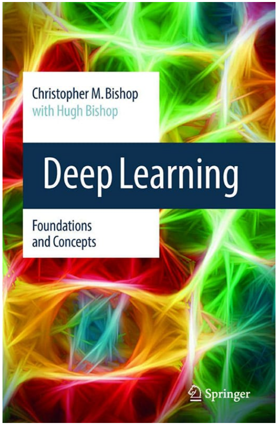
Attention, Transformers, and Large Language Models

Shuiwang Ji, Xiner Li, Shurui Gui
Department of Computer Science & Engineering
Texas A&M University

These slides are based on Chapter 12 of
Deep Learning: Foundations and Concepts

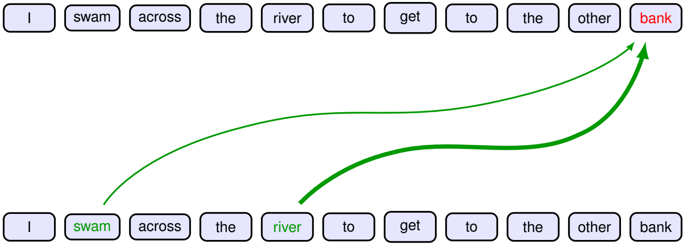


Example

I swam across the river to get to the other bank.

I walked across the road to get cash from the bank.

- Appropriate interpretation of 'bank' relies on other words from the rest of the sequence
- The particular locations that should receive more attention depend on the input sequence itself
- In a standard neural network, the weights are fixed once the network is trained
- Attention uses weights whose values depend on the specific input data

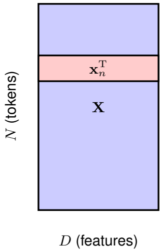


Transformer processing

The input data to a transformer is a set of vectors $\{\mathbf{x}_n\}$ of dimensionality D, where $n = 1, \ldots, N$

- Combine the data vectors into a matrix $\mathbf{X}$ of dimensions $N \times D$ in which the nth row comprises the token vector $\mathbf{x}_n^{\mathrm{T}}$, and where $n = 1, \ldots, N$ labels the rows
- A Transformer takes a data matrix as input and creates a transformed matrix $\widetilde{\mathbf{X}}$ of the same dimensionality as the output
- We can write this function in the form

$$\widetilde{\mathbf{X}} = \text{TransformerLayer}\,[\mathbf{X}]$$



Attention coefficients

- Map input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_N$ to $\mathbf{y}_1, \ldots, \mathbf{y}_N$
- The value of $\mathbf{y}_n$ should depend on all the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$
- Dependence should be stronger for important inputs
- Define each output vector $\mathbf{y}_n$ to be a linear combination of $\mathbf{x}_1, \ldots, \mathbf{x}_N$:

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm} \mathbf{x}_m$$

where

$$a_{nm} \geqslant 0, \text{and} \sum_{m=1}^{N} a_{nm} = 1.$$

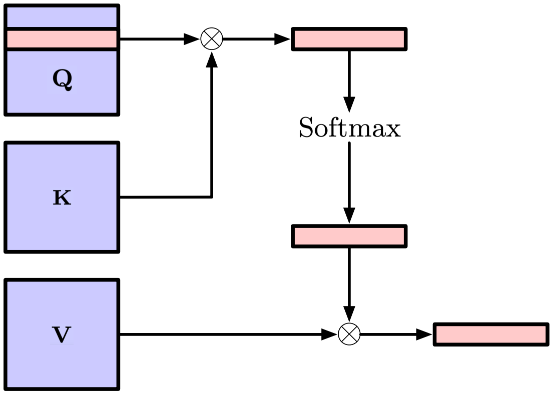
- Commonly used coefficients

$$a_{nm} = \frac{\exp\left(\mathbf{x}_n^{\mathrm{T}} \mathbf{x}_m\right)}{\sum_{i=1}^{N} \exp\left(\mathbf{x}_n^{\mathrm{T}} \mathbf{x}_i\right)}, \quad a_n = \text{Softmax} \begin{bmatrix} \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_1 \\ \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_N \end{bmatrix}$$

- We have a different set of coefficients for each output vector $\mathbf{y}_n$

Attention in general (cross attention)

- Use of query, key, and value vectors as rows of matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$

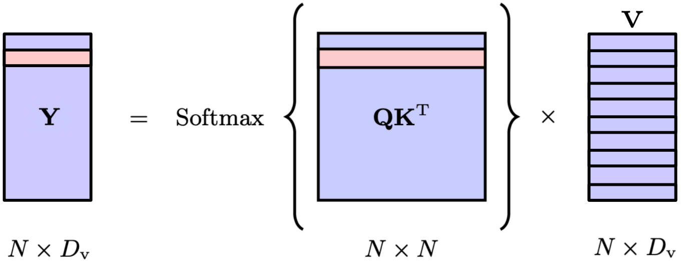


General attention in matrix form

- The attention computes

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{Q}\mathbf{K}^{\mathrm{T}}\right]\mathbf{V}$$

  where Softmax $[\mathbf{L}]$ takes the exponential of every element of $\mathbf{L}$ and then normalizes each row independently to sum to one
- Whereas standard networks multiply activations by fixed weights, here the activations are multiplied by the data-dependent attention coefficients


$\mathbf{Y}$
$=$
Softmax
$\mathbf{QK}^{\mathrm{T}}$
$\times$
$\mathbf{V}$
$N \times D_{\mathrm{v}}$
$N \times N$
$N \times D_{\mathrm{v}}$

Self-attention without parameters

- We can use data matrix $\mathbf{X}$ as $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, along with the output matrix $\mathbf{Y}$, whose rows are given by $\mathbf{y}_m$, so that

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X}$$

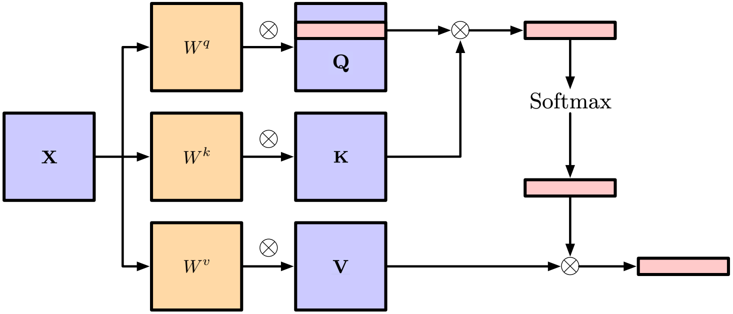
  where Softmax $[\mathbf{L}]$ takes the exponential of every element of $\mathbf{L}$ and then normalizes each row independently to sum to one
- This process is called self-attention because we are using the same sequence to determine the queries, keys, and values
- The transformation is fixed and has no capacity to learn

Self-attention with parameters

- Define

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})} \in \mathbb{R}^{N \times D_{\mathrm{k}}}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})} \in \mathbb{R}^{N \times D_{\mathrm{k}}}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})} \in \mathbb{R}^{N \times D_{\mathrm{v}}}$$

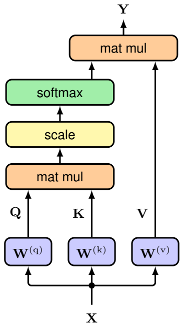
- D_{v} governs the dimensionality of the output vectors
- Setting $D_{\mathrm{v}} = D$ will facilitate the inclusion of residual connections



Dot-product scaled attention

- Let p_i denotes the i-th element of Softmax($\boldsymbol{a}$), we have $\frac{\partial p_i}{\partial a_j} = p_i(\delta_{ij} - p_j)$: small for inputs of high magnitude
- If the elements of the query and key vectors were all independent random numbers with zero mean and unit variance, then the variance of the dot product would be D_{k}
- Normalize using the standard deviation

$$\mathbf{Y} = \mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_{\mathrm{k}}}}\right]\mathbf{V}$$



Y
mat mul
softmax
scale
mat mul
Q
K
V
$\mathbf{W}^{(\mathrm{q})}$
$\mathbf{W}^{(\mathrm{k})}$
$\mathbf{W}^{(\mathrm{v})}$
X

Multi-head attention

- Suppose we have H heads indexed by $h = 1, \ldots, H$ as

$$\mathbf{H}_h = \text{Attention}\left(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h\right)$$

- Define separate query, key, and value matrices for each head using

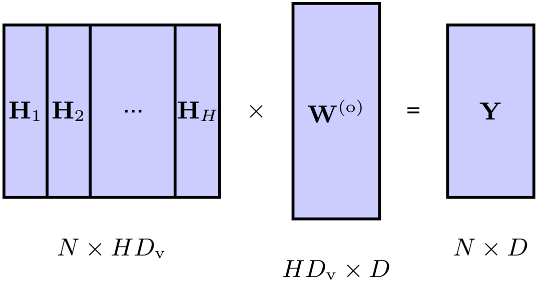
$$\begin{aligned} \mathbf{Q}_h &= \mathbf{X}\mathbf{W}_h^{(\mathrm{q})} \\ \mathbf{K}_h &= \mathbf{X}\mathbf{W}_h^{(\mathrm{k})} \\ \mathbf{V}_h &= \mathbf{X}\mathbf{W}_h^{(\mathrm{v})} \end{aligned}$$

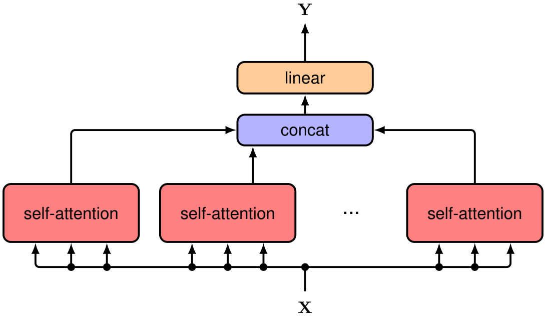
- The heads are first concatenated into a single matrix, and the result is then linearly transformed using a matrix $\mathbf{W}^{(o)}$ as

$$\mathbf{Y}(\mathbf{X}) = \text{ Concat } \left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right] \mathbf{W}^{(\mathrm{o})}$$

- Typically D_{v} is chosen to be equal to D/H so that the resulting concatenated matrix has dimension $N \times D$

Multi-head attention





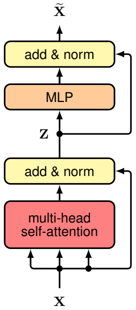
Transformer layers

- Stack multiple self-attention layers on top of each other
- Introduce residual connections and require the output dimensionality to be the same as the input dimensionality, namely $N \times D$
- Followed by layer normalization to improves training efficiency

$$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right]$$

- Sometimes the normalization layer is applied before the multi-head self-attention as

$$\mathbf{Z} = \mathbf{Y}(\text{LayerNorm}[\mathbf{X}]) + \mathbf{X}$$



MLP in Transformer layers

- The output vectors are constrained to lie in the subspace spanned by the input vectors and this limits the expressive capabilities of the attention layer
- Enhance the flexibility using a standard nonlinear neural network with D inputs and D outputs
- For example, this might consist of a two-layer fully connected network with ReLU hidden units
- This needs to preserve the ability of the transformer to process sequences of variable length
- The same shared network is applied to each of the output vectors, corresponding to the rows of Z
- This neural network layer can be improved by using a residual connection and layer normalization

Transformer layers

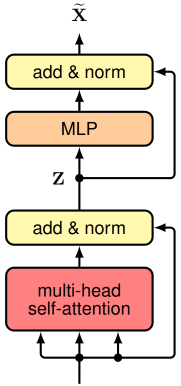
- The final output from the transformer layer has the form

$$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}[\mathbf{Z}] + \mathbf{Z}\right]$$

- Again, we can use a pre-norm as

$$\widetilde{\mathbf{X}} = \text{MLP}\left(\mathbf{Z}'\right) + \mathbf{Z}, \text{ where } \quad \mathbf{Z}' = \text{LayerNorm}\left[\mathbf{Z}\right]$$

- In a typical transformer there are multiple such layers stacked on top of each other



Positional encoding

- The transformer has the property that permuting the order of the input tokens, i.e., the rows of $\mathbf{X}$, results in the same permutation of the rows of the output matrix $\widetilde{\mathbf{X}}$ - equivariance
- The lack of dependence on token order becomes a major limitation when we consider sequential data, such as the words in a natural language
  'The food was bad, not good at all.'
  'The food was good, not bad at all.'
- Construct a position encoding vector $\mathbf{r}_n$ associated with each input position n and then combine this with the associated input token embedding $\mathbf{x}_n$
- An ideal positional encoding should provide a unique representation for each position, it should be bounded, it should generalize to longer sequences, and it should have a consistent way to express the number of steps between any two input vectors irrespective of their absolute position because the relative position of tokens is often more important than the absolute position

Natural language and word embedding

- Convert the words into a numerical representation that is suitable for use as the input to a deep neural network
- Define a fixed dictionary of words and then introduce vectors of length equal to the size of the dictionary along with a 'one hot' representation for each word
- The embedding process can be defined by a matrix $\mathbf{E}$ of size $D \times K$ where D is the dimensionality of the embedding space and K is the dimensionality of the dictionary.
- For each one-hot encoded input vector $\mathbf{x}_n$ we can then calculate the corresponding embedding vector using

$$\mathbf{v}_n = \mathbf{E}\mathbf{x}_n$$

- Word embeddings can be viewed as the first layer in a deep neural network. They can be fixed using some standard pre-trained embedding matrix, or they can be trained
- The embedding layer can be initialized either using random weight values or using a standard embedding matrix

Language models: Narrow sense

- Language models learn the joint distribution $p(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ of an ordered sequence of vectors, such as words (or tokens) in a natural language
- We can decompose the distribution into a product of conditional distributions in the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$$

- We could represent each term by a table whose entries are estimated using simple frequency counts
- However, the size of these tables grows exponentially with the length of the sequence

n-gram model and LLMs (Courtesy R. Kambhampati)

- We can assume that each of the conditional distributions is independent of all previous observations except the L most recent words, $L = 1$: bi-gram; $L = 2$: tri-gram; $L = n - 1$: n-gram
- If $L = 2$, we have

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1)\, p(\mathbf{x}_2 \mid \mathbf{x}_1) \prod_{n=3}^{N} p(\mathbf{x}_n \mid \mathbf{x}_{n-1}, \mathbf{x}_{n-2})$$

- What if $L = 0$? $p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n)$
- The size of the probability tables grows exponentially in L
- A 3,001-gram model (like ChatGPT) learns to predict the next word given the previous 3,000 words
- When $|V| = 5k$, need $50,000^{3,000}$ conditional distributions, with many zeros
- LLMs compress/approximate this gigantic table with a function
- Although LLMs have billions of parameters, they are small compared to the size of table
- LLMs Look at everything we say as a prompt to be completed

Language models: Broad sense

- Encoder only: In sentiment analysis, we take a sequence of words as input and provide a single variable representing the sentiment of the text. Here a transformer is acting as an 'encoder' of the sequence
- Decoder only: Take a single vector as input and generate a word sequence as output, for example if we wish to generate a text caption given an input image. In such cases the transformer functions as a 'decoder', generating a sequence as output
- Encoder-Decoder: In sequence-to-sequence processing tasks, both the input and the output comprise a sequence of words, for example if our goal is to translate from one language to another. In this case, transformers are used in both encoder and decoder roles

Decoder transformers

- Focus on a class of models called GPT which stands for generative pretrained transformer
- Use the transformer to construct an autoregressive model in which the conditional distributions $p\left(\mathbf{x}_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}\right)$ are expressed using a transformer
- The model takes as input a sequence consisting of the first $n-1$ tokens, and its corresponding output represents the conditional distribution for token n
- Draw a sample from this distribution then we have extended the sequence to n tokens and this new sequence can be fed back through the model to give a distribution over token $n+1$

Decoder transformers

- The GPT model consists of a stack of transformer layers that take $\mathbf{x}_1, \ldots, \mathbf{x}_N$ as input and produce $\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_N$
- Each output needs to represent a distribution over the dictionary with dimensionality K whereas the tokens have a dimensionality of D
- Make a linear transformation of each output token using a matrix $\mathbf{W}^{(\mathrm{p})}$ of $D \times K$ followed by a softmax as

$$\mathbf{Y} = \mathrm{Softmax}\left(\widetilde{\mathbf{X}}\mathbf{W}^{(\mathrm{p})}\right)$$

  where $\mathbf{Y}$ is a matrix whose nth row is $\mathbf{y}_n^{\mathrm{T}}$, and $\widetilde{\mathbf{X}}$ is a matrix whose nth row is $\widetilde{\mathbf{x}}_n^{\mathrm{T}}$
- Each softmax output unit has an associated cross-entropy error function

Decoder transformers: concurrent processing of sequence

I swam across the river to get to the other bank.

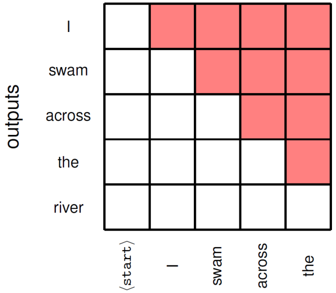
- The model can be trained using a large corpus of unlabeled natural language by taking a self-supervised approach
- Each training sample consists of a sequence of tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$ as input and $\mathbf{x}_{n+1}$ as output
- Can achieve much greater efficiency by processing an entire sequence at once so that each token acts both as a target value for the sequence of previous tokens and as an input value for subsequent tokens
- We can use 'I swam across' as an input with a target of 'the', and also use 'I swam across the' as an input sequence with an associated target of 'river', and so on
- Ensure that the network is not able to 'cheat' by looking ahead in the sequence, otherwise, cannot generate

Decoder transformers: causal language modeling

I swam across the river to get to the other bank.

- First, we shift the input sequence to the right by one step, so that input $\mathbf{x}_n$ corresponds to output $\mathbf{y}_{n+1}$ (predicted prob of $\mathbf{x}_{n+1}$), with target $\mathbf{x}_{n+1}$
- Tokens interact only via attention weights
- Second, use causal (~~masked~~) attention, in which we set to zero all of the attention weights that correspond to a token attending to any later token in the sequence (red) and then normalize the remaining elements

Figure 12.16 An illustration of the mask matrix for masked self-attention. Attention weights corresponding to the red elements are set to zero. Thus, in predicting the token 'across', the output can depend only on the input tokens '⟨start⟩' 'I' and 'swam'.



Decoder transformer architecture

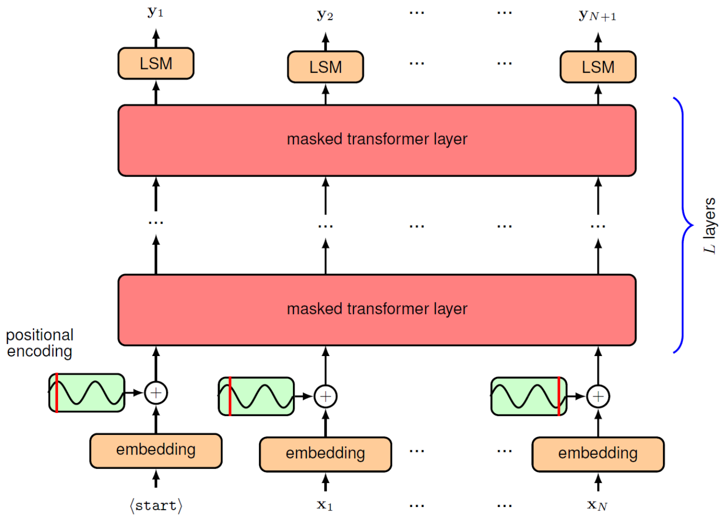


Figure 12.15 Architecture of a GPT decoder transformer network. Here 'LSM' stands for linear-softmax and denotes a linear transformation whose learnable parameters are shared across the token positions, followed by a softmax activation function. Masking is explained in the text.

Difference between training and generation/inference

- Training: the next token is given, so it is a multi-class prediction problem using cross-entropy loss
- Generation/inference: sample a token based on the computed probability, and use it as input to the network to compute the probability of the next token
- Challenge: during the learning phase, the model is trained on a human-generated input sequence, whereas when it is running generatively, the input sequence is itself generated from the model. This means that the model can drift away from the distribution of sequences seen during training

Sampling strategies during generation/inference

- The output of a decoder transformer is a probability distribution over values for the next token
- Greedy search: select the token with the highest probability - deterministic
- Simply choosing the highest probability token at each stage is not the same as selecting the highest probability sequence of tokens - why?

$$p(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \prod_{n=1}^{N} p(\mathbf{y}_n \mid \mathbf{y}_1, \ldots, \mathbf{y}_{n-1})$$

- Beam search: maintain a set of B hypotheses, each consisting of a sequence of token values up to step n
- Feed all these sequences through the network, and for each sequence we find the B most probable token values, thereby creating B^2 possible hypotheses for the extended sequence
- This list is then pruned by selecting the most probable B hypotheses according to the total probability of the extended sequence

Sampling strategies during generation/inference

- One problem with approaches such as greedy search and beam search is that they limit the diversity of potential outputs
- Generate successive tokens simply by sampling from the softmax distribution at each step, or sample from top-K
- Introduce a parameter T called temperature into softmax

$$y_i = \frac{\exp\left(a_i/T\right)}{\sum_j \exp\left(a_j/T\right)}$$

- $T \to 0$: the probability mass is concentrated on the most probable state - greedy selection
- $T = 1$: the unmodified softmax distribution
- $T \to \infty$, uniform across all states
- $0 < T < 1$: the probability is concentrated towards the higher values
- If such approaches are used, there is randomness in generation

Encoder transformers: Masked language modeling

- Take sequences as input and produce fixed-length vectors, such as class labels, as output
- An example of such a model is BERT, which stands for bidirectional encoder representations from transformers
- A randomly chosen subset of the tokens, say 15%, are replaced with a special token denoted $\langle$ mask $\rangle$
- The model is trained to predict the missing tokens

  I $\langle$ mask $\rangle$ across the river to get to the $\langle$ mask $\rangle$ bank.
- The network should predict 'swam' at output node 2 and 'other' at output node 10
- Only two of the outputs contribute to the error function and the other outputs are ignored
- BERT is 'bidirectional', so no need to shift inputs and mask outputs
- An encoder model is unable to generate sequences

Encoder transformer architecture

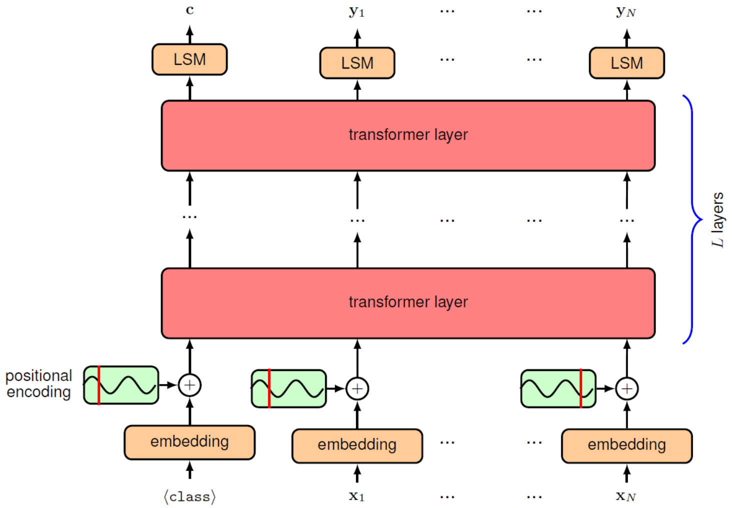


Figure 12.18 Architecture of an encoder transformer model. The boxes labelled 'LSM' denote a linear transformation whose learnable parameters are shared across the token positions, followed by a softmax activation function. The main differences compared to the decoder model are that the input sequence is not shifted to the right, and the 'look ahead' masking matrix is omitted and therefore, within each self-attention layer, every output token can attend to any of the input tokens.

Sequence-to-sequence transformers

- Consider the task of translating an English sentence into a Dutch sentence
- We can use a decoder model to generate the token sequence corresponding to the Dutch output, token by token
- The main difference is that this output needs to be conditioned on the entire input sequence
- An encoder transformer can be used to map the input token sequence into a suitable internal representation, denote by $\mathbf{Z}$
- To incorporate $\mathbf{Z}$ into the generative process, we use cross attention
- **The query vectors come from the sequence being generated, in this case the Dutch output sequence, the key and value vectors come from the sequence represented by $\mathbf{Z}$**
- The model can be trained using paired input and output sentences

Comparison of self and cross attention

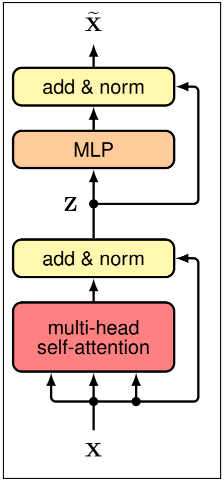
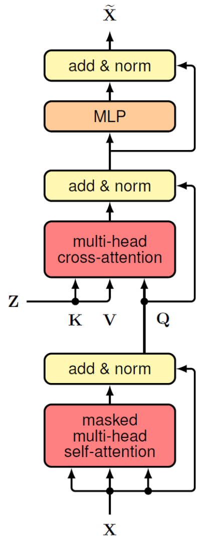
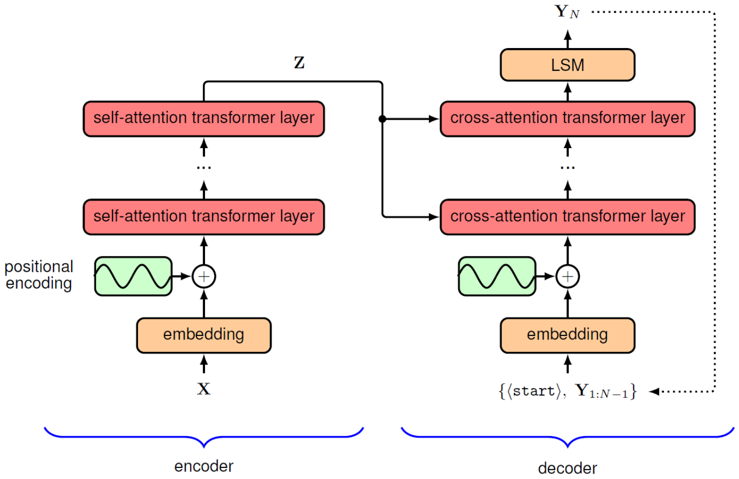


Figure 12.19 Schematic illustration of one cross-attention layer as used in the decoder section of a sequence-to-sequence transformer. Here $\mathbf{Z}$ denotes the output from the encoder section. $\mathbf{Z}$ determines the key and value vectors for the cross-attention layer, whereas the query vectors are determined within the decoder section.



Sequence to sequence transformer architecture



$\mathbf{Y}_N$
LSM
$\mathbf{Z}$
self-attention transformer layer
cross-attention transformer layer
...
self-attention transformer layer
cross-attention transformer layer
positional encoding
embedding
embedding
$\mathbf{X}$
$\{\langle\texttt{start}\rangle, \mathbf{Y}_{1:N-1}\}$
encoder
decoder

Figure 12.20 Schematic illustration of a sequence-to-sequence transformer. To keep the diagram uncluttered the input tokens are collectively shown as a single box, and likewise for the output tokens. Positional-encoding vectors are added to the input tokens for both the encoder and decoder sections. Each layer in the encoder corresponds to the structure shown in Figure 12.9, and each cross-attention layer is of the form shown in Figure 12.19.

Large language models: Pretraining

- The number of compute operations required to train a state-of-the-art machine learning model has grown exponentially since about 2012 with a doubling time of around 3.4 months
- Increasing the size of the training data set, along with increase in model parameters, leads to improvements in performance
- The impressive increase in performance of the GPT series of models through successive generations has come primarily from an increase in scale
- LLMs are trained by self-supervised learning on very large data sets of text
- A decoder transformer can be trained on token sequences in which each token acts as a labelled target example
- This 'self-labelling' hugely expands the quantity of training data available and therefore allows exploitation of deep neural networks having large numbers of parameters

Large language models: fine tuning

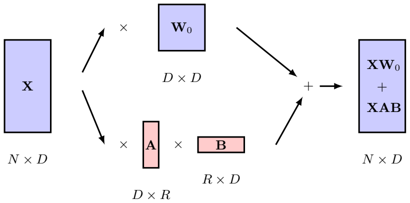
- This use of self-supervised learning led to a paradigm shift in which a large model is first pre-trained using unlabelled data and then subsequently fine-tuned using supervised learning based on a much smaller set of labelled data
- A model with broad capabilities that can be subsequently fine-tuned for specific tasks is called a foundation model
- Can freeze parts of parameters while updating others during fine-tuning
- Low-rank adaptation or LoRA: changes in the model parameters during fine-tuning lie on a manifold whose dimensionality is much smaller than the total number of learnable parameters in the model
- LoRA freezes the original model and adding additional learnable matrices into each layer of the transformer in the form of low-rank products

Large language models: fine tuning with LoRA

- Consider a weight matrix $\mathbf{W}_0$ having dimension $D \times D$, we introduce a parallel set of weights defined by the product of two matrices $\mathbf{A}$ and $\mathbf{B}$ with dimensions $D \times R$ and $R \times D$, respectively
- This layer then generates an output $\mathbf{X}\mathbf{W}_0 + \mathbf{X}\mathbf{A}\mathbf{B}$
- The number of parameters in $\mathbf{AB}$ is $2RD$ compared to the D^2 in $\mathbf{W}_0$
- If $R \ll D$ then the number of parameters that need to be adapted during fine-tuning is much smaller
- Once the fine-tuning is complete, the additional weights can be added to the original weight matrices to give a new weight matrix

$$\widehat{\mathbf{W}} = \mathbf{W}_0 + \mathbf{AB}$$

- During inference there is no additional computational overhead



Large language models: Emerging properties

- As language models have become larger and more powerful, the need for fine-tuning has diminished, with generative language models now able to solve a broad range of tasks simply through text-based interaction
- For example, if a text string

  English: the cat sat on the mat. French:

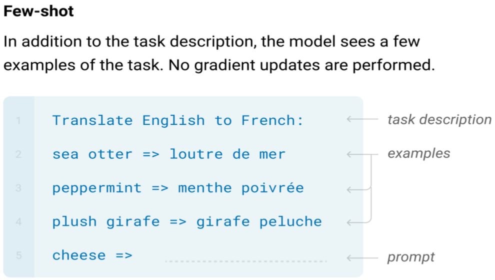
  is given as the input sequence, an autoregressive language model can generate subsequent tokens representing the French translation
- The model was not trained specifically to do translation but has learned to do so as a result of being trained on a vast corpus of data that includes multiple languages - Emerging properties

Large language models: RLHF

- To improve the user experience and the quality of the generated outputs, techniques have been developed for fine-tuning large language models through human evaluation of generated output, using methods such as reinforcement learning through human feedback or RLHF
- Such techniques have helped to create large language models with impressively easy-to-use conversational interfaces, most notably the system from OpenAI called ChatGPT

Large language models: Prompting

- The sequence of input tokens given by the user is called a prompt
- By using different prompts, the same trained neural network may be capable of solving a broad range of tasks
- The performance of the model now depends on the form of the prompt, leading to a new field called prompt engineering
- This allows the model to solve new tasks simply by providing some examples within the prompt, without needing to adapt the parameters of the model. This is an example of few-shot learning[1]



Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
Translate English to French:
sea otter => loutre de mer
peppermint => menthe poivrée
plush girafe => girafe peluche
cheese =>
```

[1] https://thegradient.pub/in-context-learning-in-context/

Interested in learning more?

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, Jianfeng Gao: Large Language Models: A Survey
https://arxiv.org/abs/2402.06196

THANKS!