top \mathbf{W}_k$ that modulates how we measure similarity (dot product) between $\mathbf{t}_i$ and $\mathbf{t}_j$. A single self-attention layer therefore measures similarity in just one way.

What if we want to measure similarity in more than one way? For example, maybe we want our net to perform some set of computations based on color similarity, another based on texture similarity, and yet another based on shape similarity? The way transformers can do this is with **multihead self-attention** (**MSA**). This method simply consists of running k attention layers in parallel. All these layers are applied to the same input $\mathbf{T}_{\texttt{in}}$. This results in k output sets of tokens, $\mathbf{T}^1_{\texttt{out}}, \ldots, \mathbf{T}^k_{\texttt{out}}$. To merge these outputs, we concatenate all of them and project back to the original dimensionality of $\mathbf{T}_{\texttt{in}}$. These steps are shown in the

math below:

$$\mathbf{T}^i_{\texttt{out}} = \texttt{attn}^i(\mathbf{T}_{\texttt{in}}) \quad \text{for } i \in \{1, \ldots, k\} \tag{26.26}$$

$$\bar{\mathbf{T}}_{\texttt{out}} = \begin{bmatrix} \mathbf{T}^1_{\texttt{out}}[0,:] & \ldots & \mathbf{T}^k_{\texttt{out}}[0,:] \\ \vdots & \vdots & \vdots \\ \mathbf{T}^1_{\texttt{out}}[N-1,:] & \ldots & \mathbf{T}^k_{\texttt{out}}[N-1,:] \end{bmatrix} \quad \triangleleft \quad \bar{\mathbf{T}}_{\texttt{out}} \in \mathbb{R}^{N \times kv} \tag{26.27}$$

$$\mathbf{T}_{\texttt{out}} = \bar{\mathbf{T}}_{\texttt{out}} \mathbf{W}_{\texttt{MSA}} \quad \triangleleft \quad \mathbf{W}_{\texttt{MSA}} \in \mathbb{R}^{kv \times d} \tag{26.28}$$

Notice that here, unlike in the single-headed self-attention layers presented previously, the value vectors need not have the same dimensionality as the token code vectors, since we are applying the projection equation (26.28).

where v is the dimensionality of the value vectors and d is the dimensionality of the code vectors of the output ([109] recommends setting $kv = d$). The matrix $\mathbf{W}_{\texttt{MSA}}$ *merges* all the heads; its values are learnable parameters. The other learnable parameters of MSA are the query, key, and value projections for each of the k attention heads.

The basic reasoning here is quite simple: if self-attention layers are a good thing, why not just add more of them? We can add more *sequential* self-attention layers by building deeper transformers, or we can add more *parallel* self-attention layers by using MSA.

26.7 The Full Transformer Architecture

A full transformer architecture is a stack of self-attention layers interleaved with tokenwise nonlinearities. These two steps are analogous to linear layers interleaved with neuronwise nonlinearities in an MLP, as shown below (figure 26.13):

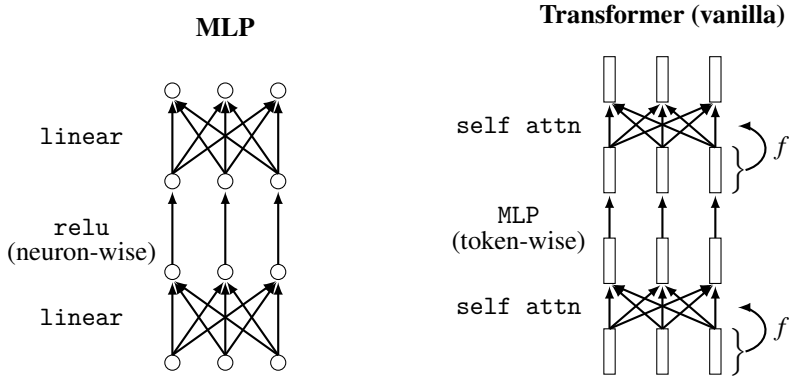

Figure 26.13: The basic transformer architecture versus an MLP.

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. In figure 26.14 we plot the ViT architecture from [109], showing where these additional pieces enter the picture.

This architecture uses layer normalization (section 12.7.3) before each attention layer and before each token-wise MLP layer. The normalization is done *within* each token (the token code vector is treated as a akin to a layer; each dimension of this vector is standardized by the mean and variance over all dimensions of this vector), so in figure 12.7.3 we refer

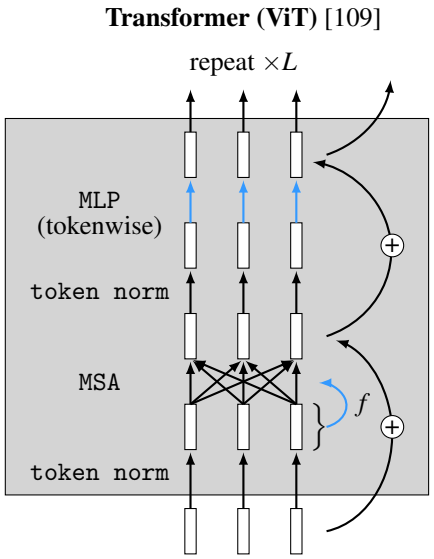

Figure 26.14: The ViT transformer architecture [109]. This set of layers forms a computational block, shaded in gray, that can be repeated L times for a depth L ViT. To clarify where the parameters live in this architecture, we have colored all the edges with learnable parameters in blue (note that the MSA merge, equation (26.28), is also learnable but not explicitly shown in this diagram).

to this layer as `token norm`. Notice that `token norm` is a tokenwise operation, just like our tokenwise MLP, but it performs a different kind of transformation and does not have learnable parameters. Residual connections are added around each group of layers.

Pseudocode for this a ViT (with single-headed attention) is given below:

```
# x : input data (RGB image)
# K : tokenization patch size
# d : token/query/key/value dimensionality (setting these all as the same)
# L : number of layers
# W_q_T, W_k_T, W_v_T : transposed query/key/value projection matrices
# mlp: tokenwise mlps

# tokenize input image
T = tokenize(x,K) # 3 x H x W image --> N x d array of token code vectors

# run tokens through all L layers
for l in range(L):

    # attention layer
    Q, K, V = nn.matmul(nn.layernorm(T),[W_q_T[l], W_k_T[l], W_v_T[l]])
    # nn.matmul does matrix multiplication
    A = nn.softmax(nn.matmul(Q,K.transpose()), dim=0)/sqrt(d)
    T = nn.matmul(A,V) + T # note residual connection

    # tokenwise mlp
    T = mlp[l](nn.layernorm(T)) + T # note residual connection

# T now contains the output token representation computed by the transformer
```

The output of a transformer, as we have so far defined it, is a set of tokens $\mathbf{T}_{\texttt{out}}$. Often we want an output of a different format, such as a single vector of logits for image classification (section 9.7.3), or in the format of an image for image-to-image tasks (section 34.6). To handle these cases, we typically define a task-specific output layer that takes $\mathbf{T}_{\texttt{out}}$ as input and produces the desired format as output. For example, to produce a vector of logit predictions we could first sum all the token code vectors in $\mathbf{T}_{\texttt{out}}$ and then, using a single linear layer, project the resulting d-dimensional vector into a K-dimensional vector (for K-way classification).

26.8 Permutation Equivariance

An important property of transformers is that they are equivariant to permutations of the input token sequence. This follows from the fact that both tokenwise layers, F_θ, and attention layers, `attn`, are **permutation equivariant**:

$$F_\theta(\texttt{permute}(\mathbf{T}_{\texttt{in}})) = \texttt{permute}(F_\theta(\mathbf{T}_{\texttt{in}})) \tag{26.29}$$

$$\texttt{attn}(\texttt{permute}(\mathbf{T}_{\texttt{in}})) = \texttt{permute}(\texttt{attn}(\mathbf{T}_{\texttt{in}})) \tag{26.30}$$

where `permute` is a permutation of the order of tokens in $\mathbf{T}_{\texttt{in}}$ (i.e., permutes the rows of the matrix). 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. Since the full transformer architecture is just composition of these two types of layers (plus, potentially, residual connections and token normalization, which are also permutation equivariant), and because composing two permutation equivariant functions results in a permutation equivariant operation, we have:

$$\texttt{transformer}(\texttt{permute}(\mathbf{T}_{\texttt{in}})) = \texttt{permute}(\texttt{transformer}(\mathbf{T}_{\texttt{in}})) \tag{26.31}$$

This property is visualized in figure 26.15.

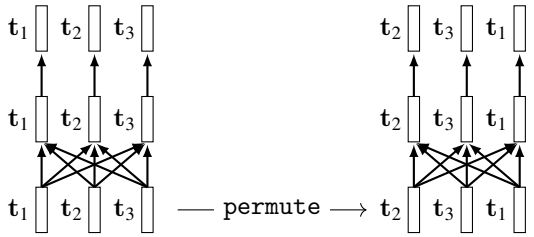

Figure 26.15: Transformers are permutation equivariant. For notational simplicity, we omit layer indices on the token variables here.

It is often useful to understand layers in terms of their invariances and equivariances. Convolutational 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). 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 with respect to sets of tokens rather than sets of neurons), and so on.

A generally good strategy is to select layers that reflect the symmetries in your data 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, this could disrupt spatial layout that is important for recognition. We will see in section 26.11 how transformers use something called positional codes to reinsert useful information about spatial layout.

26.9 CNNs in Disguise

Transformers provide a new way of thinking about data processing, and it may seem like they are very different from past architectures. However, as we have alluded to, they actually have many commonalities with CNNs. In fact, most (but not all) of the transformer architecture can be viewed as a CNN in disguise. In this section we will walk through several of the layers we learned about above, and see how they are in fact performing convolutions.

26.9.1 Tokenization

The first step in working with transformers is to tokenize the input. The most basic way to do this is to chop up the input image into non-overlapping patches of size $K \times K$, then convert these patches to vectors via a linear projection. You might already have noticed that this operation can be written as convolution; after all we said the whole idea of CNNs is to chop the signal into patches. In particular, this form of tokenization can be written as a convolutional layer with kernel size and stride both equal to K:

$$\mathbf{T}[n(N/K)+m, c_2] =$$

$$b[c_2] + \sum_{c_1=1}^{C_{\texttt{in}}} \sum_{k_1,k_2=-K}^{K} w[c_1, c_2, k_1, k_2] x_{\texttt{in}}[c_1, Kn-k_1, Km-k_2] \quad \triangleleft \quad \text{(tokenization)} \quad (26.32)$$

where, for RGB images, $\mathbf{x}_{\texttt{in}} \in \mathbb{R}^{3\times N\times M}$, $C_{\texttt{in}} = 3$, and $C_{\texttt{out}} = d$ (the token dimensionality). This math assumes N and M are evenly divisible by K; if they aren't then the input can be resized or padded until they are.

Although the equation starts to look complicated, it is just a `conv` operator with the following parameters:

```
T = conv(x_in, channels_in=3, channels_out=d, kernel=K, stride=K) # tokenize
```

26.9.2 Query-Key-Value Projections

Next let's look at the query, key, and value projections that are part of the attention layers. For simplicity, we will consider just the query projection, since key and value follow exactly the same pattern.

We wrote this operation as a matrix multiply $\mathbf{T}_{\texttt{in}}\mathbf{W}_q^\top$ (equation (26.17)). What this multiply is doing is applying the same linear transformation ($\mathbf{W}_q$) to each token vector (each

row of $\mathbf{T}_{\texttt{in}}$). Applying the same linear operation to each element in a sequence is exactly what convolution does. Specifically, the query operation can be written as convolving the set of N d-channel tokens with a filter bank of m filters, with kernel size 1, producing a new set of N m-channel tokens. This equivalence is visualized below (figure 26.16):

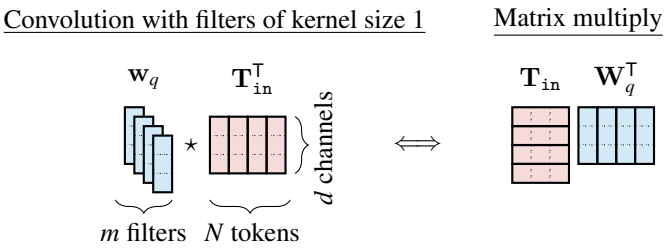

Figure 26.16: The query, key, and value projections in transformers can be written either as a convolution or a matrix multiply.

Convolution actually appears all over in linear algebra, and in fact *every matrix product can be written as a convolution*! Whenever you see a product $\mathbf{AB}$, you can think of it as the convolution of a multichannel filter bank $\mathbf{B}$ (one filter in each row; kernel size 1) with the signal $\mathbf{A}$ (time indexes rows, channels in the columns).

Therefore, the query, key, and value projections are all multichannel convolutions with kernels of size 1.

26.9.3 Tokenwise MLP

Next we will consider the token-wise MLP layer. A token-wise MLP applies the same MLP F_θ to each token in a sequence. The F_θ consists of linear layers and pointwise nonlinearities. For simplicity, we will assume no biases (as an exercise, this can be relaxed). The linear layers in F_θ all have the following form:

$$\mathbf{t}_{\texttt{out}} = \mathbf{W}\mathbf{t}_{\texttt{in}} \tag{26.33}$$

When we apply such a layer to each token in the sequence, we have:

$$\mathbf{T}_{\texttt{out}} = \mathbf{T}_{\texttt{in}}\mathbf{W}^\mathsf{T} \tag{26.34}$$

Notice that this looks just like the query operation we covered in the previous section, equation (26.17). Therefore, the same result holds: the linear layers of the token-wise MLP can all be written as convolutions with kernel size 1.

Now the pointwise nonlinearities in the MLP are applied neuronwise, so these layers function identically to the pointwise nonlinearity in CNNs. This is the full set of layers in the MLP, and therefore we have that a token-wise MLP can be written as a series of convolutions interleaved with neuronwise-nonlinearities, i.e. a CNN.

26.9.4 The Similarities between CNNs and Transfomers

Breaking up into chunks 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.

As we have now seen, most layers in transformers are convolutional. These layers break up the signal processing problem into chunks, then process each chunk independently and identically. Some of the other operations in transformers – normalization layers, residual connections, etc – are also common in CNNs. So what is *different* between transformers and CNNs?

26.9.5 The Differences between CNNs and Transformers

We use the term "1x1 filter" to refer to any filter whose kernel size is 1 in all its dimensions, regardless of whether the signal is one-dimensional, two-dimensional, three-dimensional, etc.

26.9.5.1 CNNs can have kernels with non-unitary spatial extent When we wrote them as convolutions, the query-key-value projections and token-wise MLPs *only used 1x1 filters*. In fact it cannot be otherwise. If you used larger kernel it would break the permutation invariance property of transformers, since the output of the filters would depend on which token is next to which. This is one of the key differences between CNNs and transformers. CNNs use $K \times K$ filters, and this makes it so adjacent image regions get processed together. Transformers use 1x1 filters which means the network has no architectural way of knowing about spatial structure (which token is next to which). For vision problems, where spatial structure is often crucially important, transformers can instead be given knowledge of position through the *inputs* to the network, rather than through the architectural structure. We will cover this idea in section 26.11.

26.9.5.2 Transformers have attention layers Attention layers are *not* convolutional. They do not factor the processing into independent chunks but instead perform a global operation, in which all input tokens can interact. The linear combination of tokens that results is not a mixing operation found in CNNs and addresses the limitation of CNNs being myopic, with each filter only seeing information in its receptive field.

26.10 Masked Attention

For simplicity, in this section we depict tokens with one-dimensional code vectors, but remember $\mathbf{T}$ would have d columns for d-dimensional code vectors.

The $\mathbf{T}_{1:n}$ is shorthand for the sequence of tokens $\begin{bmatrix} \mathbf{t}_1^T \\ \vdots \\ \mathbf{t}_n^T \end{bmatrix}$.

Sometimes we want to restrict which tokens can attend to which. This can be done by *masking* the attention matrix, which just means fixing some of the weights to be zero. This can be useful for many settings, including learning features via masked autoencoding [192], cross-attention between different data modalities [497], and for sequential prediction [78]. To illustrate, we will describe the sequential prediction use case.

A common problem is to predict the $(n+1)$-th token in a sequence given the previous n tokens. For example, we may be trying to predict tokens that represent the next frame in a video, the next word in a sentence, or the weather on the next day.

A simple way to model this prediction problem is with a linear layer: $\mathbf{y}_{n+1} = \mathbf{A}\mathbf{T}_{1:n}$. Here is what this looks like diagrammatically, and on the right is the layer shown as matrix multiplication:

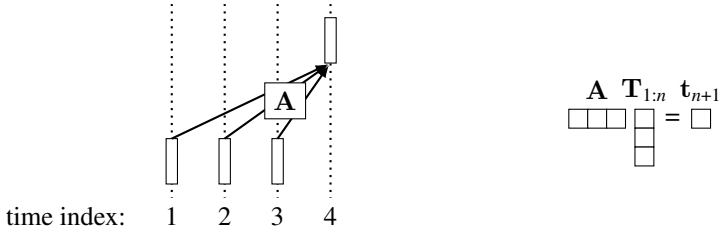

Figure 26.17: Masked prediction of time index 4 from time indices 1-3.

During training, we will give examples like this:

$$\{\mathbf{t}_1, \ldots, \mathbf{t}_n\} \rightarrow \mathbf{t}_{n+1} \tag{26.35}$$

$$\{\mathbf{t}_1, \ldots, \mathbf{t}_{n-1}\} \rightarrow \mathbf{t}_n \tag{26.36}$$

$$\{\mathbf{t}_1, \ldots, \mathbf{t}_{n-2}\} \rightarrow \mathbf{t}_{n-1} \tag{26.37}$$

and so on. We can make all these predictions at once with a single matrix multiply (figure 26.18):

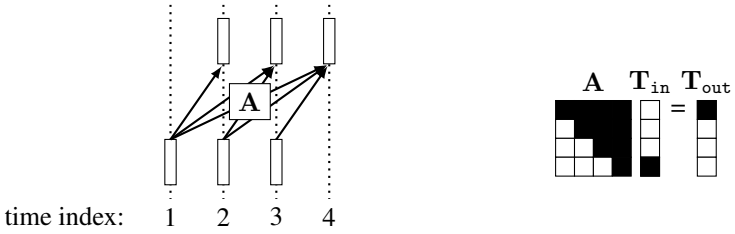

Figure 26.18: Masked attention to make multiple causal predictions at once. Black cells are masked; they are filled with zeros.

This way, one forward pass makes N predictions rather than one prediction. This is equivalent to doing a single next token prediction N times, but it all happens in a single matrix multiply, using the matrix shown on the right.

This kind of matrix is called *causal* because each output index i only depends on input indices j such that $j < i$. If $\mathbf{A}$ is an attention matrix, then this strategy is called **causal attention**. This is a masking strategy where each token can only attend to *previous* tokens in the sequence. This approach can dramatically speed up training because all the sub-sequence prediction problems (predict $\mathbf{t}_{n-1}$ given $\mathbf{T}_{1:n-2}$, predict $\mathbf{t}_n$ given $\mathbf{T}_{1:n-1}$, predict $\mathbf{t}_{n+1}$ given $\mathbf{T}_{1:n}$) are supervised at the same time.

This also works for transformers with more than one layer, where the masking strategy looks like shown in figure 26.19.

Notice that the output tokens on every layer l have the property that $\mathbf{t}_n^l$ only depends on $\mathbf{T}_{1:n-1}^0$, where $\mathbf{T}^0$ are the initial set of tokens input into the transformer. Also notice that, after the first layer, all subsequent layers can use causal attention that is not shifted in time, and the previous property is still maintained. Finally, notice that the subnetwork that predicts each subsequent output token overlaps substantially with the subnetworks that predict each previous token. That is, there is sharing of computation between all the prediction problems. We will see a more concrete application of this strategy when we get to autoregressive models in section 32.7.

26.11 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

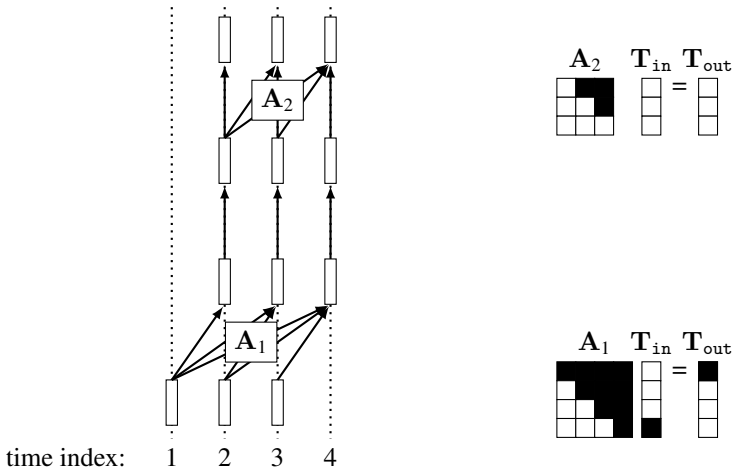

Figure 26.19: Multi-layer masked attention achieves causal prediction with a deep net.

Note that masked attention layers are not permutation invariant because which tokens get masked depends on their ordering. Because of this, masked attention models do not necessarily need positional encodings in order to become sensitive to position [190].

permuted). A consequence is that tokens do not know encode their position within the representation of the signal. Sometimes, however, 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-coordinates. 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:

$$\mathbf{p}_x = [\sin(x), \sin(x/B), \sin(x/B^2), \ldots, \sin(x/B^P)]^\mathsf{T} \tag{26.38}$$

$$\mathbf{p}_y = [\sin(y), \sin(y/B), \sin(y/B^2), \ldots, \sin(y/B^P)]^\mathsf{T} \tag{26.39}$$

$$\mathbf{p} = \begin{bmatrix} \mathbf{p}_x \\ \mathbf{p}_y \end{bmatrix} \tag{26.40}$$

where x and y are the coordinates of the token. This representation is visualized in figure 26.20:

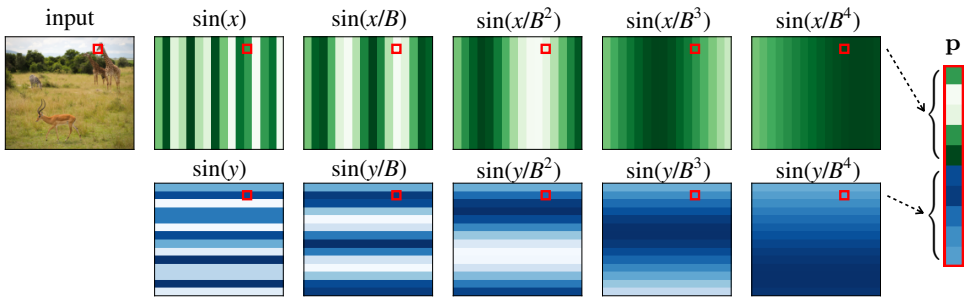

Figure 26.20: Positional codes.

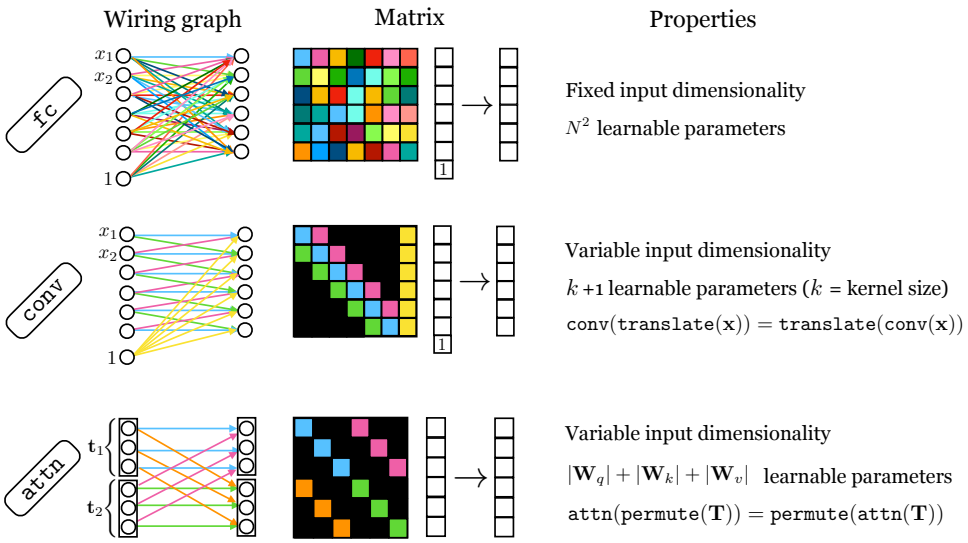

Figure 26.21: Comparing different kinds of layers that all represent an affine transformation between the inputs and outputs.

Another strategy is to simply let the positional codes be *learned* by the model, which could potentially result in a better representation of space than sinusoidal codes [109].

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 [303]. Positional encodings also appear in radiance fields, which we cover in chapter 45.

26.12 Comparing Fully Connected, Convolutional, and Self-Attention Layers

The shorthand "fc" is often used to indicate a fully-connected linear layer.

Many layers in deep nets are special kinds of affine transformations. Three we have seen so far are fully-connected layers (`fc`), convolutional layers (`conv`), and self-attention layers (`attn`). All these layers are alike in that their forward pass can be written as $\mathbf{X}_{\texttt{out}} = \mathbf{W}\mathbf{X}_{\texttt{in}} + \mathbf{b}$ 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 $\mathbf{X}_{\texttt{in}}$. In `conv` layers this function is very simple: just make a Toeplitz matrix that repeats the convolutional kernel(s) to match the dimensionality of $\mathbf{X}_{\texttt{in}}$. In `attn` 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 `attn` layers have low-rank structure, but different kinds of low-rank structure. In figure 26.21, we show what these matrices look like, and also catalog some of the other important properties of each of these layers.

26.13 Concluding Remarks

As of this writing, transformers are the dominant architecture in computer vision and in fact in most fields of artificial intelligence. They combine many of the best ideas from earlier architectures—convolutional patchwise processing, residual connections, relu nonlinearities, and normalization layers—with several newer innovations, in particular, vector-valued tokens, attention layers, and positional codes. Transformers can also be considered as a special case of **graph neural networks** (**GNNs**). We do not have a separate chapter on GNNs in this book since GNNs, other than transformers, are not yet popular in computer vision. GNNs are a very general class of architecture for processing *sets* by forming a graph of operations over the set. A transformer is doing exactly that: it takes an input set of tokens and, layer by layer, applies a network of transformations over that set until after enough layers a final representation or prediction is read out.