CHAPTER 11

Transformers and large language models

Convolutional neural networks model patterns in space, and recurrent neural networks model patterns in time. In both cases, the models look for locality, and by doing so the number of model parameters θ is kept relatively small, and can be made independent of the size n of the dataset. This makes CNNs and RNNs efficient to describe, and helps make them trainable. However, these models struggle with patterns that are extended in space or time; for example, how do we account for the influence of words that appear early on in a sentence on much later parts? In these models, when addressing such problems using long range patterns, the gradients calculated during backpropagation often are too small to stably update the value of a weight.

A different approach to representing patterns in space and time is provided by the *transformer* architecture, in which patterns are specified by weight matrices that are learned, using a mechanism known as *attention*. In this chapter, we describe transformers from the bottom up, starting with the idea of vector embeddings and tokens (Section 11.1), then the attention mechanism (Section 11.2) at the heart of each transformer. We then assemble all these ideas together to arrive at the full transformer architecture in Section 11.3.

11.1 Vector embeddings and tokens

Consider the challenge of understanding human language. Words that appear to be very different may have quite similar semantics to a listener, depending on the context in which they are used. For example, "sweet" is to "sugar" as "savory" is to "salt", and it is desirable to be able to capture such semantic relationships by mapping words to numerical vectors, e.g., representing word w with a d-dimensional vector v_w. The multidimensional vectors capture the fact that some words statistically occur closer to other words in written text. For example, "quantum" likely appears more often in the vicinity of "physics" than near "weather," whereas "rain" and "weather" are more likely to occur in proximity to each other.

Remarkably, the statistical likelihood of words appearing together can produce a vector mapping with the property that distances between vectors capture semantic similarities between words. In particular, if u and v are column vectors corresponding to two words, then the *cosine* similarity S_C between them is given by

$$S_C = \frac{u^T v}{|u|\,|v|} = \cos\alpha\,, \tag{11.1}$$

where $|u|$ and $|v|$ are the lengths of the vectors, and α is the angle between them. The cosine similarity is $+1$ when $u = v$, zero when the two vectors are perpendicular to each other, and -1 when the two vectors are diametrically opposed to each other. Thus, higher values of S_C correspond to closer semantic meaning of the words corresponding to the vectors.

The Euclidean distance between vector representations of words may also capture semantic meaning. To some extent, if $v_{\texttt{word}}$ is the vector for `word`, then

$$v_{\texttt{paris}} - v_{\texttt{france}} + v_{\texttt{italy}} \approx v_{\texttt{rome}} \tag{11.2}$$

since the idea might be that since the capital city of France is Paris, then the vector representations may capture the fact that the capital city of Italy is Rome.

The mapping of a word to a semantically meaningful vector is known as a *word embedding*, and is often referred to in the field as *word2vec*, following one of the early methodologies. How good is a given embedding? This can be tricky, because early methods assign the same embedding to a word independent of context. For example, the word "bank" has very different meaning when used as "river bank" versus "bank audit."

Nevertheless, to some extent, each embedding is arbitrary, and there are common methods by which good embeddings are obtained. For example, an embedding may be considered good if it accurately captures the conditional probability for a given word to appear next in a sequence of words. You probably have a good idea of what words might typically fill in the blank at the end of this sentence:

After the rain, the grass was ___________

Or a model could be built that tries to correctly predict words in the middle of sentences:

The child fell ___________ during the long car ride

The model can be built by minimizing a loss function that penalizes incorrect word guesses, and rewards correct ones. This is done on a very large corpus of written material, such as all of Wikipedia, or even all the accessible digitized written materials produced by humans.

The skip-gram model gives an explicit conceptual example of this approach, using an objective function $J(\theta)$ that is based on the conditional probability $p(w_{t+j}|w_t)$ for word w_{t+j} to appear j words away from the word w_t at location t in a corpus of words. Specifically, the objective function to be minimized is the average of the logarithms of the conditional probabilities:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{|j|\leqslant c, j\neq 0} \log\, p(w_{t+j}|w_t)\,, \tag{11.3}$$

where $w_1, w_2, \cdots w_T$ is the sequence of training words, and c is the maximum "skip" distance being considered. The conditional probabilities are determined by a softmax over the vector embeddings:

$$p(w_{t+j}|w_t) = \frac{\exp\left(v_{w_{t+j}}^T v_{w_j}\right)}{\sum_w \exp\left(v_w^T v_w\right)}\,, \tag{11.4}$$

where the denominator normalizes the softmax output to be between zero and one, and the sum is over all the k words in the vocabulary. In its simplest form, the d-dimensional vector v_w embedding word w may be computed through a simple matrix multiplication

$$v_w = \underbrace{\begin{bmatrix} | & | & \\ v_{w_1} & v_{w_2} & \cdots \\ | & | & \end{bmatrix}}_{W^T} \underbrace{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \end{bmatrix}}_{I(w)}, \tag{11.5}$$

where $I(w) \in \mathbb{R}^{k\times 1}$ is the one-hot encoding of word w, and the weight matrix $W \in \mathbb{R}^{k\times d}$ (not to be confused with word w) maps words to vector embeddings, by virtue of each column of W^T being the vector embedding of a particular word. More complex models for encoding words into vectors can of course be employed; generally this is done using a multi-layer neural network, where θ captures all of the model parameters employed for the encoding model.

Note that the skip-gram model is elegant but overly simplistic. In particular, in the form presented, this model is unfortunately impractical to train because computing the derivative of the logarithm of $p(w_{t+j}|w_t)$ entails a cost proportional to the size of the vocabulary, that can have millions of terms. However, in practice it is reasonable to prune the set of words that are included in the conditional probability calculation, because there may be a-priori information about which words are very unlikely or more likely to be relevant.

Moreover, in general it is desirable to build vector embeddings not just for words, but instead for *tokens*. For language models, tokens may be subparts of words, such as common groupings of characters that represent phonemes. Also, vector embeddings are useful for a wide range of data beyond text. Modern machine learning analyses of images, audio, and many other systems (robot dynamics!) now often begin with building representations of the data using vectorized tokens.

11.2 Attention

We have seen how CNN and RNN structures model spatial patterns in images and causal sequences in temporal data. Suppose we want to model more general patterns that have neither spatial or temporal regularity, but instead are much more complex. One remarkably effective approach to accomplish this utilizes a model known as the *attention mechanism*.

Consider a dictionary with keys κ mapping to some values $v(\kappa)$. For example, let κ be the name of some foods, such as `pizza`, `apple`, `sandwich`, `donut`, `chili`, `burrito`, `sushi`, `hamburger`, ... The corresponding values may be information about the food, such as where it is available, how much it costs, or what its ingredients are.

Suppose that instead of looking up foods by a specific name, we wanted to query by cuisine, e.g., "`mexican`" foods. And say what we want is a probability distribution over the foods, $p(\kappa|q)$ indicating which are best matches for a given query q. Clearly, we cannot simply look for the word "`mexican`" among the dictionary keys, since that word is not a food.

What does work is to utilize vector embeddings of the query and the keys! With these, we can look for keys that are semantically close to the given query. In particular, let $\bar{\kappa} \in \mathbb{R}^{d_k\times n_\kappa}$ be a matrix of the one-hot encodings of all the n_κ keys, where d_k is the size of the vocabulary. And suppose we encode these as vectors using the encoder matrix $W \in R^{d_k\times d}$, such that

$$K = W^T\bar{\kappa} \tag{11.6}$$

is a $\mathbb{R}^{d\times n_\kappa}$ matrix of vector encodings of the keys. Then given the $\mathbb{R}^{d\times 1}$ encoding $Q = W^T q$ for query $q \in \mathbb{R}^{n_\kappa \times 1}$ (which is a one-hot encoding of a word) similarly, we can lookup semantically close keys by computing the $R^{n_\kappa \times 1}$ vector $K^T Q$. The elements of this vector give the cosine similarities between the query and the keys. We then obtain the desired probability distribution using a softmax (see Chapter 7) applied to the cosine similarity vector:

$$p(\kappa|q) = \text{softmax}\left(K^T Q\right) . \tag{11.7}$$

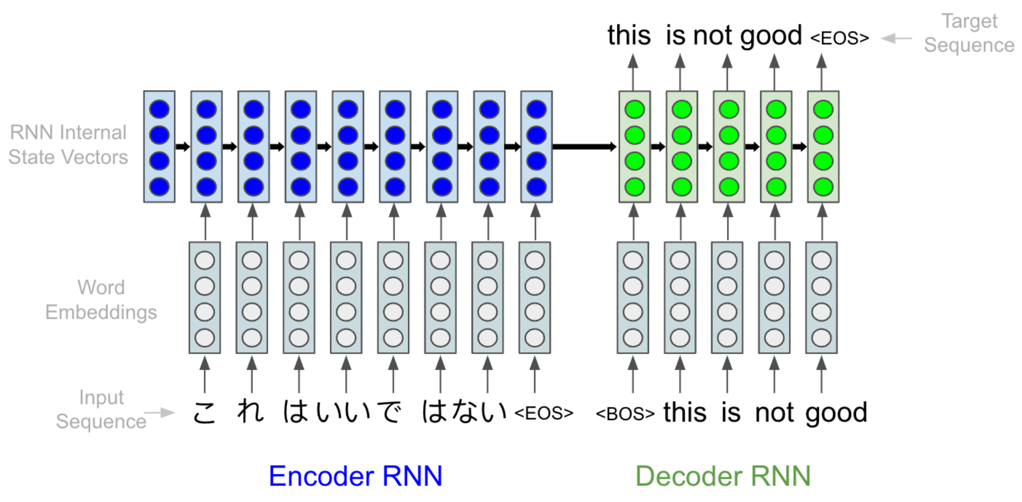
This vector-based lookup mechanism has come to be known as "attention" in the sense that $p(\kappa|q)$ is a conditional probability distribution that says which keys κ should be given attention for a given query q. The $K^T Q$ term is known as a "dot product" attention function and there are other variants that involve more complex attention functions. The conditional probability distribution $p(\kappa|q)$ gives the "attention weights," and the weighted average value

$$\begin{aligned} \text{attention}(q, \kappa, \nu) &= \sum_\kappa p(\kappa|q)\, \nu(\kappa) && (11.8) \\ &= \text{softmax}\left(K^T Q\right)\, \nu(\kappa) && (11.9) \end{aligned}$$

is the "attention output." The meaning of this weighted average value may be ambiguous when the values are just words. However, the attention output really becomes meaningful when the value are vectors in some large space, such as in the word2vec semantic embedding example given above.

11.2.1 Towards language models with attention

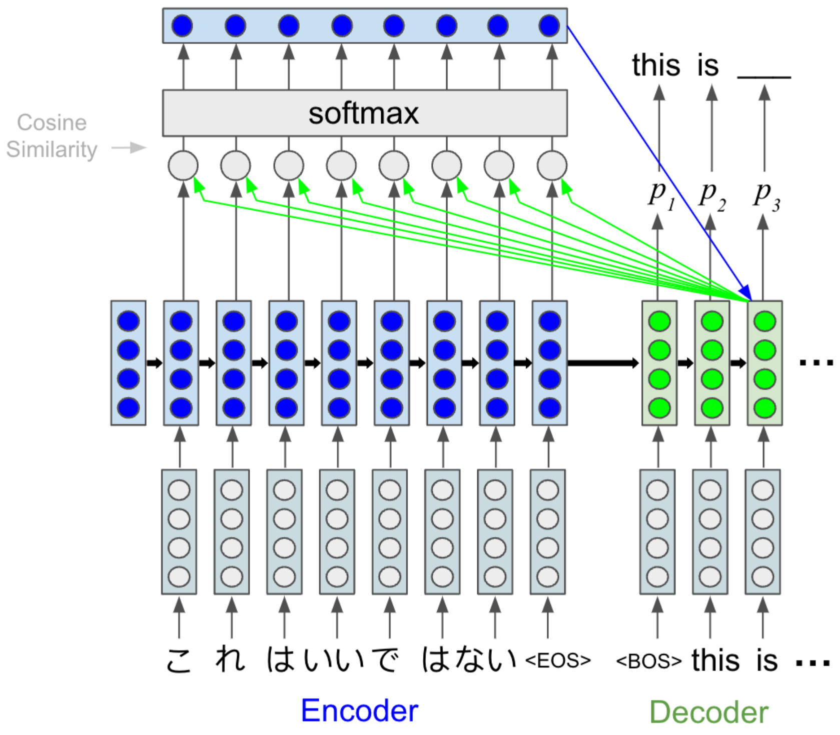
A *language model*, at its simplest level, predicts the next word to come in a sequence of words, given previous words in the sequence. For example, consider a prompt and a response. For example, say the prompt is a phrase in Japanese, "ko re wa ii de wa nai," and the desired response is its translation into English. One way to perform this translation task uses two RNN's in the form we previously saw in Section 10.3, where an encoder RNN generates a vector embedding of the prompt, then a decoder RNN generates the response given the final internal state of the encoder RNN:



Each word is first turned into a vector embedding, then processed by the RNN. The input sequence is the prompt, and the encoder RNN processes this input until it receives a special "end of sequence" (EOS) symbol. The final state of the encoder RNN is then fed into the decoder RNN, which starts with the "begin of sequence" (BOS) symbol. The decoder RNN is trained to output one word of the target sequence at each time step. The output word at time t is fed in as the input to the decoder RNN for time $t + 1$, then the decoder RNN is used to generate the output word at time $t + 1$.

This can work pretty well for short prompts. The challenge comes when the input sequence is long, because all the information about the prompt must be encoded in the final state of the encoder RNN. And the state of an RNN typically becomes increasingly uncorrelated with a given input the further away it is in time. For language translation, this is a pressing problem when the two languages have different grammar. In Japanese, the sentiment of a sentence can be completely changed by the suffix added to the verb at the very end of the sentence. This means that to accurately translate Japanese to English, the encoder RNN in our example needs to know both the initial and final Japanese words. But the RNN will perform increasingly poorly as the input sentence gets longer.

Alternatively, an attention mechanism can be employed. Conceptually, what we want is for the current state of the decoder RNN to be able to pay attention to a specific state vector of the encoder RNN, instead of just looking at the encoder RNN's final state. For example, after outputting the word "is," the decoder RNN in our Japanese translation scenario would want to know most about encoder RNN's state specifically at the word "nai" in the input, since that would generate the following English word in the output, "not." We can accomplish this by employing the state of the encoder RNN as being both the keys and values, and letting the current state of the decoder RNN be the query, using this conceptual structure:



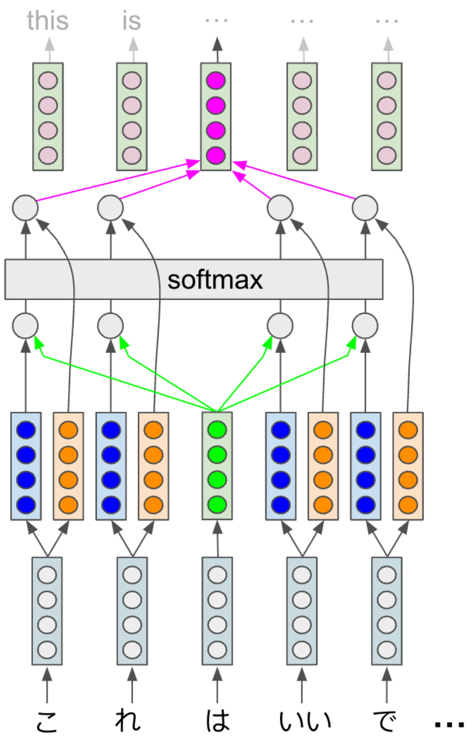
This is the conceptual structure of a seq2seq RNN with attention. The top vector at the output of the softmax is the attention output, and this output is then used together with the decoder RNN's current state to produce the next output word.

This usage of an attention mechanism bestows the RNNs with a kind of long-term memory, since generation of the output word can now utilize information from any word in the input sequence that is used to generate the attention output. There are more sophisticated structures than the simple conceptual example given above. For instance, the keys and values could use different vector embeddings, and an embedding could be applied to the values used to generate the attention output. How the decoder employs the attention output to generate the next word can also be sophisticated. And attention mechanisms are valuable not just for sequential data, but also for spatial data such as images. Note that all of these embeddings and generation processes are generally neural networks that are trainable using back-propagation. Actually training these embeddings can be difficult in practice, because of the large number of degrees of freedom, particularly as the number of keys and values grows. But the main point to take away from this discussion is that attention mechanisms allows modeling of a far richer set of patterns than the temporally local RNN models, or even the spatially local CNN models, when applied to images.

11.3 Transformers

Looking at the structure of the seq2seq RNN with attention, you might wonder why the RNN is needed, given the attention mechanism. In fact, the RNN structure is not needed, and can be replaced with a suitable combination of attention mechanisms, using a structure known as a "transformer." At the heart of the transformer is the idea of self-attention.

Self-attention is an attention mechanism where the keys, values, and queries are all generated from the same input. The idea is that for some input x, we generate a key $\kappa = (W^k)^T x$ using a weight matrix W^k, a value $v = (W^v)^T x$ with weight matrix W^v, and a query using $(W^q)^T x$. The attention mechanism then uses the attention function $\kappa^T q$ to determine which inputs are most relevant to a given q. In the context of our language translation example, this means that attention is used to determine which tokens (moving forward, let us think more generally, focusing on tokens instead of words) are most strongly related to each other. The attention output is then used to produce an output token. The neural network structure looks like this:



This diagram shows the middle input token generating a query that is then combined with the keys computed with all the other tokens to generate the attention weights via a softmax. The output of the softmax is then combined with values computed from the other tokens, to generate the attention output corresponding to the middle input token. Repeating this for each input token then generates the output.

Note that the size of the output is the same as the size of the input. Also, observe that there is no apparent notion of ordering of the input words in the depicted structure. Positional information can be added by encoding a number for token (giving say, the token's position relative to the start of the sequence) into the vector embedding of each token. And note that a given query need not pay attention to all other tokens in the input; in this example, the token used for the query is not used for a key or value.

More generally, a *mask* may be applied to limit which tokens are used in the attention computation. For example, one common mask limits the attention computation to tokens that occur previously in time to the one being used for the query. This prevents the attention mechanism from "looking ahead" in scenarios where the transformer is being used to generate one token at a time.

This single stage of self-attention is the principal component of a *transformer block*, and it is meaningfully different from an RNN: the transformer does not have a recurrent structure. Thus, back-propagation-though-time does not need to be used, and the vanishing gradients problem of RNN's does not arise in the same way. Moreover, all input tokens can be processed at once, instead of one at a time. This means a transformer can be much faster to train than a comparable RNN: instead of requiring a number of training steps that grows linearly with the size of the input, the transformer can be trained in a single step, as long as sufficient computational resources are available for the parallel processing.

Each self-attention stage is trained to have key, value, and query embeddings that lead it to pay specific attention to some particular feature of the input. We generally want to pay attention to many different kinds of features in the input; for example, in translation one feature might be be the verbs, and another might be objects or subjects. A transformer utilizes multiple instances of self-attention, each known as an "attention head," to allow combination of attention paid to many different features.

11.3.1 Formal definition of a transformer

A transformer is the composition of a number of transformer blocks, each of which has multiple heads. Rather than depicting this graphically, it is worth returning to the beauty of the underlying equations[1]. Please note that in the following, we take samples $x^{(i)}$ to be colum vectors, i.e. $\mathbb{R}^{d\times 1}$. Formally, a transformer block is a parameterized function f_θ that maps $\mathbb{R}^{d\times n} \to R^{d\times n}$, where n is the number of tokens, d is the dimension of each token, and θ are the model parameters. If $x \in \mathbb{R}^{d\times n}$ then $f_\theta(x) = z$ where queries, keys, and values are embedded via encoding matrices:

$$\begin{aligned} Q^{(h)}(x^{(i)}) &= (W_{h,q})^T x^{(i)} && (11.10)\\ K^{(h)}(x^{(i)}) &= (W_{h,k})^T x^{(i)} && (11.11)\\ V^{(h)}(x^{(i)}) &= (W_{h,v})^T x^{(i)} && (11.12) \end{aligned}$$

and $W_{h,q}, W_{h,k}, W_{h,v} \in \mathbb{R}^{d\times d_k}$, where d_k is the size of the key space, $i \in \{1, \cdots, n\}$ is an integer index over tokens, and $h \in \{1, \cdots, H\}$ is an index over "transformer heads." The attention weights $\alpha_{ij}^{(h)}$ for head h, query i and key j are then given by

$$\alpha_{ij}^{(h)} = \text{softmax}_j \left(\frac{Q^{(h)}(x^{(i)})^T K^{(h)}(x^{(j)})}{\sqrt{d_k}} \right) . \qquad (11.13)$$

[1]The presentation here follows the lovely notes by John Thickstun, rewritten into the standard notational conventions of 6.390.

Here, softmax_j is a softmax over the d_k-dimensional vector indexed by j, so in Eq. 11.13 this means a softmax computed over keys. In this equation, the normalization by $\sqrt{d_k}$ is done to reduce the magnitude of the dot product, which would otherwise grow undesirably large with increasing d_k.

The un-normalized attention outputs are a weighted sum over the attention outputs for each head,

$$u'^{(i)} = \sum_{h=1}^{H} W_{h,c}^T \sum_{j=1}^{n} \alpha_{ij}^{(h)} V^{(h)}(x^{(j)}), \tag{11.14}$$

where $W_{h,c} \in \mathbb{R}^{d_k \times d}$. This is standardized and combined with $x^{(i)}$ using a LayerNorm function (defined below) to become

$$u^{(i)} = \text{LayerNorm}\left(x^{(i)} + u'^{(i)}; \gamma_1, \beta_1\right) \tag{11.15}$$

with parameters $\gamma_1, \beta_1 \in \mathbb{R}^d$. The un-normalized transformer block output $z'^{(i)}$ is then given by

$$z'^{(i)} = W_2^T \text{ReLU}\left(W_1^T u^{(i)}\right) \tag{11.16}$$

with weights $W_1 \in \mathbb{R}^{d \times m}$ and $W_2 \in \mathbb{R}^{m \times d}$. This is then standardized and combined with $u^{(i)}$ to give the final output $z^{(i)}$:

$$z^{(i)} = \text{LayerNorm}\left(u^{(i)} + z'^{(i)}; \gamma_2, \beta_2\right), \tag{11.17}$$

with parameters $\gamma_2, \beta_2 \in \mathbb{R}^d$. These vectors are then assembled (e.g., through parallel computation) to produce $z \in \mathbb{R}^{n \times d}$.

The LayerNorm function transforms a d-dimensional input z with parameters $\gamma, \beta \in \mathbb{R}^d$ into

$$\text{LayerNorm}(z; \gamma, \beta) = \gamma \frac{z - \mu_z}{\sigma_z} + \beta, \tag{11.18}$$

where μ_z is the mean and σ_z the standard deviation of z:

$$\mu_z = \frac{1}{d} \sum_{i=1}^{d} z_i \tag{11.19}$$

$$\sigma_z = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (z_i - \mu_z)^2}. \tag{11.20}$$

Layer normalization is done to improve convergence stability during training.

The model parameters θ comprise the weight matrices $W_{h,q}, W_{h,k}, W_{h,v}, W_{h,c}, W_1, W_2$ and the LayerNorm parameters $\gamma_1, \gamma_2, \beta_1, \beta_2$. A *transformer* is the composition of L transformer blocks, each with its own parameters:

$$f_{\theta_L} \circ \cdots \circ f_{\theta_2} \circ f_{\theta_1}(x) \in R^{d \times n}. \tag{11.21}$$

The hyperparameters of this model are d, d_k, m, H, and L, where typically $d = 512$, $d_k = 64$, $m = 2048$, $H = 8$, and $L \geqslant 6$.

11.3.2 Variations and training

Many variants on this transformer structure exist. For example, the LayerNorm may be moved to other stages of the neural network. Or a more sophisticated attention function may be employed instead of the simple dot product used in Eq. 11.13. Transformers may also be used in pairs, for example, one to process the input and a separate one to generate the output given the transformed input. Self-attention may also be replaced with cross-attention, where some input data are used to generate queries and other input data generate keys and values. Positional encoding and masking are also common, though they are left implicit in the above equations for simplicity.

How are transformers trained? The number of parameters in θ can be very large; modern transformer models like GPT4 have tens of billions of parameters or more. A great deal of data is thus necessary to train such models, else the models may simply overfit small datasets.

Training large transformer models is thus generally done in two stages. A first "pre-training" stage employs a very large dataset to train the model to extract patterns. This is done with unsupervised (or self-supervised) learning and unlabelled data. For example, the well-known BERT model was pre-trained using sentences with words masked off. The model was trained to predict the masked words. BERT was also trained on sequences of sentences, where the model was trained to predict whether two sentences are likely to be contextually close together or not. The pre-training stage is generally very expensive.

The second "fine-tuning" stage trains the model for a specific task, such as classification or question answering. This training stage can be relatively inexpensive, but it generally requires labeled data.