$$v_A = \sum_{i=1}^{m} v_{A,i} \tag{15.49}$$

We can similarly compare the hypothesis to the premise using

$$\alpha_j = \sum_{i=1}^{m} \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{kj})} a_i \tag{15.50}$$

$$v_{B,j} = g([b_j, \alpha_j]),\ j = 1, \ldots, n \tag{15.51}$$

$$v_B = \sum_{j=1}^{n} v_{B,j} \tag{15.52}$$

At the end, we classify the output using another MLP $h : \mathbb{R}^{2H}$ $\mathbb{R}^3$:

$$\hat{y} = h([v_A, v_B]) \tag{15.53}$$

See code.probml.ai/book1/entailment_attention_mlp_torch for some sample code.

We can modify this model to learn other kinds of mappings from sentence pairs to output labels. For example, in the semantic textual similarity task, the goal is to predict how semantically related two input sentences are. A standard dataset for this is the STS Benchmark [ Cer+17], where relatedness ranges from 0 (meaning unrelated) to 5 (meaning maximally related).

### 15.4.7 Soft vs hard attention

If we force the attention heatmap to be sparse, so that each output can only attend to one input location instead of a weighted combination of all of them, the method is called hard attention. We compare these two approaches for an image captioning problem in Figure 15.22. Unfortunately, hard attention results in a nondifferentiable training objective, and requires methods such as reinforcement learning to fit the model. See [Xu+15] for the details.

It seems from the above examples that these attention heatmaps can "explain" why the model generates a given output. However, the interpretability of attention is controversial (see e.g., [JW19; WP19; SS19; Bru+19] for discussion).

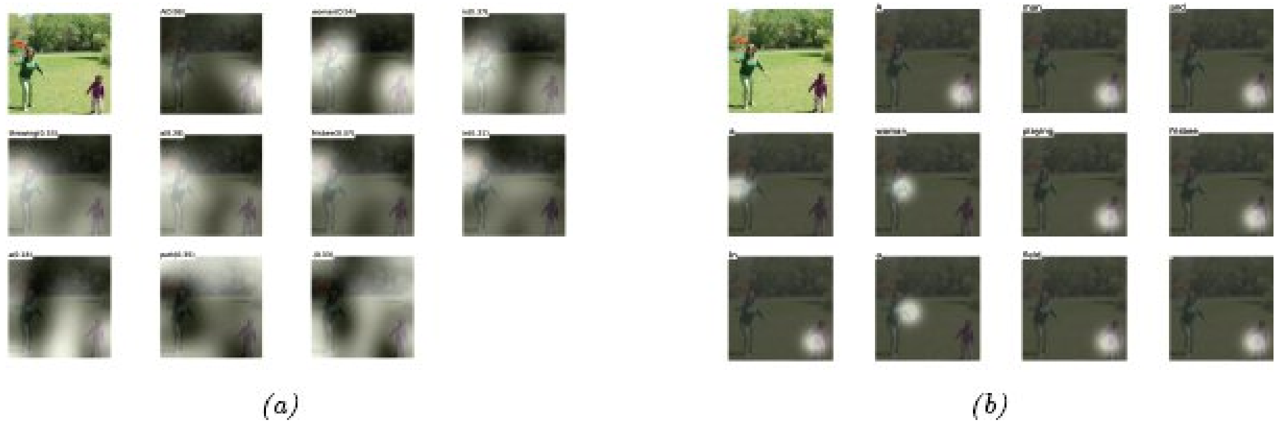

*Figure 15.22: Image captioning using attention. (a) Soft attention. Generates "a woman is throwing a frisbee in a park". (b) Hard attention. Generates "a man and a woman playing frisbee in a field". From Figure 6 of [Xu+15]. Used with kind permission of Kelvin Xu.*

## 15.5 Transformers

The transformer model [Vas+17] is a seq2seq model which uses attention in the encoder as well as the decoder, thus eliminating the need for RNNs, as we explain below. Transformers have been used for many (conditional) sequence generation tasks, such as machine translation [Vas+17], constituency parsing [Vas+17], music generation [Hua+18], protein sequence generation [Mad+20; Cho+20b], abstractive text summarization [Zha+19a], image generation [Par+18] (treating the image as a rasterized 1d sequence), etc.

The transformer is a rather complex model that uses several new kinds of building blocks or layers. We introduce these new blocks below, and then discuss how to put them all together.[4]

## 15.5.1 Self-attention

In Section 15.4.4 we showed how the decoder of an RNN could use attention to the input sequence in order to capture contexual embeddings of each input. However, rather than the decoder attending to the encoder, we can modify the model so the encoder attends to itself. This is called self attention [CDL16; Par+16b].

In more detail, given a sequence of input tokens $x_1, \ldots, x_n$, where $x_i$ $\mathbb{R}^d$, self-attention can generate a sequence of outputs of the same size using

$$y_i = \text{Attn}(x_i, (x_1, x_1), \ldots, (x_n, x_n)) \tag{15.54}$$

where the query is $x_i$, and the keys and values are all the (valid) inputs $x_1, \ldots, x_n$.

To use this in a decoder, we can set $x_i = y_{i1}$, and $n = i$ 1, so all the previously generated outputs are available. At training time, all the outputs are already known, so we can evaluate the above function in parallel, overcoming the sequential bottleneck of using RNNs.

In addition to improved speed, self-attention can give improved representations of context. As an example, consider translating the English sentences “The animal didn’t cross the street because it was too *tired*” and “The animal didn’t cross the street because it was too *wide*” into French. To generate a pronoun of the correct gender in French, we need to know what “it” refers to (this is called coreference resolution). In the first case, the word “it” refers to the animal. In the second case, the word “it” now refers to the street.

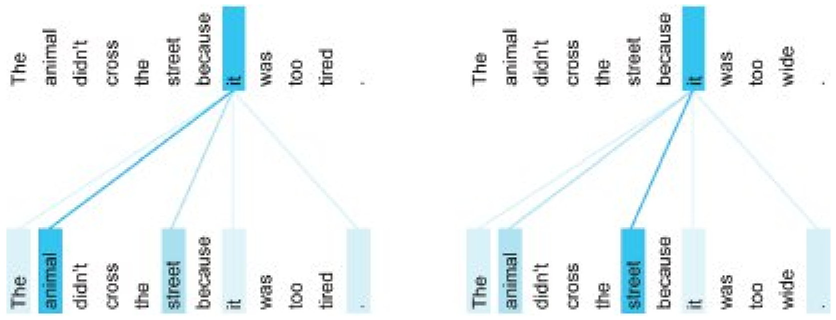
The animal didn't cross the street because it was too tired .
The animal didn't cross the street because it was too tired .
The animal didn't cross the street because it was too wide .
The animal didn't cross the street because it was too wide .

*Figure 15.23: Illustration of how encoder self-attention for the word “it” differs depending on the input context. From https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html. Used with kind permission of Jakob Uszkoreit.*

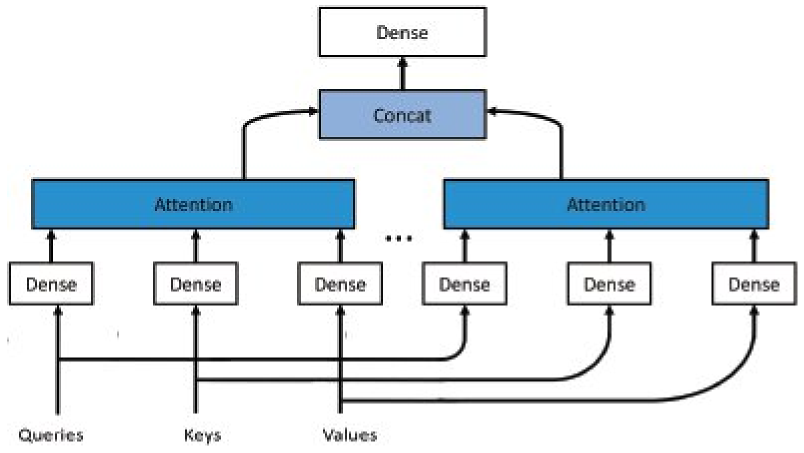

*Figure 15.24: Multi-head attention. Adapted from Figure 9.3.3 of [Zha+20].*

Figure 15.23 illustrates how self attention applied to the English sentence is able to resolve this ambiguity. In the first sentence, the representation for “it” depends on the earlier representations of “animal”, whereas in the latter, it depends on the earlier representations of “street”.

## 15.5.2 Multi-headed attention

If we think of an attention matrix as like a kernel matrix (as discussed in Section 15.4.2), it is natural to want to use multiple attention matrices, to capture different notions of similarity. This is the basic idea behind multi-headed attention (MHA). In more detail, query a given $\boldsymbol{q} \in \mathbb{R}^{d_q}$, keys $\boldsymbol{k}_j \in \mathbb{R}^{d_k}$, and values $\boldsymbol{v}_j \in \mathbb{R}^{d_v}$, we define the *i*'th attention head to be

$$\boldsymbol{h}_i = \mathrm{Attn}(\mathbf{W}_i^{(q)}\boldsymbol{q}, \{\mathbf{W}_i^{(k)}\boldsymbol{k}_j, \mathbf{W}_i^{(v)}\boldsymbol{v}_j\}) \in \mathbb{R}^{p_v} \tag{15.55}$$

where $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$, and $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ are projection matrices. We then stack the *h* heads together, and project to $\mathbb{R}^{p_o}$ using

$$\boldsymbol{h} = \mathrm{MHA}(\boldsymbol{q}, \{\boldsymbol{k}_j, \boldsymbol{v}_j\}) = \mathbf{W}_o \begin{pmatrix} \boldsymbol{h}_1 \\ \vdots \\ \boldsymbol{h}_h \end{pmatrix} \in \mathbb{R}^{p_o} \tag{15.56}$$

where $h_i$ is defined in Equation (15.55), and $\mathbf{W}_o \in \mathbb{R}^{p_o \times h p_v}$. If we set $p_q h = p_k h = p_v h = p_o$, we can compute all the output heads in parallel. See code.probml.ai/book1/multi_head_attention for some sample code.

## 15.5.3 Positional encoding

The performance of “vanilla” self-attention can be low, since attention is permutation invariant, and hence ignores the input word ordering. To overcome this, we can concatenate the word embeddings with a positional embedding, so that the model knows what order the words occur in.

One way to do this is to represent each position by an integer. However, neural networks cannot natively handle integers. To overcome this, we can encode the integer in binary form. For example, if we assume the sequence length is $n = 3$, we get the following sequence of $d = 3$-dimensional bit vectors for each location: 000, 001, 010, 011, 100, 101, 110, 111. We see that the right most index toggles the fastest (has highest frequency), whereas the left most index (most significant bit) toggles the slowest. (We could of course change this, so that the left most bit toggles fastest.) We can represent this as a position matrix P $\mathbb{R}^{n \times d}$.

We can think of the above representation as using a set of basis functions (corresponding to powers of 2), where the coefficients are 0 or 1. We can obtain a more compact code by using a different set of basis functions, and real-valued weights. [Vas+17] propose to use a sinusoidal basis, as follows:

$$p_{i,2j} = \sin\left(\frac{i}{C^{2j/d}}\right), \; p_{i,2j+1} = \cos\left(\frac{i}{C^{2j/d}}\right), \tag{15.57}$$

where $C = 10,000$ corresponds to some maximum sequence length. For example, if $d = 4$, the *i*'t row is

$$\boldsymbol{p}_i = [\sin(\frac{i}{C^{0/4}}), \cos(\frac{i}{C^{0/4}}), \sin(\frac{i}{C^{2/4}}), \cos(\frac{i}{C^{2/4}})] \tag{15.58}$$

Figure 15.25a shows the corresponding position matrix for $n = 60$ and $d = 32$. In this case, the left-most columns toggle fastest. We see that each row has a real-valued “fingerprint” representing its location in the sequence. Figure 15.25b shows some of the basis functions (column vectors) for dimensions 6 to 9.

The advantage of this representation is two-fold. First, it can be computed for arbitrary length inputs (up to $T$ $C$), unlike a learned mapping from integers to vectors. Second, the representation of one location is linearly predictable from any other, given knowledge of their relative distance. In particular, we have $p_{t+\phi} = f(p_t)$, where $f$ is a linear transformation. To see this, note that

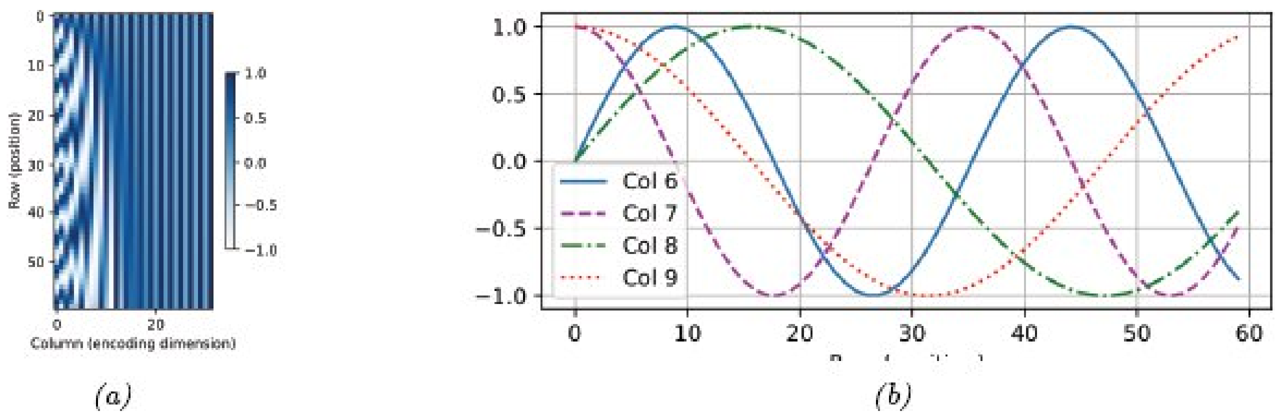

*Figure 15.25: (a) Positional encoding matrix for a sequence of length n* = 60 *and an embedding dimension of size d* = 32. *(b) Basis functions for columsn 6 to 9. Generated by code at* [*figures.probml.ai/book1/15.25*](figures.probml.ai/book1/15.25).

$$\begin{pmatrix}\sin(\omega_k(t+\phi)) \\ \cos(\omega_k(t+\phi))\end{pmatrix} = \begin{pmatrix}\sin(\omega_k t)\cos(\omega_k \phi) + \cos(\omega_k t)\sin(\omega_k \phi) \\ \cos(\omega_k t)\cos(\omega_k \phi) - \sin(\omega t)\sin(\omega_k \phi)\end{pmatrix} \tag{15.59}$$

$$= \begin{pmatrix}\cos(\omega_k \phi) & \sin(\omega_k \phi) \\ -\sin(\omega_k \phi) & \cos(\omega_k \phi)\end{pmatrix} \begin{pmatrix}\sin(\omega_k t) \\ \cos(\omega_k t)\end{pmatrix} \tag{15.60}$$

So if $\phi$ is small, then $p_{t+\phi}$ $p_t$. This provides a useful form of inductive bias.

Once we have computed the positional emebddings P, we need to combine them with the original word embeddings X using the following:[5]

$$\text{POS}(\text{Embed}(\mathbf{X})) = \mathbf{X} + \mathbf{P}. \tag{15.61}$$

## 15.5.4 Putting it all together

A transformer is a seq2seq model that uses self-attention for the encoder and decoder rather than an RNN. The encoder uses a series of encoder blocks, each of which uses multi-headed attention (Section 15.5.2), residual connections (Section 13.4.4), and layer normalization (Section 14.2.4.2). More precisely, the encoder block can be defined as follows:

```
def EncoderBlock(X):
  Z = LayerNorm(MultiHeadAttn(Q=X, K=X, V=X) + X)
  E = LayerNorm(FeedForward(Z) + Z)
  return E
```

The overall encoder is defined by applying positional encoding to the embedding of the input sequence, following by *N* copies of the encoder block, where *N* controls the depth of the block:

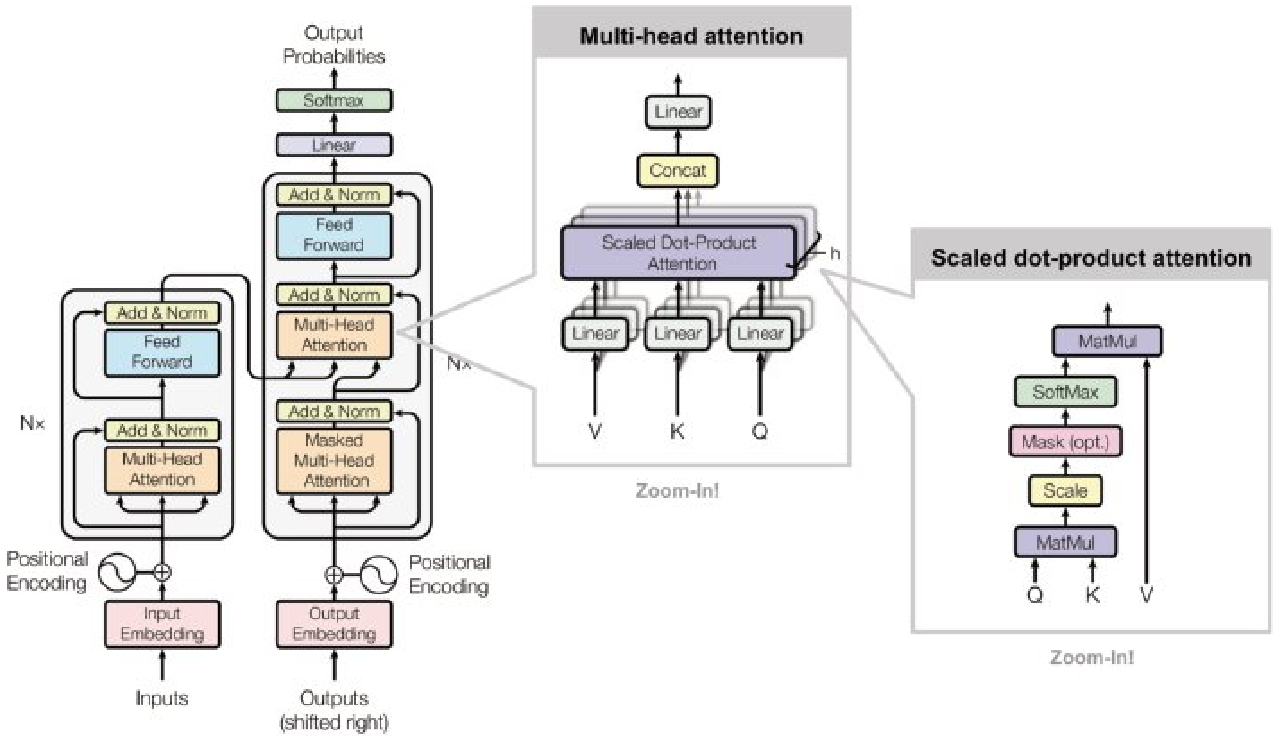

*Figure 15.26: The transformer. From [Wen18]. Used with kind permission of Lilian Weng. Adapted from Figures 1–2 of [Vas+17].*

```
def Encoder(X, N):
  E = POS(Embed(X))
  for n in range(N):
     E = EncoderBlock(E)
  return E
```

See the LHS of Figure 15.26 for an illustration.

The decoder has a somewhat more complex structure. It is given access to the encoder via another multi-head attention block. But it is also given access to previously generated outputs: these are shifted, and then combined with a positional embedding, and then fed into a masked (causal) multi-head attention model. Finally the output distribution over tokens at each location is computed in parallel.

In more detail, the decoder block is defined as follows:

```
def DecoderBlock(X, E):
  Z = LayerNorm(MultiHeadAttn(Q=X, K=X, V=X) + X)
  Z’ = LayerNorm(MultiHeadAttn(Q=Z, K=E, V=E) + Z)
  D = LayerNorm(FeedForward(Z’) + Z’)
  return D
```

The overall decoder is defined by $N$ copies of the decoder block:

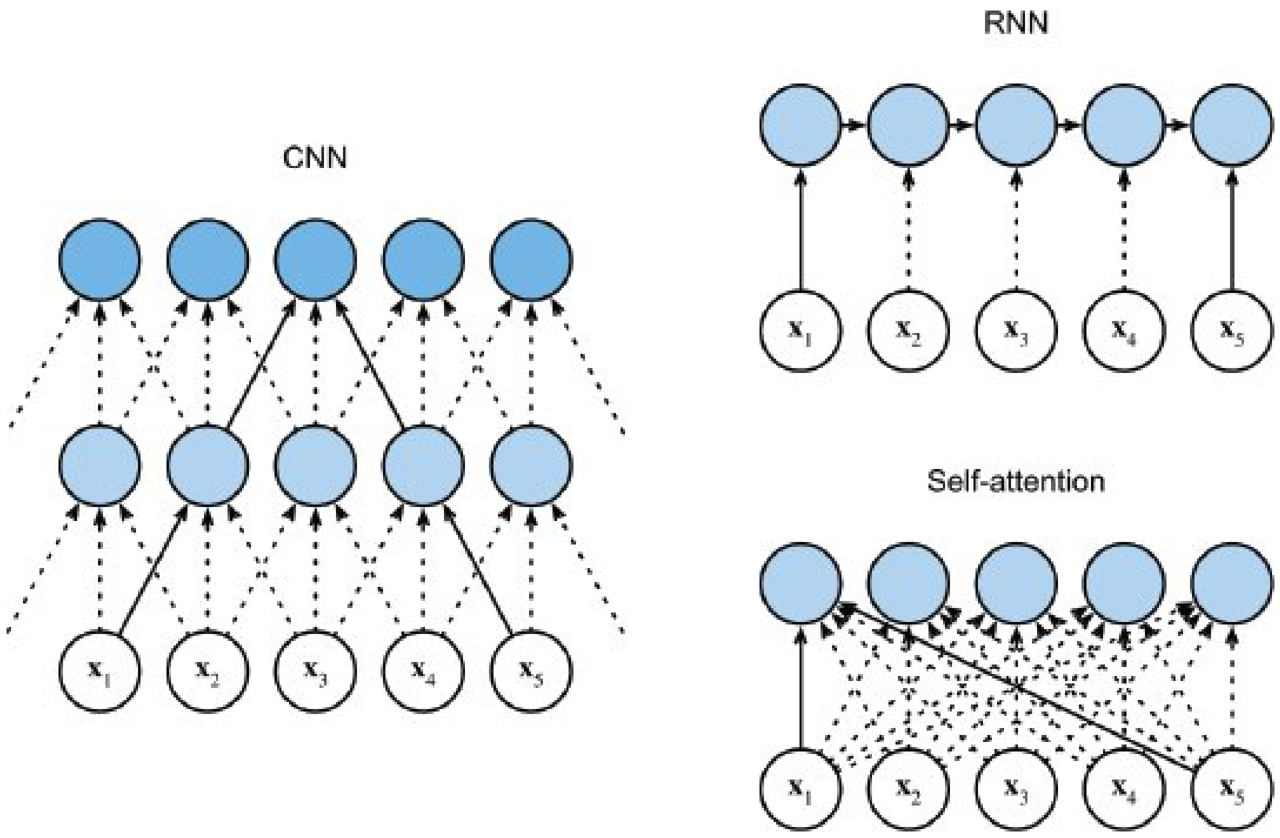

*Figure 15.27: Comparison of (1d) CNNs, RNNs and self-attention models. From Figure 10.6.1 of [Zha+20]. Used with kind permission of Aston Zhang.*

```
def Decoder(X, E, N):
  D = POS(Embed(X))
  for n in range(N):
     D = DecoderBlock(D,E)
  return D
```

See the RHS of Figure 15.26 for an illustration.

During training time, all the inputs X to the decoder are known in advance, since they are derived from embedding the lagged target output sequence. During inference (test) time, we need to decode sequentially, and use masked attention, where we feed the generated output into the embedding layer, and add it to the set of keys/values that can be attended to. (We initialize by feeding in the start-of-sequence token.) See code.probml.ai/book1/transformers_torch for some sample code, and [Rus18; Ala18] for a detailed tutorial on this model.

### 15.5.5 Comparing transformers, CNNs and RNNs

In Figure 15.27, we visually compare three different architectures for mapping a sequence $x_{1:n}$ to another sequence $y_{1:n}$: a 1d CNN, an RNN, and an attention-based model. Each model makes different tradeoffs in terms of speed and expressive power, where the latter can be quantified in terms of the maximum path length between any two inputs. See Table 15.1 for a summary.

For a 1d CNN with kernel size $k$ and $d$ feature channels, the time to compute the output is $O(knd^2)$, which can be done in parallel. We need a stack of $n/k$ layers, or $\log_k(n)$ if we use dilated convolution, to ensure all pairs can communicate. For example, in Figure 15.27, we see that $x_1$ and $x_5$ are initially 5 apart, and then 3 apart in layer 1, and then connected in layer 2.

| Layer type | Complexity | Sequential ops. | Max. path length |
|---|---|---|---|
| Self-attention | $O(n^2d)$ | $O(1)$ | $O(1)$ |
| Recurrent | $O(nd^2)$ | $O(n)$ | $O(n)$ |
| Convolutional | $O(knd^2)$ | $O(1)$ | $O(\log_k n)$ |

*Table 15.1: Comparison of the transformer with other neural sequential generative models. n is the sequence length, d is the dimensionality of the input features, and k is the kernel size for convolution. Based on Table 1 of [Vas+17].*

For an RNN, the computational complexity is $O(nd^2)$, for a hidden state of size $d$, since we have to perform matrix-vector multiplication at each step. This is an inherently sequential operation. The maximum path length is $O(n)$.

Finally, for self-attention models, every output is directly connected to every input, so the maximum path length is $O(1)$. However, the computational cost is $O(n^2d)$. For short sequences, we typically have $n \ll d$, so this is fine. For longer sequences, we discuss various fast versions of attention in Section 15.6.

### 15.5.6 Transformers for images *

CNNs (Chapter 14) are the most common model type for processing image data, since they have useful built-in inductive bias, such as locality (due to small kernels), equivariance (due to weight tying), and invariance (due to pooling). Suprisingly, it has been found that transformers can also do well at image classification, at least if trained on enough data. (They need a lot of data to overcome their lack of relevant inductive bias.)

In particular, [Dos+21] present the ViT model (vision transformer), that chops the input up into 16x16 patches, projects each patch into an embedding space, and then passes this set of embeddings $x_{1:T}$ to a transformer, analogous to the way word embeddings are passed to a transformer. The input is also prepended with a special [CLASS] embedding, $x_0$. The output of the transformer is a set of encodings $e_{0:T}$; the model maps $e_0$ to the target class label $y$, and is trained in a supervised way. See Figure 15.28 for an illustration.

After supervised pretraining, the model is fine-tuned on various downstream classification tasks, an approach known as transfer learning (see Section 19.2 for more details). When trained on “small” datasets such as ImageNet (which has 1k classes and 1.3M images), they find that they cannot outperform a pretrained CNN ResNet model (Section 14.3.4) known as BiT (big transfer) [Kol+20]. However, when trained on larger datasets, such as ImageNet-21k (with 21k classes and 14M images), or the Google-internal JFT dataset (with 18k classes and 303M images), they find that ViT does better than BiT at transfer learning. It is also cheaper to train than ResNet at this scale. (However, training is still expensive: the large ViT model on ImageNet-21k takes 30 days on a Google Cloud TPUv3 with 8 cores!)

### 15.5.7 Other transformer variants *

Many extensions of transformers have been published in the last few years. For example, the Gshard paper [Lep+21] shows how to scale up transformers to even more parameters by replacing some of the feed forward dense layers with a mixture of experts (Section 13.6.2) regression module. This allows for sparse conditional computation, in which only a subset of the model capacity (chosen by the gating network) is used for any given input.

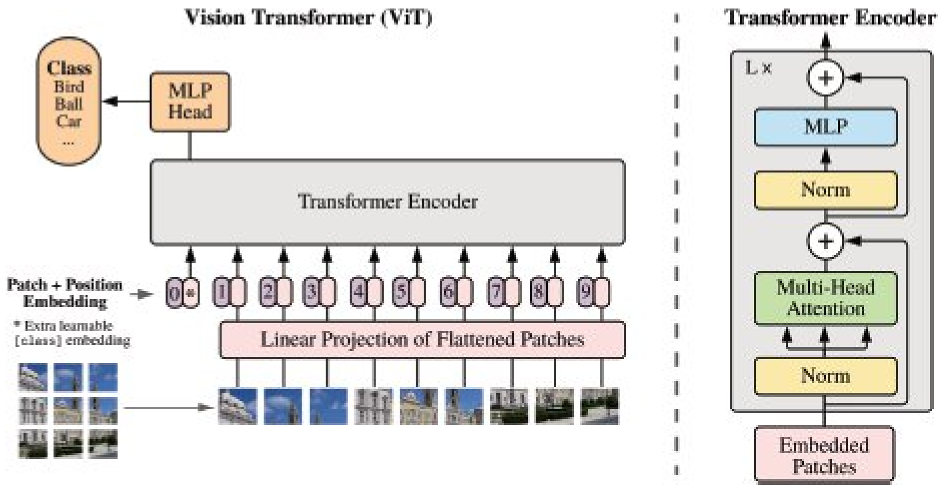

*Figure 15.28: The Vision Transformer (ViT) model. This treats an image as a set of input patches. The input is prepended with the special CLASS embedding vector (denoted by *) in location 0. The class label for the image is derived by applying softmax to the final ouput encoding at location 0. From Figure 1 of [Dos+21]. Used with kind permission of Alexey Dosovitskiy*

As another example, the conformer paper [Gul+20] showed how to add convolutional layers inside the transformer architecture, which was shown to be helpful for various speech recognition tasks.

## 15.6 Efficient transformers *

*This section is written by Krzysztof Choromanski.*

Regular transformers take $O(N^2)$ time and space complexity, for a sequence of length $N$, which makes them impractical to apply to long sequences. In the past few years, researchers have proposed several more efficient variants of transformers to bypass this difficulty. In this section, we give a brief survey of some of these methods (see Figure 15.29 for a summary). For more details, see e.g., [Tay+20a; Tay+20b; Lin+21].

### 15.6.1 Fixed non-learnable localized attention patterns

The simplest modification of the attention mechanism is to constrain it to a fixed non-learnable localized window, in other words restrict each token to attend only to a pre-selected set of other tokens. If for instance, each sequence is chunked into $K$ blocks, each of length $\frac{N}{K}$, and attention is conducted only within a block, then space/time complexity is reduced from $O(N^2)$ to $\frac{N^2}{K}$. For $K \gg 1$ this constitutes substantial overall computational improvements. Such an approach is applied in particular in [Qiu+19a; Par+18]. The attention patterns do not need to be in the form of blocks. Other approaches involve strided / dilated windows, or hybrid patterns, where several fixed attention patterns are combined together [Chi+19b; BPC20].