Understanding Deep Learning

Chapter 9: Transformers

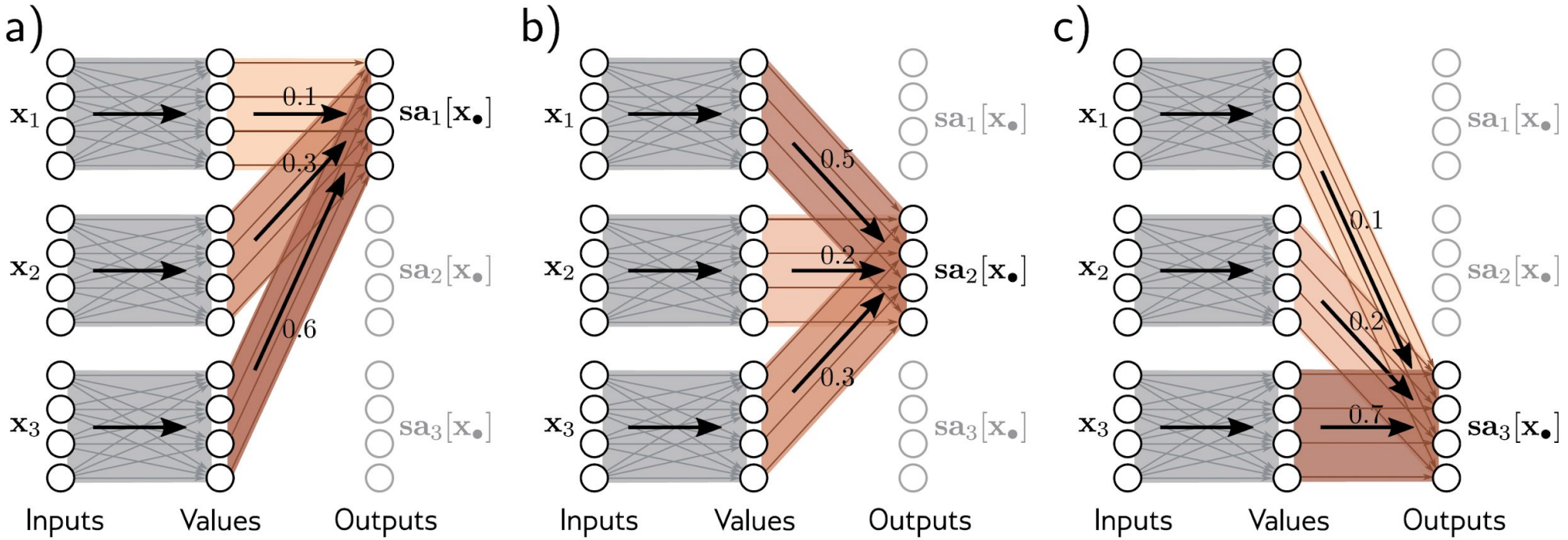

Figure 12.1 Self-attention as routing. The self-attention mechanism takes N inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^D$ (here $N = 3$ and $D = 4$) and processes each separately to compute N value vectors. The n^{th} output $\mathbf{sa}_n[\mathbf{x}_1, \ldots \mathbf{x}_N]$ (written as $\mathbf{sa}_n[\mathbf{x}_\bullet]$ for short) is then computed as a weighted sum of the N value vectors, where the weights are positive and sum to one. a) Output $\mathbf{sa}_1[\mathbf{x}_\bullet]$ is computed as $a[\mathbf{x}_1, \mathbf{x}_1] = 0.1$ times the first value vector, $a[\mathbf{x}_2, \mathbf{x}_1] = 0.3$ times the second value vector, and $a[\mathbf{x}_3, \mathbf{x}_1] = 0.6$ times the third value vector. b) Output $\mathbf{sa}_2[\mathbf{x}_\bullet]$ is computed in the same way, but this time with weights of 0.5, 0.2, and 0.3. c) The weighting for output $\mathbf{sa}_3[\mathbf{x}_\bullet]$ is different again. Each output can hence be thought of as a different routing of the N values.

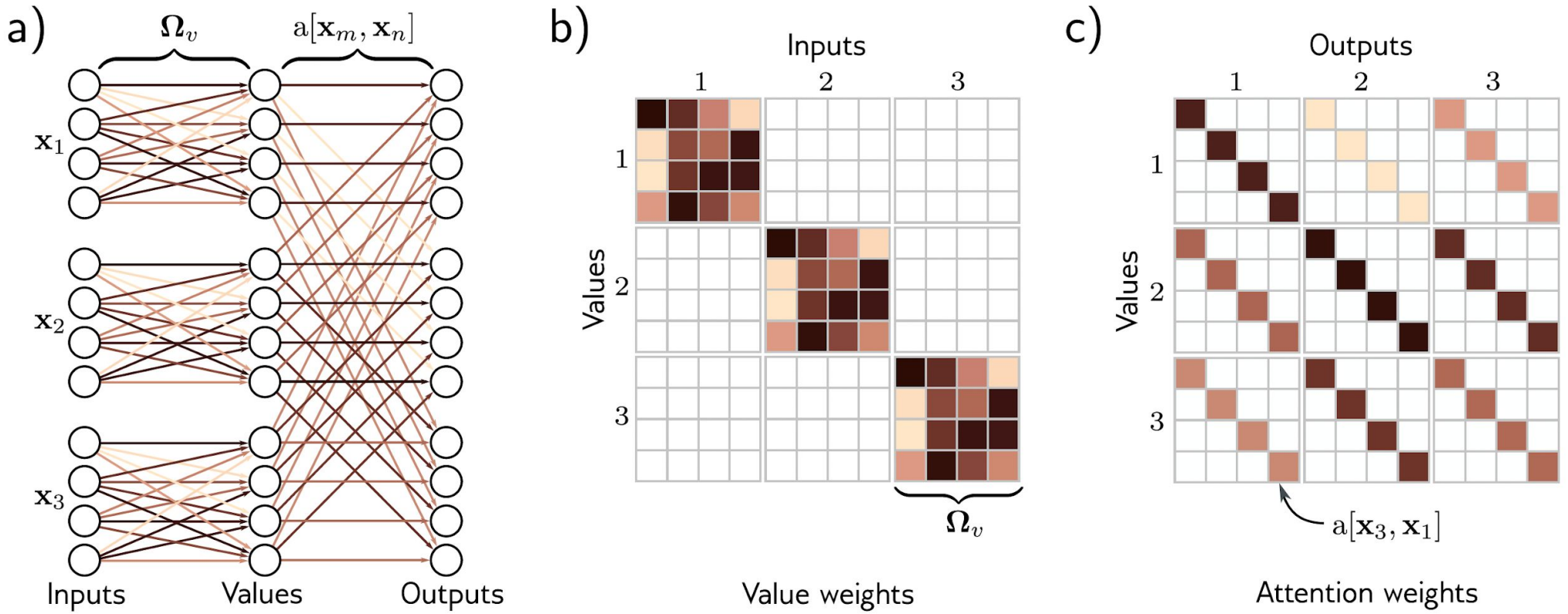

Figure 12.2 Self-attention for $N = 3$ inputs $\mathbf{x}_n$, each with dimension $D = 4$. a) Each input $\mathbf{x}_n$ is operated on independently by the same weights $\boldsymbol{\Omega}_v$ (same color equals same weight) and biases $\boldsymbol{\beta}_v$ (not shown) to form the values $\boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_n$. Each output is a linear combination of the values, with a shared attention weight $a[\mathbf{x}_m, \mathbf{x}_n]$ defining the contribution of the m^{th} value to the n^{th} output. b) Matrix showing block sparsity of linear transformation $\boldsymbol{\Omega}_v$ between inputs and values. c) Matrix showing sparsity of attention weights relating values and outputs.

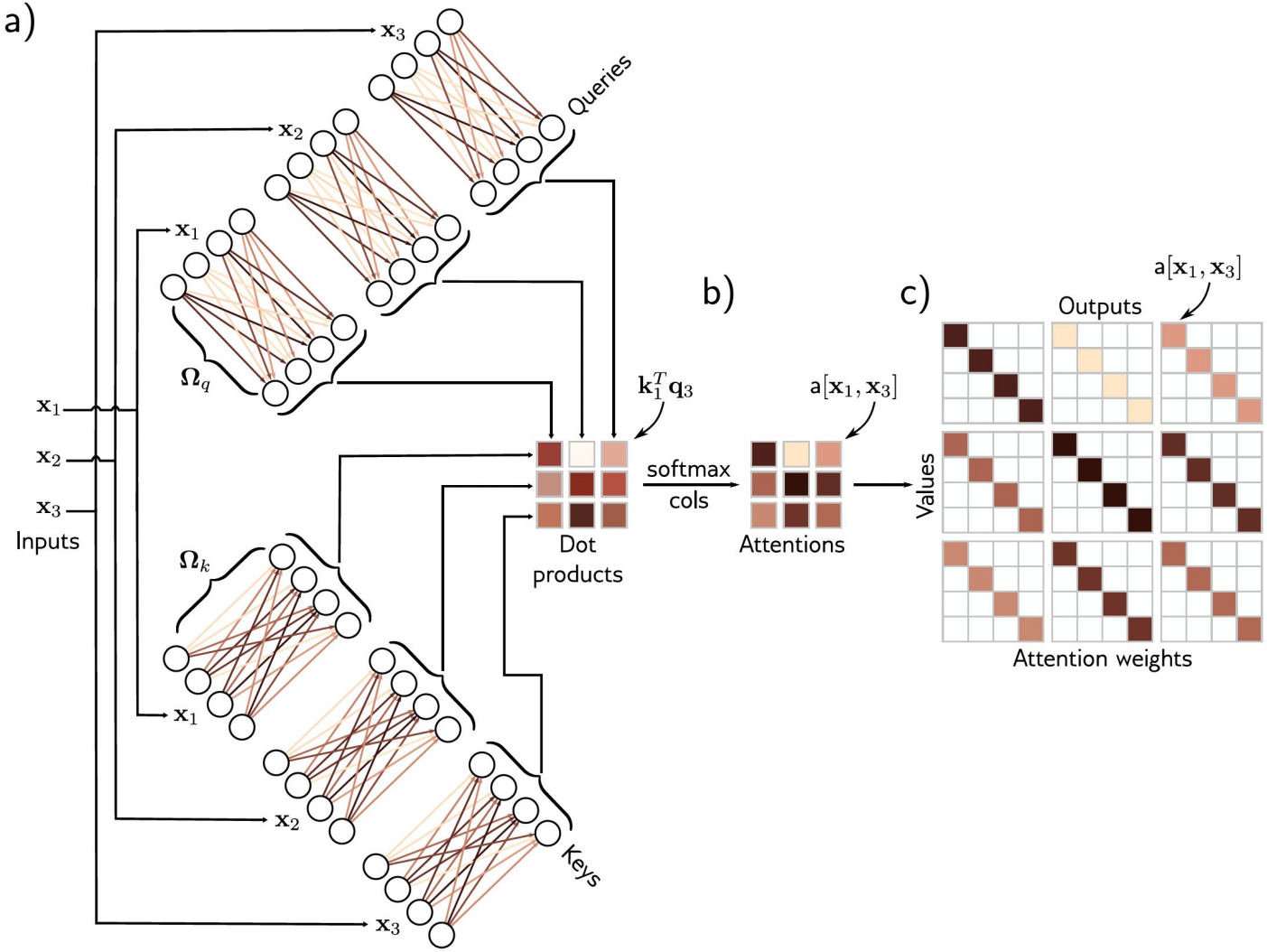

Figure 12.3 Computing attention weights. a) Query vectors $\mathbf{q}_n = \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q\mathbf{x}_n$ and key vectors $\mathbf{k}_n = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{x}_n$ are computed for each input $\mathbf{x}_n$. b) The dot products between each query and the three keys are passed through a softmax function to form non-negative attentions that sum to one. c) These route the value vectors (figure 12.1) via the sparse matrix from figure 12.2c.

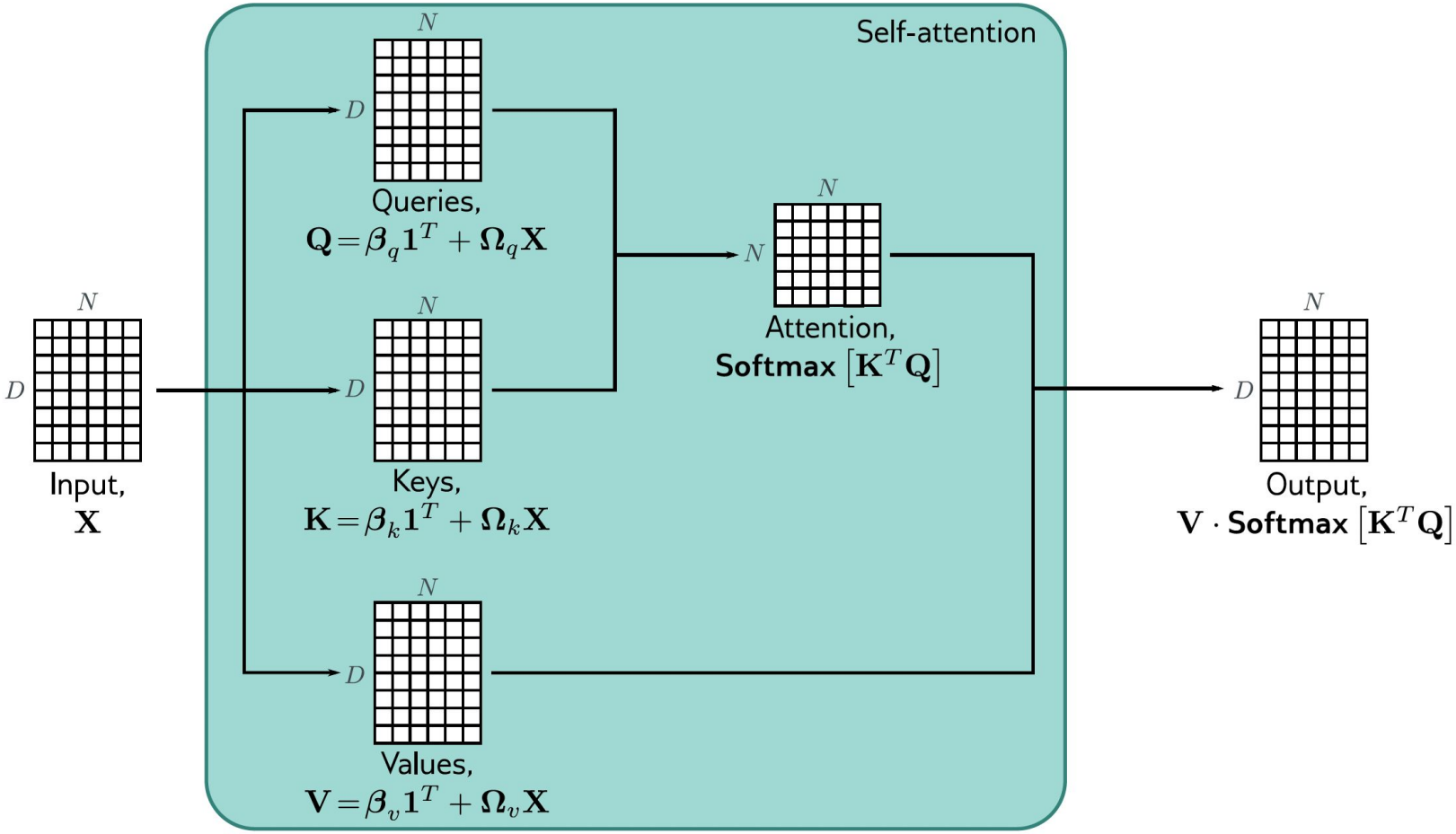

Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication, and a softmax operation is applied independently to each column of the resulting matrix to calculate the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

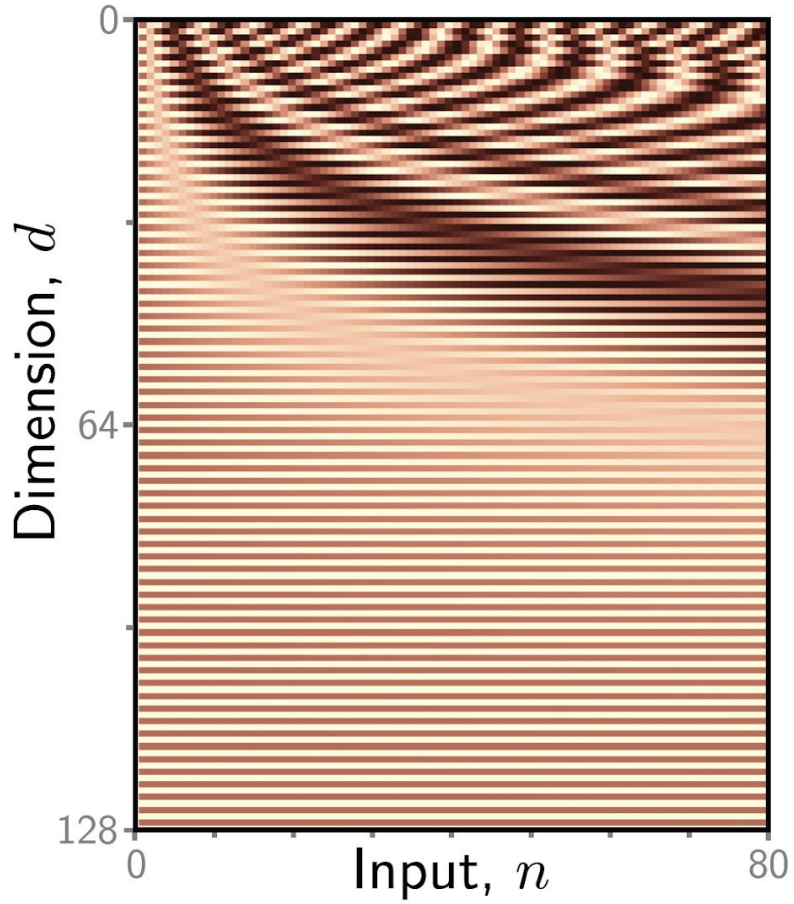

Figure 12.5 Positional encodings. The self-attention architecture is equivariant to permutations of the inputs. To ensure that inputs at different positions are treated differently, a positional encoding matrix $\mathbf{\Pi}$ can be added to the data matrix. Each column is different, so the positions can be distinguished. Here, the position encodings use a predefined procedural sinusoidal pattern (which can be extended to larger values of N if necessary). However, in other cases, they are learned.

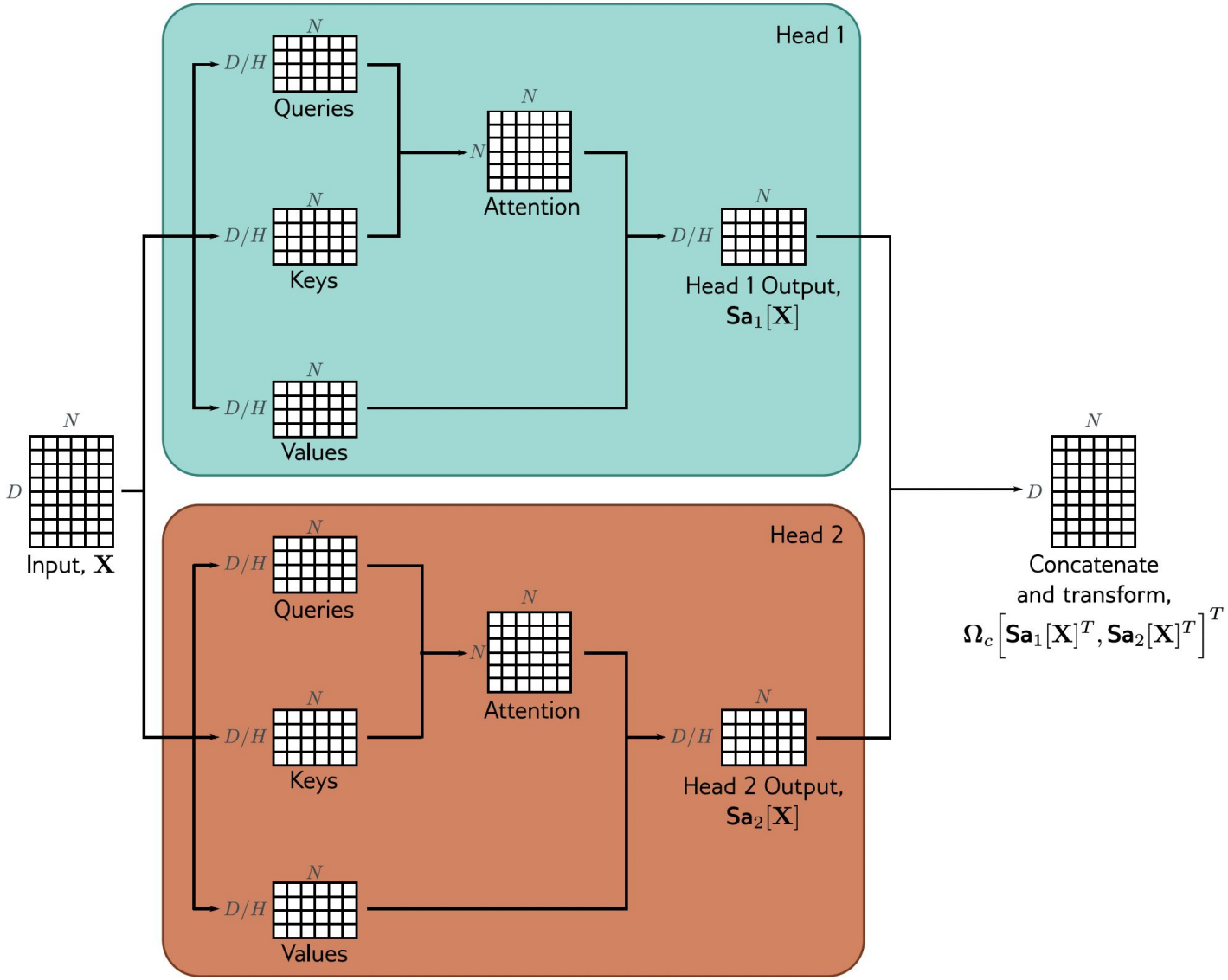

Figure 12.6 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads." Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated, and another linear transformation $\boldsymbol{\Omega}_c$ is used to recombine them.

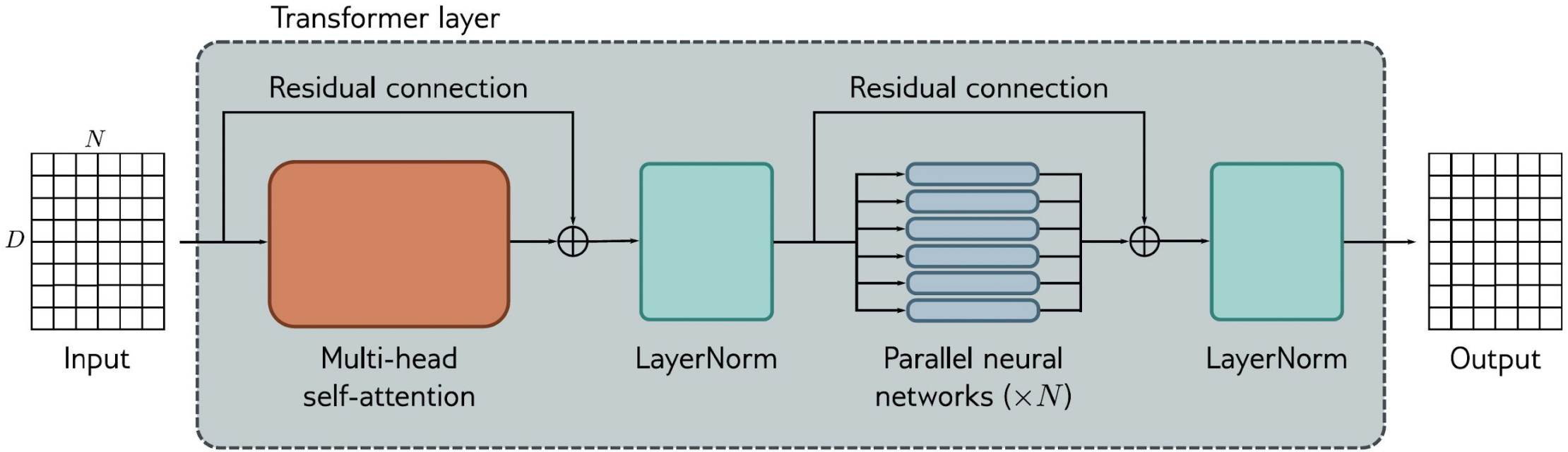

Figure 12.7 The transformer. The input consists of a $D \times N$ matrix containing the D-dimensional word embeddings for each of the N input tokens. The output is a matrix of the same size. The transformer consists of a series of operations. First, there is a multi-head attention block, allowing the word embeddings to interact with one another. This forms the processing of a residual block, so the inputs are added back to the output. Second, a LayerNorm operation is applied. Third, there is a second residual layer where the same fully connected neural network is applied separately to each of the N word representations (columns). Finally, LayerNorm is applied again.

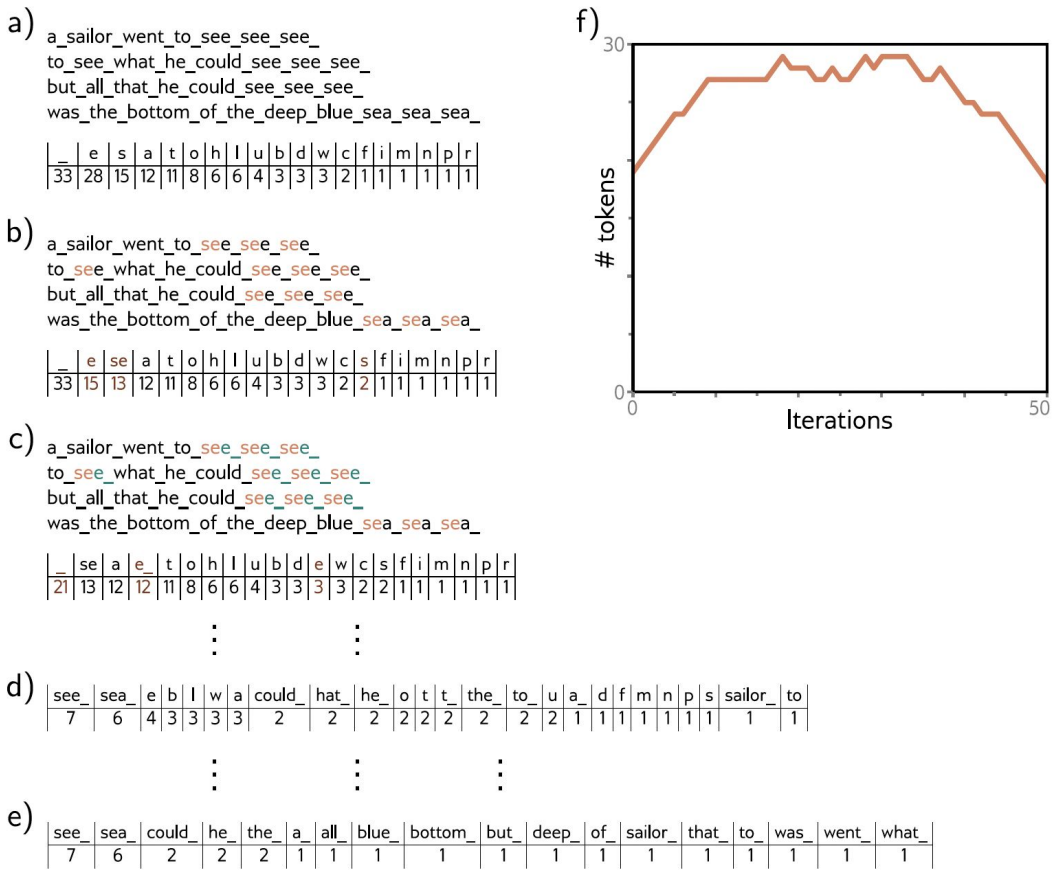

Figure 12.8 Sub-word tokenization. a) A passage of text from a nursery rhyme. The tokens are initially just the characters and whitespace (represented by an underscore), and their frequencies are displayed in the table. b) At each iteration, the sub-word tokenizer looks for the most commonly occurring adjacent pair of characters (in this case, se) and merges them. This creates a new token and decreases the counts for the original tokens s and e. c) At the second iteration, the algorithm merges e and the whitespace character_. Note that the last character of the first token to be merged cannot be whitespace, which prevents merging across words. d) After 22 iterations, the tokens consist of a mix of letters, word fragments, and commonly occurring words. e) If we continue this process indefinitely, the tokens eventually represent the full words. f) Over time, the number of tokens increases as we add word fragments to the letters and then decreases again as we merge these fragments. In a real situation, there would be a very large number of words, and the algorithm would terminate when the vocabulary size (number of tokens) reached a predetermined value. Punctuation and capital letters would also be treated as separate input characters.

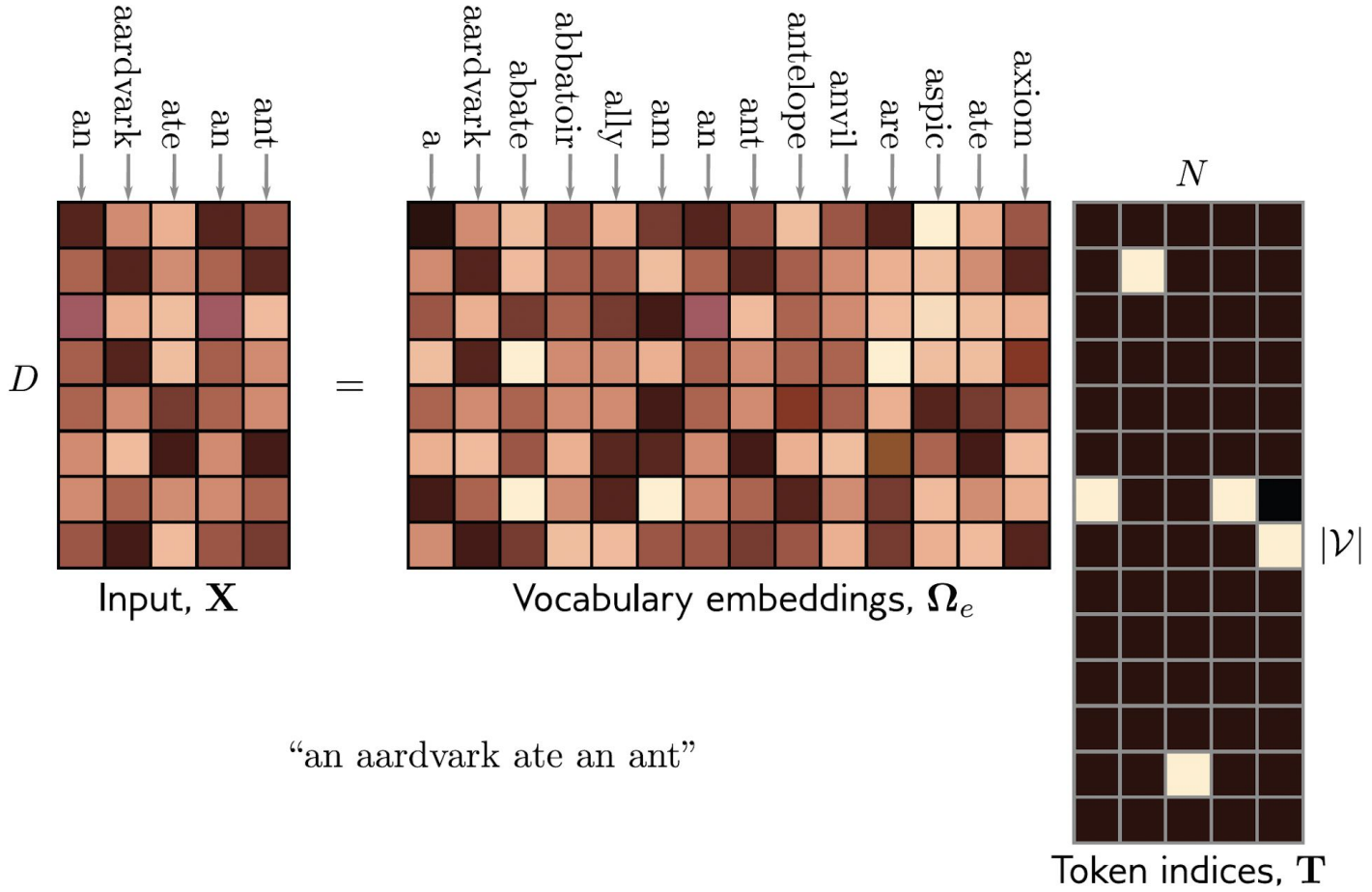

Figure 12.9 The input embedding matrix $\mathbf{X} \in \mathbb{R}^{D \times N}$ contains N embeddings of length D and is created by multiplying a matrix $\boldsymbol{\Omega}_e$ containing the embeddings for the entire vocabulary with a matrix containing one-hot vectors in its columns that correspond to the word or sub-word indices. The vocabulary matrix $\boldsymbol{\Omega}_e$ is considered a parameter of the model and is learned along with the other parameters. Note that the two embeddings for the word an in $\mathbf{X}$ are the same.

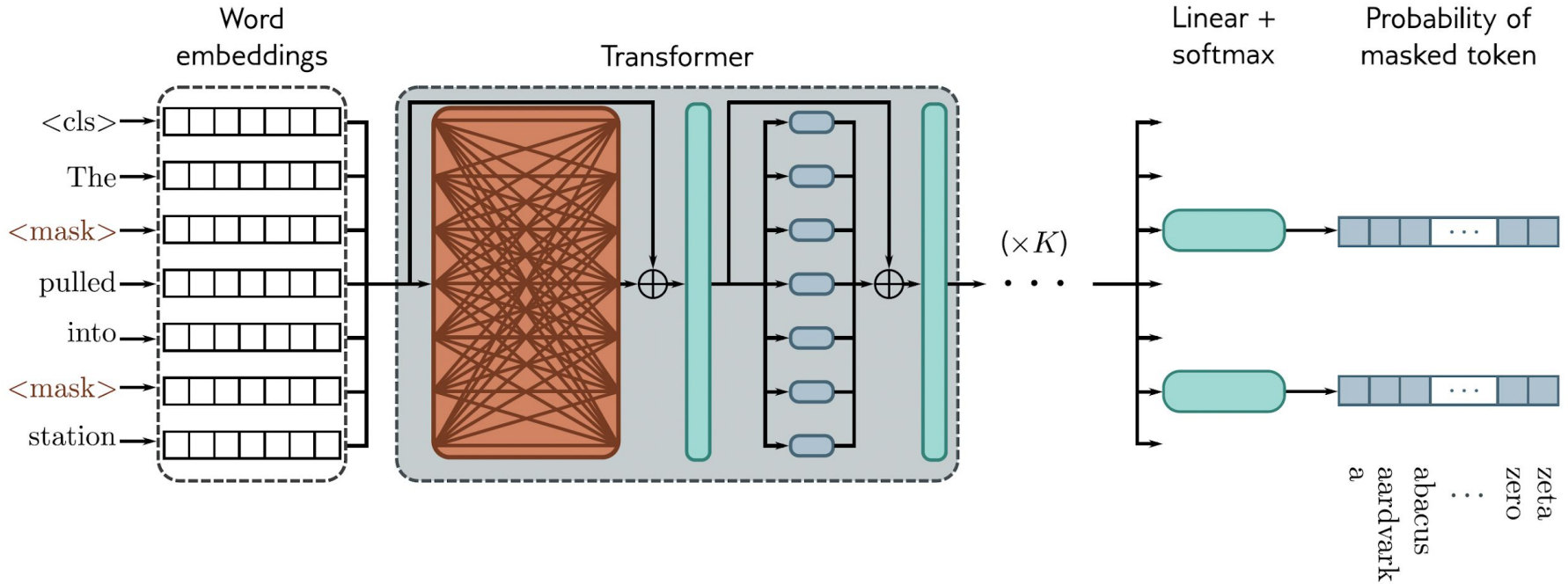

Figure 12.10 Pre-training for BERT-like encoder. The input tokens (and a special <cls> token denoting the start of the sequence) are converted to word embeddings. Here, these are represented as rows rather than columns, so the box labeled "word embeddings" is $\mathbf{X}^T$. These embeddings are passed through a series of transformers (orange connections indicate that every token attends to every other token in these layers) to create a set of output embeddings. A small fraction of the input tokens is randomly replaced with a generic <mask> token. In pre-training, the goal is to predict the missing word from the associated output embedding. As such, the output embeddings are passed through a softmax function, and the multiclass classification loss (section 5.24) is used. This task has the advantage that it uses both the left and right context to predict the missing word but has the disadvantage that it does not make efficient use of data; here, seven tokens need to be processed to add two terms to the loss function.

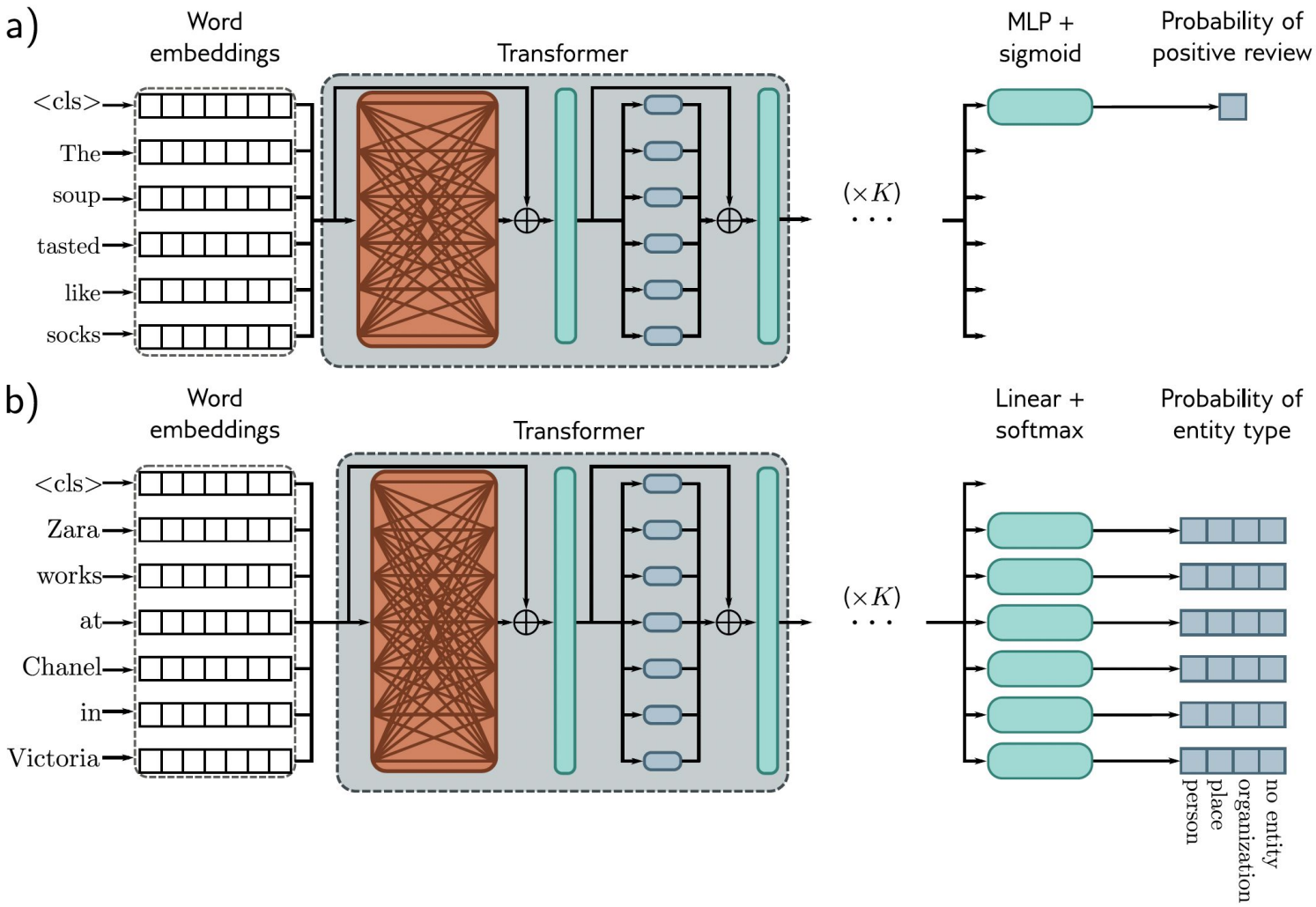

Figure 12.11 After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required. a) Example text classification task. In this sentiment classification task, the <cls> token embedding is used to predict the probability that the review is positive. b) Example word classification task. In this named entity recognition problem, the embedding for each word is used to predict whether the word corresponds to a person, place, or organization, or is not an entity.

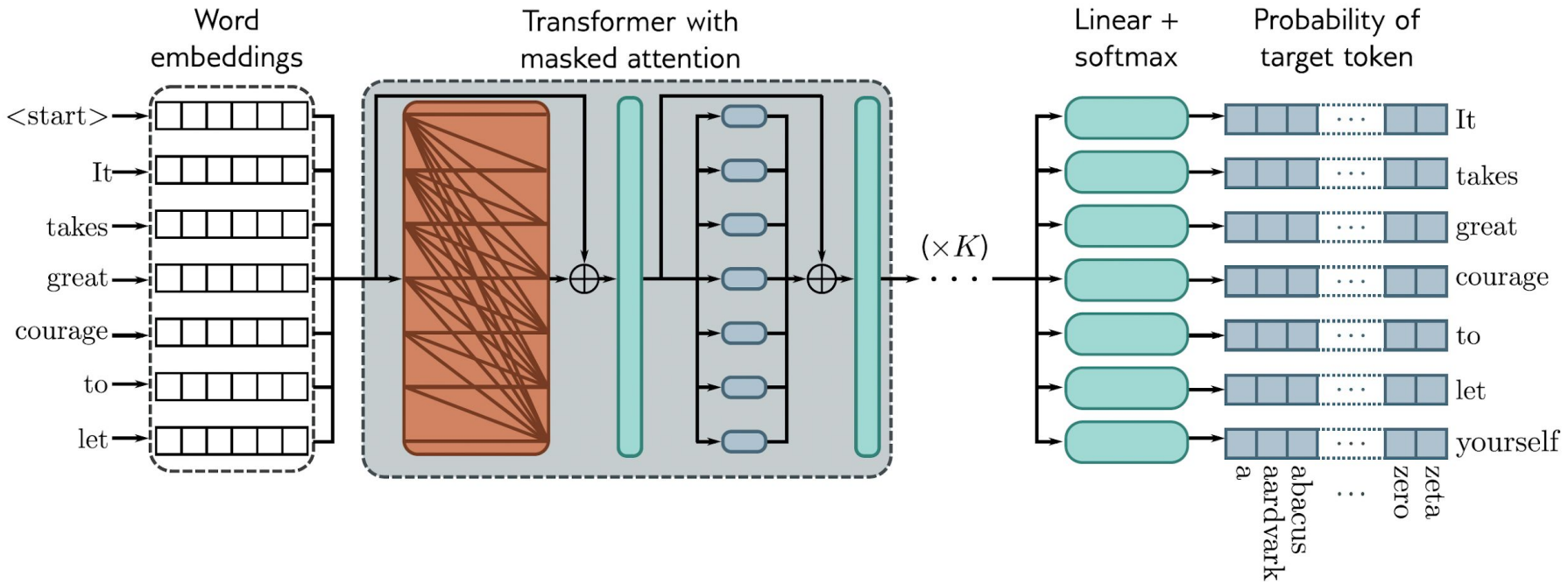

Figure 12.12 Training GPT3-type decoder network. The tokens are mapped to word embeddings with a special <start> token at the beginning of the sequence. The embeddings are passed through a series of transformers that use masked self-attention. Here, each position in the sentence can only attend to its own embedding and the embeddings of tokens earlier in the sequence (orange connections). The goal at each position is to maximize the probability of the following ground truth token in the sequence. In other words, at position one, we want to maximize the probability of the token It; at position two, we want to maximize the probability of the token takes; and so on. Masked self-attention ensures the system cannot cheat by looking at subsequent inputs. The autoregressive task has the advantage of making efficient use of the data since every word contributes a term to the loss function. However, it only exploits the left context of each word.

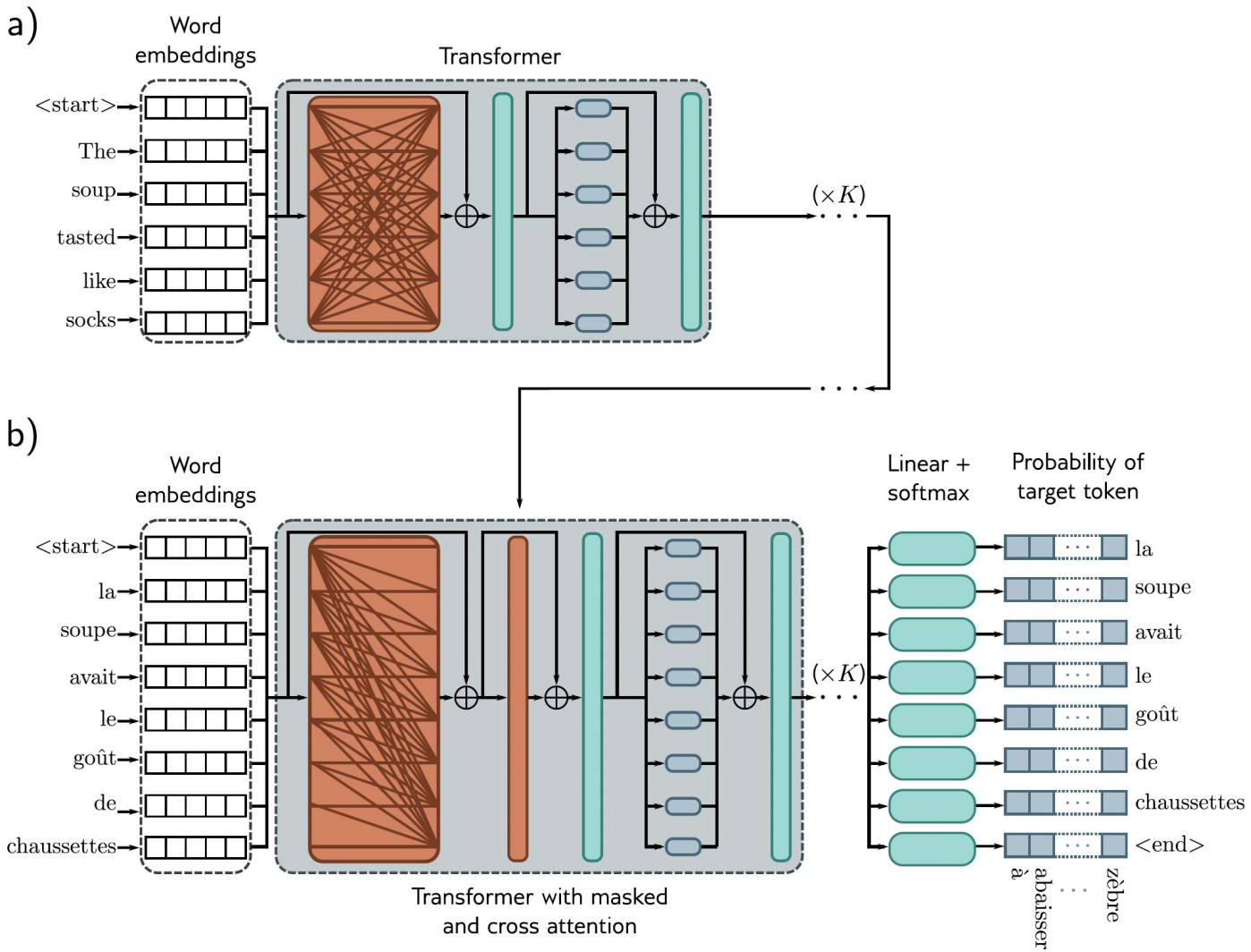

Figure 12.13 Encoder-decoder architecture. Two sentences are passed to the system with the goal of translating the first into the second. a) The first sentence is passed through a standard encoder. b) The second sentence is passed through a decoder that uses masked self-attention but also attends to the output embeddings of the encoder using cross-attention (orange rectangle). The loss function is the same as for the decoder model; we want to maximize the probability of the next word in the output sequence.

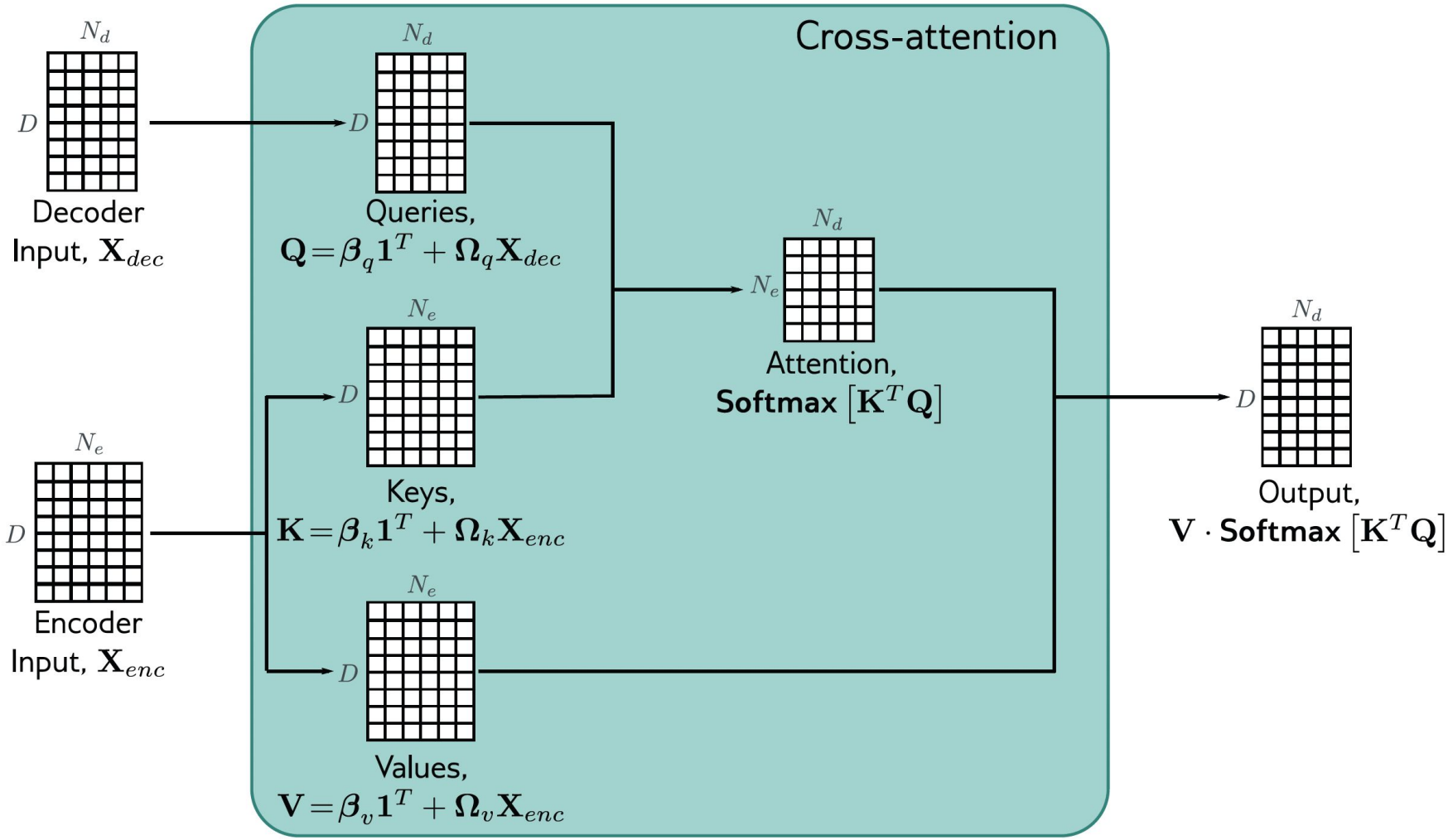

Figure 12.14 Cross-attention. The flow of computation is the same as in standard self-attention. However, the queries are now calculated from the decoder embeddings $\mathbf{X}_{dec}$, and the keys and values from the encoder embeddings $\mathbf{X}_{enc}$.

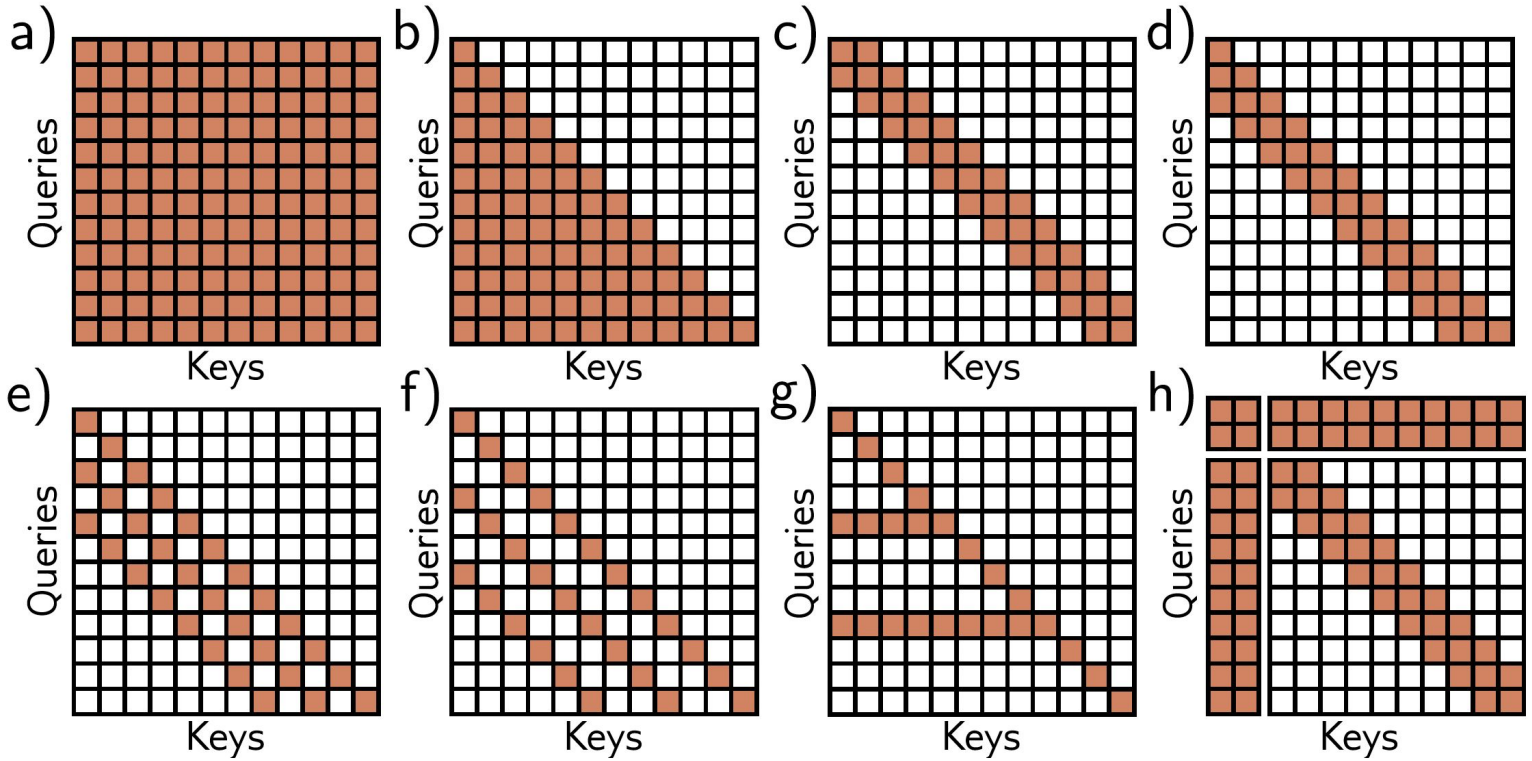

Figure 12.15 Interaction matrices for self-attention. a) In an encoder, every token interacts with every other token, and computation expands quadratically with the number of tokens. b) In a decoder, each token only interacts with the previous tokens, but complexity is still quadratic. c) Complexity can be reduced by using a convolutional structure (encoder case). d) Convolutional structure for decoder case. e–f) Convolutional structure with dilation rate of two and three (decoder case). g) Another strategy is to allow selected tokens to interact with all the other tokens (encoder case) or all the previous tokens (decoder case pictured). h) Alternatively, global tokens can be introduced (left two columns and top two rows). These interact with all of the tokens as well as with each other.

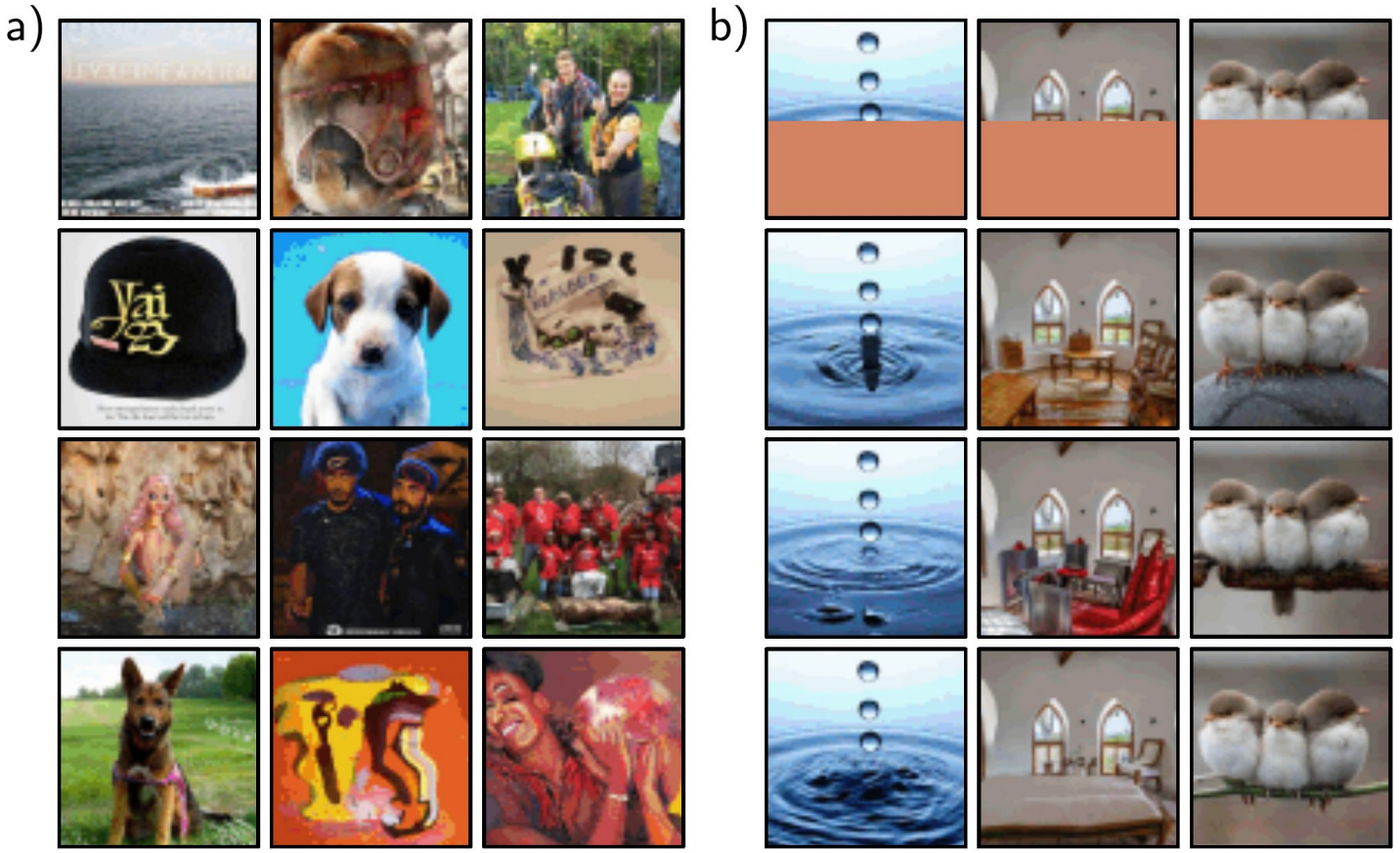

Figure 12.16 ImageGPT. a) Images generated from the autoregressive ImageGPT model. The top-left pixel is drawn from the estimated empirical distribution at this position. Subsequent pixels are generated in turn, conditioned on the previous ones, working along the rows until the bottom-right of the image is reached. For each pixel, the transformer decoder generates a conditional distribution as in equation 12.15, and a sample is drawn. The extended sequence is then fed back into the network to generate the next pixel, and so on. b) Image completion. In each case, the lower half of the image is removed (top row), and ImageGPT completes the remaining part pixel by pixel (three different completions shown). Adapted from https://openai.com/blog/image-gpt/.

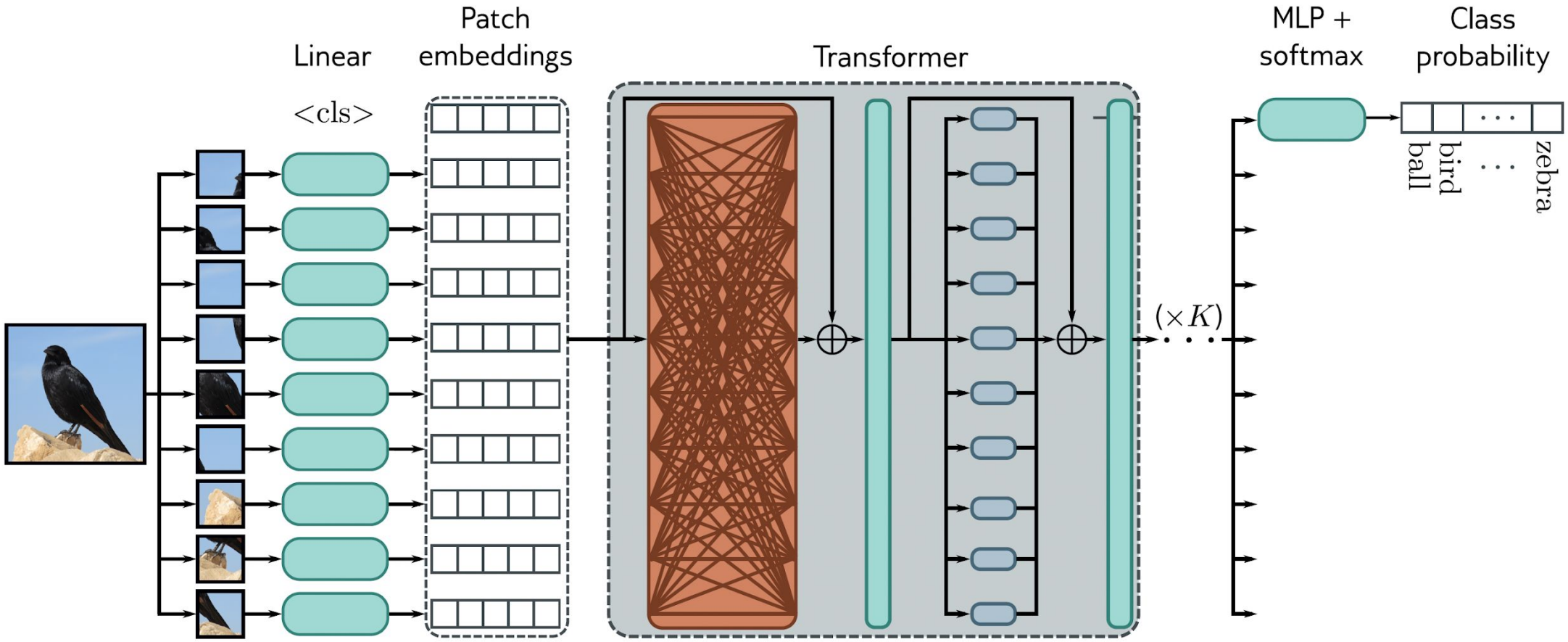

Figure 12.17 Vision transformer. The Vision Transformer (ViT) breaks the image into a grid of patches (16×16 in the original implementation). Each of these is projected via a learned linear transformation to become a patch embedding. These patch embeddings are fed into a transformer encoder network, and the <cls> token is used to predict the class probabilities.

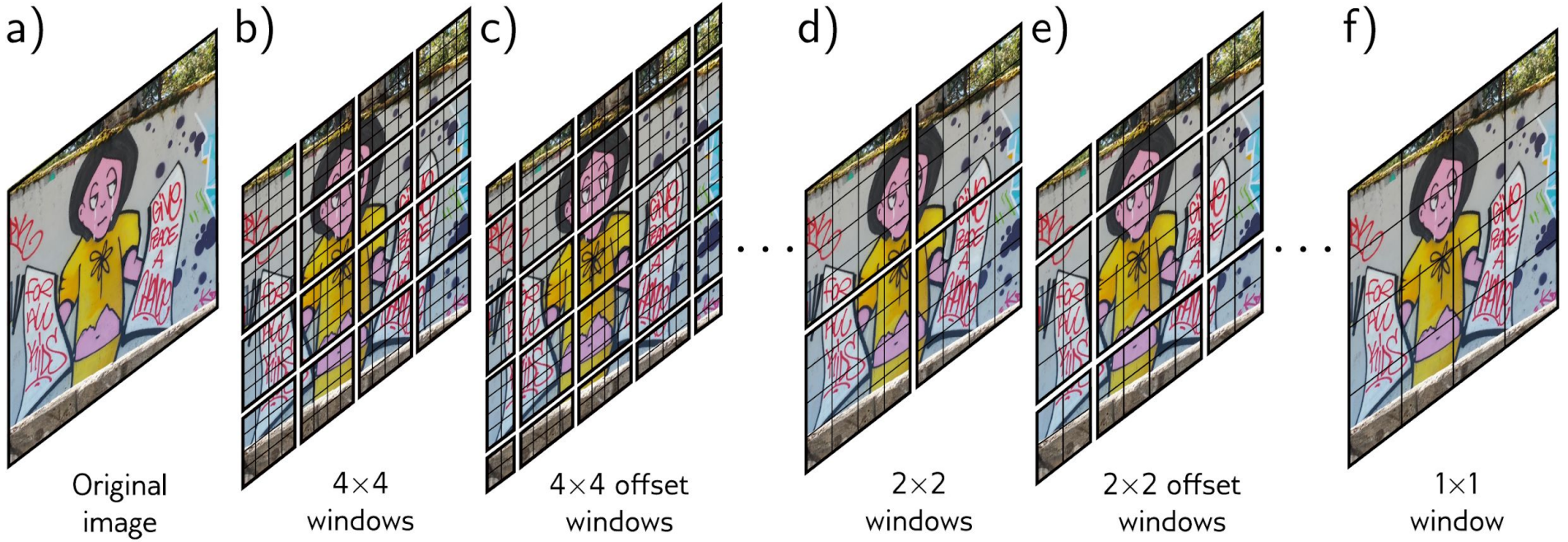

Figure 12.18 Shifted window (SWin) transformer (Liu et al., 2021c). a) Original image. b) The SWin transformer breaks the image into a grid of windows and each of these windows into a sub-grid of patches. The transformer network applies self-attention to the patches within each window independently. c) Each alternate layer shifts the windows so that the subsets of patches that interact with one another change, and information can propagate across the whole image. d) After several layers, the 2×2 blocks of patch representations are concatenated to increase the effective patch (and window) size. e) Alternate layers use shifted windows at this new lower resolution. f) Eventually, the resolution is such that there is just a single window, and the patches span the entire image.

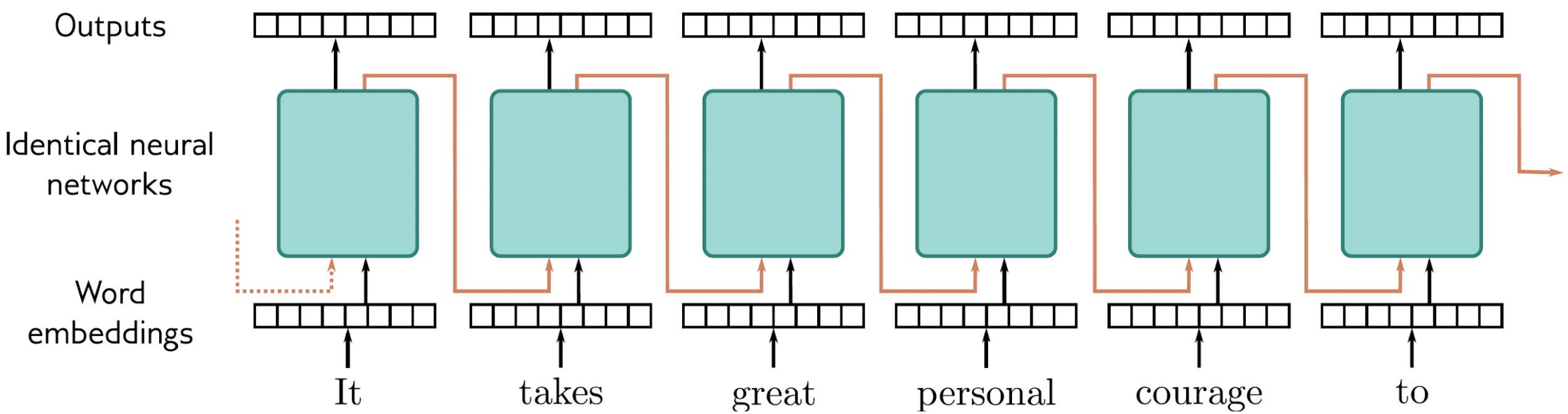

Figure 12.19 Recurrent neural networks (RNNs). The word embeddings are passed sequentially through a series of identical neural networks. Each network has two outputs; one is the output embedding, and the other (orange arrows) feeds back into the next neural network, along with the next word embedding. Each output embedding contains information about the word itself and its context in the preceding sentence fragment. In principle, the final output contains information about the entire sentence and could be used to support classification tasks similarly to the <cls> token in a transformer encoder model. However, RNNs sometimes gradually "forget" about tokens that are further back in time.