Outline

1. Network architecture
2. Transform blocks and other details
3. Masked self-attention
4. Prediction

Outline

1. Network architecture
2. Transform blocks and other details
3. Masked self-attention
4. Prediction

Network Design I

- Recall that we hope to have a function f that can efficiently calculate

$$f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:1}), \ldots, f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:j}), \ldots$$

as shown in the following figure.

Network Design II

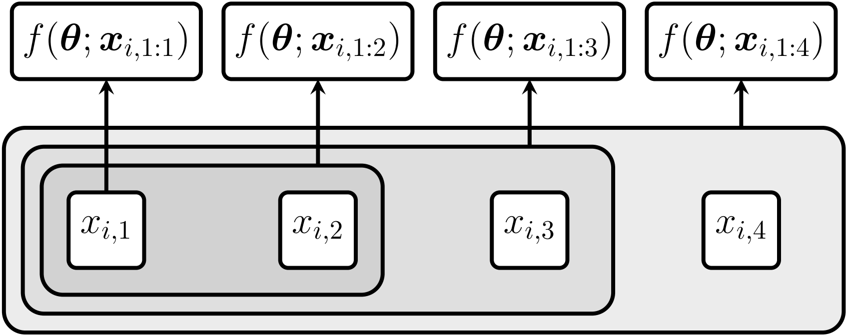


Figure 1: A sequence of next-token predictions

Network Design III

- The situation is similar to the least-square approximation discussed earlier for time-series prediction.
- The difference is that instead of a linear function, here we use a more complicated one.
- Different types of neural networks can serve as our f.
- For example, we can modify convolutional networks as the main component of our f.

  See details in, for example, https://dlsyscourse.org/slides/transformers.pdf.

Network Design IV

- We will describe the most used network for LLMs: transformer.
- However, it is possible that in the future we can develop better networks.

Overall Architecture I

- The architecture used by LLMs contains several transformer blocks.
- Transformer (Vaswani et al., 2017) is an effective network for many applications.
- The overall architecture of an LLM is as follows.

Overall Architecture II

SoftMax: next token probability

$\uparrow$

Linear layer

$\uparrow$

Several transformer blocks

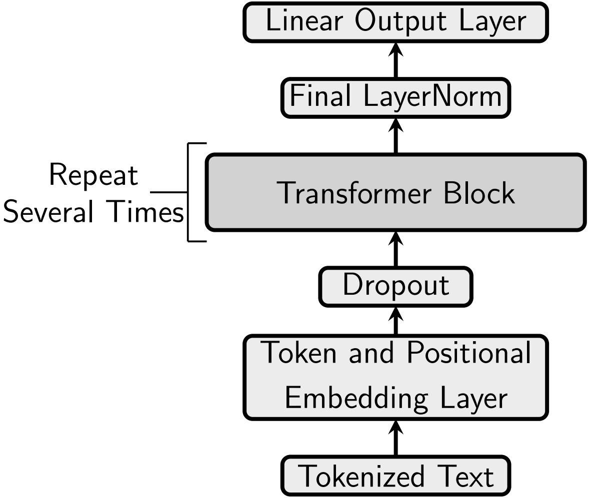
$\uparrow$

Token + position embedding

Overall Architecture III

- We specifically discuss the architecture used in the implementation in GPT-2-small (Radford et al., 2019). Their code is at https://github.com/openai/gpt-2/blob/master/src/model.py.
- Similar architectures have been adopted in other places, such as NanoGPT (https://github.com/karpathy/nanoGPT).
- Precisely, what GPT-2 does is the following network.

Overall Architecture IV



Overall Architecture V

- For the initialization of model weights, some common ways are available. For example, we can randomly draw values from normal distribution with zero mean.

Input and Embedding Vectors I

- Now let us discuss the input of the network.
- To begin, we assume that
  - each word (token) in our Vocabulary corresponds to an embedding vector, and
  - each position of $1, \dots, T$ corresponds to an embedding vector.
- We then combine these two embedding vectors as one vector.
- Various ways are possible for the combination. For example, we can concatenate the two vectors as a longer one. Or we can sum up the two vectors.

Input and Embedding Vectors II

- Then $\boldsymbol{x}_{1:T}$ becomes the following matrix.

$$\begin{array}{c} x_1 \\ \vdots \\ x_T \end{array} \begin{bmatrix} \text{------} \\ \vdots \\ \text{------} \end{bmatrix} \in \mathbf{R}^{T \times d} \tag{1}$$

  where d is the dimension of the embedding vector.
- This matrix becomes the input of the architecture.
- Up to now, it seems that we assume all documents have the same length T.
- Of course, this assumption is in general untrue.
- What we do is:

Input and Embedding Vectors III

- if document length $> T$, use only the first T tokens, and
- if document length $< T$, we add "empty" values after the end of the document.

- The main reason of using a fixed document length is for easily conducting operations.
- Totally we have

$$|\text{Vocabulary}|$$

Input and Embedding Vectors IV

vectors for word embedding, and

$$T$$

vectors for position embedding.

- For each of the T words (tokens) in the document, we extract

  a word embedding vector

  and

  a position embedding vector.
- All these embedding vectors are trainable parameters.

Outline

1. Network architecture
2. Transform blocks and other details
3. Masked self-attention
4. Prediction

A Transformer Block I

- For each transformer block, we let
  - Z be the input matrix, and
  - Z^{out} be the output matrix.
- We manage to have that

$$Z^{\text{out}} \in \mathbf{R}^{T \times d}$$

has the same dimensionality as Z.

- By doing so, the output can be the input of the next block.
- We repeat this process for several blocks.

A Transformer Block II

- Typically a transformer block involves
  - a multi-head attention layer, and
  - feed-forward layers.
- Usually, we surround each of the two components with a normalization layer and a residual connection.

A Transformer Block III

- Thus the mathematical operations in a block are as follows.

$$
\begin{aligned}
\tilde{Z} &= \mathsf{LayerNorm}(Z) && (2)\\
Z &\leftarrow Z + \mathsf{DropOut}(\mathsf{MultiHead}(\tilde{Z})) && (3)\\
\tilde{Z} &= \mathsf{LayerNorm}(Z) && (4)\\
Z^{\mathsf{out}} &= Z + \mathsf{DropOut}(\mathsf{GELU}(\tilde{Z}W_1)W_2) && (5)
\end{aligned}
$$

A Transformer Block IV

- We can re-write (17)-(3) in just one line:

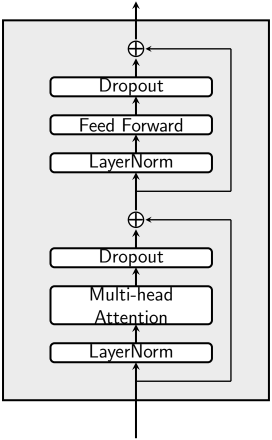
$$Z \leftarrow Z + \text{DropOut}(\text{MultiHead}(\text{LayerNorm}(Z)))$$

  but to have a symbol $\tilde{Z}$ as the input of the attention layer, we use two equations.
- Subsequently we discuss each operation in detail.
- Besides the attention layer in (3), what else in one transform block (e.g., layer normalization or dropout) may slightly vary across LLM implementations.

A Transformer Block V

- Our operations in (17)-(5) can be illustrated in the following figure.[1]

A Transformer Block VI



- In the figure, $\oplus$ means the residual connection, which will be explained later.

A Transformer Block VII

[1]Modified from
https://sebastianraschka.com/pdf/slides/2024-acm.pdf

Transformer and Attention I

- Eq. (3) is the core of a transformer block: a multi-head self-attention layer.
- We start with discussing single-head attention.
- If the input matrix is

$$\tilde{Z} \in \mathbf{R}^{T \times d},$$

the attention operation is

$$\text{SoftMax}\left(\frac{\tilde{Z} W_Q W_K^\top (\tilde{Z})^\top}{\sqrt{d}}\right) \tilde{Z} W_V. \tag{6}$$

Transformer and Attention II

- We consider three trainable weight matrices

$$W_Q \in \mathbf{R}^{d \times d}, W_K \in \mathbf{R}^{d \times d}, W_V \in \mathbf{R}^{d \times d}$$

to convert the input matrix $\tilde{Z}$ to

$$\tilde{Z}W_Q \in \mathbf{R}^{T \times d}, \quad \tilde{Z}W_K \in \mathbf{R}^{T \times d}, \quad \tilde{Z}W_V \in \mathbf{R}^{T \times d}.$$

- In (6), we can combine $W_Q W_K^\top$ as one single weight matrix.

Transformer and Attention III

- However, people still write them separately because in more general situations, we may consider

$$W_Q \in \mathbf{R}^{d \times d'} \text{ and } W_K \in \mathbf{R}^{d \times d'}$$

  with $d' < d$. That is, $W_Q W_K^\top$ becomes a low-rank approximation of the $d \times d$ weight matrix. Then we need two matrices instead of one.
- We will see this situation in multi-head attention.

Transformer and Attention IV

- In (6), the SoftMax function is applied on each row z of an input matrix in the following way.

$$\mathrm{SoftMax}(\mathbf{z}) = \begin{bmatrix} \frac{\exp(z_1)}{\sum_j \exp(z_j)} \\ \vdots \\ \frac{\exp(z_T)}{\sum_j \exp(z_j)} \end{bmatrix}. \tag{7}$$

- Let us briefly talk about the attention operation in (6).

Transformer and Attention V

- If the SoftMax$(\cdot)$ part is not there, then (6) reduces to
$$\tilde{Z}W_V,$$
which is no more than a feed-forward operation.
- What
$$\text{SoftMax}\left(\frac{\tilde{Z}W_Q W_K^\top (\tilde{Z})^\top}{\sqrt{d}}\right)$$
give are weights for the T words in $\tilde{Z}$.
- Thus in (6) we do a weighted average of words in $\tilde{Z}$.

Transformer and Attention VI

- The purpose is to transform the representation of each word based on its relationship to other words in the same document.
- We do not get into details here because our focus is not on explaining the attention mechanism.
- In practice, we extend the single-head attention to multi-head for capturing different types of relationships between words in the document.

Transformer and Attention VII

- Specifically, we combine results of h heads:

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_h) W_O, \tag{8}$$

where

$$\text{head}_i = \text{SoftMax}\left(\frac{\tilde{Z} W_Q^i (W_K^i)^\top (\tilde{Z})^\top}{\sqrt{d}}\right) \tilde{Z} W_V^i \in \mathbf{R}^{T \times d/h}. \tag{9}$$

- Note that GPT-2 includes an additional DropOut applied to the output of the SoftMax function.

Transformer and Attention VIII

- That is, the head_i in GPT-2 is defined by

$$\text{DropOut}(\text{SoftMax}(\frac{\tilde{Z}W_Q^i(W_K^i)^\top(\tilde{Z})^\top}{\sqrt{d}}))\tilde{Z}W_V^i$$

  instead of (9).
- Due to the use of h heads, we now have

$$W_Q^i \in \mathbf{R}^{d\times d/h}, W_K^i \in \mathbf{R}^{d\times d/h}, W_V^i \in \mathbf{R}^{d\times d/h}. \quad (10)$$

Transformer and Attention IX

- Earlier we talked about if $W_Q^i(W_K^i)^\top$ can become just one matrix. We cannot do that here because W_Q^i and W_K^i are no longer squared matrices.
- In (8), Concat() is a function which concatenates matrices together. That is,

$$\begin{aligned} & \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \\ = \ & [\text{head}_1, \ldots, \text{head}_h] \in \mathbf{R}^{T \times d}. \end{aligned}$$

- Another advantage of using multiple heads is that the computations for $\text{head}_i, \forall i$ can be done in parallel.

Transformer and Attention X

- We further have in (8) that

$$W_O \in \mathbf{R}^{d \times d}. \tag{11}$$

- The use of W_O is like we have a linear layer after concatenation.

Layer Normalization I

- First let us give the mathematical operations in (17) and (4).
- For any row $\boldsymbol{z}$ of the input matrix, the normalized row is

$$\text{Normalize}(\boldsymbol{z}) = \boldsymbol{a} \odot \frac{\boldsymbol{z} - \text{mean}(\boldsymbol{z})}{\text{std}(\boldsymbol{z})} + \boldsymbol{b}, \tag{12}$$

where mean$(\cdot)$ and std$(\cdot)$ are the mean and standard deviation, and $\odot$ means the component-wise product between two vectors.

Layer Normalization II

- In (12),
$$\boldsymbol{a} \in \mathbf{R}^d, \boldsymbol{b} \in \mathbf{R}^d$$
are learnable parameters shared across rows of the input matrix.
- The reason of applying layer normalization is to avoid too large or too small gradient values.
- In deep learning, we have operations across layers to calculate the gradient.
- Such a long sequence of operations may cause very large or small values (think about the multiplication of several numbers).

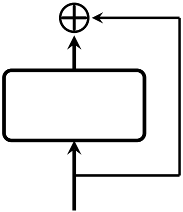
Layer Normalization III

- Several techniques are available to address this issue of too large or too small gradient values, and layer normalization is one of them.
- As we can see, the normalization operation in (12) avoids values being extreme.

Residual Connections I

- Residual connection is another technique to make the problem of too small and too large gradient values less serious.
- It also improves the overall training stability.
- The operation is simply to sum the input and output of one (or several) neural network layer.
- Usually we use the following flowchart to represent residual connections:

Residual Connections II

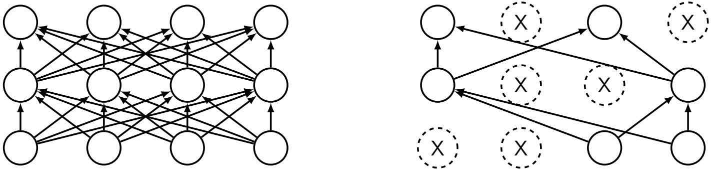


- The residual connection can be applied to any network layer with the same input/output dimensionality.

Dropout Operations I

- Dropout is a widely-applied technique to prevent overfitting in neural networks.
- From Srivastava et al. (2014), "the key idea is to randomly drop units (along with their connections) from the neural network during training."
- Take a feed-forward neural network as an example. In the following figure, we can see that at each layer, we remove some units (neurons).

Dropout Operations II



(a) Feed-Forward Neural Network (b) After applying dropout

- Consequently, we no longer need some corresponding weights.
- Under each mini-batch of the stochastic gradient method, the network “thinned” by dropout is fixed so we can do the sub-gradient calculation.

Dropout Operations III

- Across different mini-batches, the networks are slightly different.
- It is like that we are training an ensemble of several networks.
- To control the dropout operation, we introduce a rate p to denote the probability that a neuron is retained.
- In the prediction stage, for every instance, we go through the network once, so we need a fixed setting.

Dropout Operations IV

- Thus, we do not remove any neurons or their connections. Instead, we multiply every weight of the dropout layer by the rate p.
- In some frameworks (e.g., PyTorch and TensorFlow), p means the rate of elements being removed.
- Thus, in the prediction stage we should multiply every weight of the dropout layer by $1 - p$.

Dropout Operations V

- Interestingly, what PyTorch and TensorFlow do is that at the training stage, they scale output of each dropout layer by a factor $1/(1-p)$. Then in prediction, the dropout layer does not do anything.
- In this case, dropout can be simply removed in prediction.
- For example, the released code of GPT-2 is for inference. So we do not see any dropout operation in the code.

Dropout Operations VI

- While the dropout operation was first proposed for fully-connected layers, we can extend it to other types of architectures.
- For example, in some implementations, dropout is applied in the attention layers.
- Specifically, after the softmax function in (7), some elements in the $\mathbf{R}^{d\times d}$ matrix are not used during training.

Feed-Forward Layers I

- In each transformer block, after multi-head attention, GPT-2 considers two feed-forward layers.
- In (5), the GELU (Gaussian Error Linear Units) activation function is as follows.

$$\text{GELU}(\tilde{z}) = \tilde{z} \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{\tilde{z}}{\sqrt{2}}\right)\right], \tag{13}$$

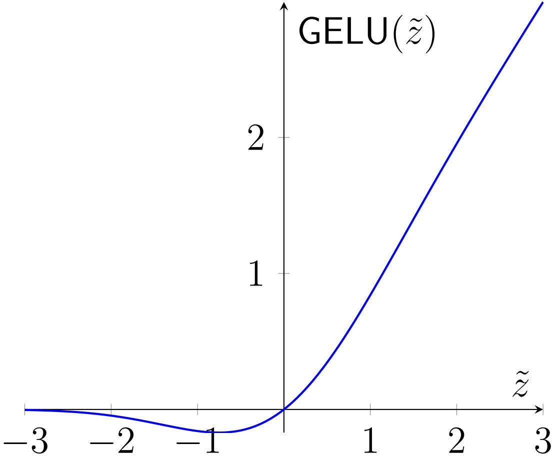
where $\text{erf}(\tilde{z})$ denotes the error function, defined by:

$$\text{erf}(\tilde{z}) = \frac{2}{\sqrt{\pi}} \int_0^{\tilde{z}} e^{-t^2}\, dt.$$

Feed-Forward Layers II

- The GELU function shown below is a smooth version of the ReLU activation function.

Feed-Forward Layers III



Feed-Forward Layers IV

- In GPT-2,

$$W_1 \in \mathbf{R}^{d\times 4d} \text{ and } W_2 \in \mathbf{R}^{4d\times d}. \tag{14}$$

- From (5), we see that to multiply with $\tilde{Z}$ and to have the same output size as the input, the number of rows of W_1 and the number of columns of W_2 must be both d. However, the choice of $4d$ appears to be arbitrary.

Final Linear Output Layer I

- After all transformer blocks, we get an output matrix:
$$Z^{\text{out}} \in \mathbf{R}^{T \times d}.$$
- We must convert each row vector to an index in the Vocabulary set as our next-word prediction.
- To this end, we have a final linear layer with weight matrix
$$W^{\text{final}} \in \mathbf{R}^{d \times |\text{Vocabulary}|}.$$

Final Linear Output Layer II

- Then from

$$Z^{\text{out}} \times W^{\text{final}} \in \mathbf{R}^{T \times |\text{Vocabulary}|}, \tag{15}$$

  for every token in $1, \ldots, T$, we select the index corresponding to the largest of the $|\text{Vocabulary}|$ values as the prediction.
- Interestingly, people use the same word embedding vectors for the input matrices as the weights of the final linear layer.
- Recall we said that all word and position embedding vectors are trainable parameters.

Final Linear Output Layer III

- By this setting, instead of two $|\text{Vocabulary}| \times d$ matrices, we save the space by using only one.

Number of Parameters I

- In large language models, people often show the number of parameters to reflect the model size.
- For example, the total number of the model "GPT2-small" is said to have around 124 millions of parameters.
- We show details to calculate the number of parameters.
- To do so, we check weights in different parts of an LLM model:
    - weights in transformer blocks,
    - weights in the final linear layer, and

Number of Parameters II

- weights in the input matrices.

- In each transformer block, from (10), (11), and (14), we have

$$W_Q^i \in \mathbf{R}^{d\times d/h}, W_K^i \in \mathbf{R}^{d\times d/h}, W_V^i \in \mathbf{R}^{d\times d/h}, i = 1, \ldots,$$

$$W_O \in \mathbf{R}^{d\times d},$$

and

$$W_1 \in \mathbf{R}^{d\times 4d},\ W_2 \in \mathbf{R}^{4d\times d}.$$

Number of Parameters III

Thus, the total number is

$$
\begin{aligned}
& 4 \times d^2 + 4 \times d^2 + 4 \times d^2 \\
= & 12 \times d^2.
\end{aligned}
$$

- For the final linear layer, the number of weights is
$$|\text{Vocabulary}| \times d.$$
- Now let us check the remaining parts.
- The input matrix is the combination of two parts: token embedding and position embedding.

Number of Parameters IV

- Recall we said that all word and position embedding vectors are trainable parameters.
- For token embedding, because according to document contents, we find each token's corresponding embedding, the space needed is

$$|\text{Vocabulary}| \times d.$$

- However, we have mentioned that the same weights are used for the final linear layer, so no extra space is needed.

Number of Parameters V

- For position embedding, because we have T possible positions, the number of weights is

$$T \times d.$$

- For GPT-2-small:
  - Number of attention blocks $=$ 12,
  - $d =$ 768,
  - |Vocabulary| $=$ 50,257,[2]
  - $T =$ 1,024.

Number of Parameters VI

- The sum is

$$
\begin{aligned}
& 12 \times d^2 \times \text{number of blocks} + |\text{Vocabulary}| \times d + \\
& T \times d \\
= \; & 12 \times 768^2 \times 12 + 50{,}257 \times 768 + 1{,}024 \times 768 \\
= \; & 124{,}318{,}464 \approx 124 \text{ Million.}
\end{aligned}
$$

- The GPT-2 paper (Rashed et al., 2019) wrongly stated that the number of parameters is 117 million, though later they stated 124 million in other places.

[2] In some subsequent implementation, |Vocabulary| is increased to 50,304, the nearest multiple of 64 for efficiency.

Outline

1. Network architecture
2. Transform blocks and other details
3. Masked self-attention
4. Prediction

Next-token Prediction I

- Recall that we have a sequence of next-token predictions; see Figure 1.
- However, in the earlier description of the architecture for training, we may not always take this situation into account.
- For example, in our self-attention operation, we calculate the relationship between all words in the word sequence.

Next-token Prediction II

- This situation violates a condition mentioned earlier for training an auto-regressive model: the training decision function should be the same as the prediction decision function.
- In self-attention, we should sequentially do

$$\text{SoftMax}\left(\frac{\begin{bmatrix}\text{“I”}\end{bmatrix} W_Q W_K^\top \begin{bmatrix}\text{“I”}\end{bmatrix}^\top}{\sqrt{d}}\right)_{1\times 1} \begin{bmatrix}\text{“I”}\end{bmatrix}_{1\times d} W_V$$

$$\text{SoftMax}\left(\frac{\begin{bmatrix}\text{“I”}\\ \text{“am”}\end{bmatrix} W_Q W_K^\top \begin{bmatrix}\text{“I”}\\ \text{“am”}\end{bmatrix}^\top}{\sqrt{d}}\right)_{2\times 2} \begin{bmatrix}\text{“I”}\\ \text{“am”}\end{bmatrix}_{2\times d} W_V$$

$$\vdots$$

Next-token Prediction III

- The reason is that to have

  $$\text{input: ``I''}$$

  and

  $$\text{output: ``am,''}$$

  which corresponds to

  $$x_{i,1} \rightarrow f(\boldsymbol{\theta}; \boldsymbol{x}_{i,1:1}) \approx x_{i,2}$$

  in Figure 1, we can only calculate the word relationships of the current input document.

Next-token Prediction IV

- Now "I" is the only word in our document.
- Therefore, the suitable setting is different from what we did earlier.

Masked Self-attention I

- The formulation we gave earlier was

$$\begin{bmatrix} \text{“I”} \\ \vdots \\ \text{“researcher”} \end{bmatrix} W_Q W_K^\top \begin{bmatrix} \text{“I”} \\ \vdots \\ \text{“researcher”} \end{bmatrix}^\top \in \mathbf{R}^{T\times T}.$$

- To be consistent with the prediction, what we should use is only the lower triangular part of the above matrix:

$$\begin{bmatrix} (1,1) & & & \\ (2,1) & (2,2) & & \\ \vdots & \vdots & \ddots & \\ (T,1) & \cdots & \cdots & (T,T) \end{bmatrix}.$$

Masked Self-attention II

- Therefore, in the training procedure, we must mask all entries above the diagonal.
- In practice, people just assign these entries to $-\infty$:

$$\begin{bmatrix} (1,1) & -\infty & \cdots & -\infty \\ (2,1) & (2,2) & -\infty & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ (T,1) & \cdots & \cdots & (T,T) \end{bmatrix}.$$

Masked Self-attention III

- Then in the SoftMax operation, because

$$e^{-\infty} = 0,$$

we have the desired matrix.

Feed-forward Layers I

- An interesting question is if we have the same issue in other operations.
- Let us briefly check the feed-forward layers.
- Recall in (5) we calculate

$$\text{GELU}(\tilde{Z}W_1)W_2. \tag{16}$$

Feed-forward Layers II

- To have the same setting as prediction, we should sequentially do

$$
\begin{aligned}
&\text{GELU}(\begin{bmatrix} \text{“I”} \end{bmatrix}_{1\times d}(W_1)_{d\times 4d})W_2 \\
&\text{GELU}(\begin{bmatrix} \text{“I”} \\ \text{“am”} \end{bmatrix}_{2\times d}(W_1)_{d\times 4d})W_2 \\
&\vdots
\end{aligned}
$$

- The operations are precisely the same as those in (16), so we are fine.

Feed-forward Layers III

- Up to this point, we see that operations like (16) lead us to have a desired property for training an auto-regressive model: we assemble all the next-token predictions together in the training process.
- This makes efficient matrix computation for fast training.

Outline

1 Network architecture

2 Transform blocks and other details

3 Masked self-attention

4 Prediction

Naive Prediction Step by Step I

- In (15), we wrote that the final output of the network is

$$Z^{\text{out}} \times W^{\text{final}} \in \mathbf{R}^{T \times |\text{Vocabulary}|}.$$

  Then for every token in $1, \ldots, T$, we select the index corresponding to the largest of the $|$Vocabulary$|$ values as the prediction.
- However, this setting is for training. For our given sequence,

$$z_1, z_2, \ldots .$$

  we have the following training and target pairs

Naive Prediction Step by Step II

training instances	target values
z_1	z_2
z_1, z_2	z_3
$\vdots$	$\vdots$

We consider all of them together, like

training instances	target values
$z_1, \cdots, z_{T-1}$	$z_2, \cdots, z_T$

- For prediction, we can only consider one pair at a time (e.g., first (z_1, z_2) and then $((z_1, z_2), z_3)$), as z_2 is not available in advance and thus cannot be used as input to the model until it has been generated.

Naive Prediction Step by Step III

- Generally speaking,

$$z_1, \ldots, z_T \rightarrow \text{ an estimate of } z_{T+1}$$

- Therefore, if

$$Z^{\text{out}} = \begin{bmatrix} \left(\boldsymbol{z}_1^{\text{out}}\right)^\top \\ \vdots \\ \left(\boldsymbol{z}_T^{\text{out}}\right)^\top \end{bmatrix},$$

all we need is

$$\left(\boldsymbol{z}_T^{\text{out}}\right)^\top \times W^{\text{final}} \in \mathbf{R}^{1 \times |\text{Vocabulary}|}.$$

Naive Prediction Step by Step IV

- To get $\boldsymbol{z}_T^{\text{out}}$, let us check the needed operations.
- Consider the input

$$Z = \begin{bmatrix} \boldsymbol{z}_1^\top \\ \vdots \\ \boldsymbol{z}_T^\top \end{bmatrix}.$$

Naive Prediction Step by Step V

Eqs. (17)-(5) become

$$
\begin{aligned}
\tilde{\boldsymbol{z}}_T^\top &= \text{LayerNorm}(\boldsymbol{z}_T^\top) \\
\boldsymbol{z}_T^\top &\leftarrow \boldsymbol{z}_T^\top + \text{DropOut}(\text{MultiHead}(\tilde{\boldsymbol{z}}_T^\top)) \\
\tilde{\boldsymbol{z}}_T^\top &= \text{LayerNorm}(\boldsymbol{z}_T^\top) \\
\boldsymbol{z}_T^{\text{out}} &= \boldsymbol{z}_T^\top + \text{DropOut}(\text{GELU}(\tilde{\boldsymbol{z}}_T^\top W_1) W_2)
\end{aligned}
$$

- Note that LayerNorm is an operation on a row vector
- We clearly see that in the feed-forward layers, all we need is $\boldsymbol{z}_T$ instead of $\boldsymbol{z}_1, \ldots \boldsymbol{z}_{T-1}$

Naive Prediction Step by Step VI

- However, in the attention operation, (6) becomes

$$\text{SoftMax}\left(\frac{\tilde{\boldsymbol{z}}_T^\top W_Q W_K^\top (\tilde{Z})^\top}{\sqrt{d}}\right)\tilde{Z} W_V \in \mathbf{R}^{1\times d}. \quad (17)$$

- Therefore, the whole Z (and $\tilde{Z}$) is needed for the operation, indicating that the model still needs to process T tokens.
- Fortunately, the dependency on the full matrices Z and $\tilde{Z}$ can actually be handled through caching

Naive Prediction Step by Step VII

- Therefore, most computation cost related to the previous $T-1$ tokens can be saved, leading to much faster prediction.
- This technique, commonly referred to as **KV caching**, has been widely adopted in practice and will be detailedly explained in the following slides.

Efficient Prediction with KV Caching I

- Recalling the attention operation in (17), we have

$$W_K^\top (\tilde{Z})^\top = W_K^\top \left[\tilde{\boldsymbol{z}}_1 \cdots \tilde{\boldsymbol{z}}_T \right]$$

and

$$\tilde{Z} W_V = (W_V^\top \tilde{Z}^\top)^\top = \left(W_V^\top \left[\tilde{\boldsymbol{z}}_1 \cdots \tilde{\boldsymbol{z}}_T \right] \right)^\top .$$

- In these two terms,

$$W_K^\top \left[\tilde{\boldsymbol{z}}_1 \cdots \tilde{\boldsymbol{z}}_{T-1} \right] \text{ and } W_V^\top \left[\tilde{\boldsymbol{z}}_1 \cdots \tilde{\boldsymbol{z}}_{T-1} \right]$$

have already been calculated.

Efficient Prediction with KV Caching II

- Here we define

$$\boldsymbol{k}_i = W_K^\top \tilde{\boldsymbol{z}}_i \text{ and } \boldsymbol{v}_i = W_V^\top \tilde{\boldsymbol{z}}_i,$$

  where $\boldsymbol{k}_i$ and $\boldsymbol{v}_i$ are the **key** and **value** vectors for the ith token.
- Then, the attention operation

$$\mathrm{SoftMax}\left(\frac{\tilde{\boldsymbol{z}}_T^\top W_Q W_K^\top (\tilde{Z})^\top}{\sqrt{d}}\right)\tilde{Z} W_V \in \mathbf{R}^{1\times d}.$$

Efficient Prediction with KV Caching III

becomes

$$\text{SoftMax}(\frac{\tilde{\boldsymbol{z}}_T^\top W_Q[\boldsymbol{k}_1 \cdots \boldsymbol{k}_T]}{\sqrt{d}})[\boldsymbol{v}_1 \cdots \boldsymbol{v}_T]^\top.$$

- As mentioned above,

$$\boldsymbol{k}_1 \cdots \boldsymbol{k}_{T-1} \text{ and } \boldsymbol{v}_1 \cdots \boldsymbol{v}_{T-1}$$

have already been calculated during the previous $T-1$ steps.

Efficient Prediction with KV Caching IV

- If we cache all these $T-1$ key and value vectors and reuse them in the calculation of

$$\text{SoftMax}\left(\frac{\tilde{\boldsymbol{z}}_T^\top W_Q \left[\boldsymbol{k}_1 \cdots \boldsymbol{k}_{T-1}\ \boldsymbol{k}_T\right]}{\sqrt{d}}\right) \left[\boldsymbol{v}_1 \cdots \boldsymbol{v}_{T-1}\ \boldsymbol{v}_T\right]^\top,$$

  then this operation will only depend on $\tilde{\boldsymbol{z}}_T$.
- The caching of key and value vectors is the **KV caching** mentioned above.
- In the case with KV caching, the model only needs to take the Tth token to generate the $T+1$ token, like

Efficient Prediction with KV Caching V

Input token	Next token	Cached vectors
z_1	z_2	$[]$
z_2	z_3	$[\boldsymbol{k}_1, \boldsymbol{v}_1]$
z_3	z_4	$[\boldsymbol{k}_1, \boldsymbol{v}_1, \boldsymbol{k}_2, \boldsymbol{v}_2]$
$\vdots$	$\vdots$	$\vdots$

- An implementation of KV caching can be found at `https://github.com/openai/gpt-2`.

Initialization I

- Having discussed in detail how LLMs perform autoregressive prediction in practice, we now turn to how LLMs initialize and terminate token generation.
- Recall the example of autoregressive prediction,

Pre-context		The next token
I	$\rightarrow$	am
I am	$\rightarrow$	a
I am a	$\rightarrow$	machine
I am a machine	$\rightarrow$	learning
I am a machine learning	$\rightarrow$	researcher

Initialization II

- We initialize the generation by conditioning on the pre-context "I".
- In practice, this way to initialize the token generation is referred to as **Conditional generation**.
- In this way, people provide a pre-context (e.g., a question or a command), which is generally called a **prompt**, and then the LLM generates a new token sequence as an answer conditioned on this prompt.
- For example, given the prompt "Who are you?", GPT-2 may generate the tokens like

Initialization III

Pre-context		The next token
Who are you?	$\rightarrow$	I
Who are you? I	$\rightarrow$	am
Who are you? I am	$\rightarrow$	GPT-2
Who are you? I am GPT-2	$\rightarrow$	.

- Depending on the specific LLM, special tokens may need to be added to the prompt.
- Unlike GPT-2 that does not introduce any special tokens, another LLM called Llama adds the special token ‘<s>’, the beginning-of-sentence (BOS) token, to the beginning of the prompt.

Initialization IV

- Then, the example above becomes

Pre-context		The next token
<s> Who are you?	$\rightarrow$	I
<s> Who are you? I	$\rightarrow$	am
<s> Who are you? I am	$\rightarrow$	Llama
<s> Who are you? I am Llama	$\rightarrow$	.

- Whether or not special tokens are added depends on whether these tokens were included in the training (instance, label) pairs of the LLM you are using.

Termination I

- After initializing the token generation, we must decide how to terminate.
- Generally, there are two ways:
    - By setting a **maximum sequence length**;
    - By using a **special token**, like ‘$< \backslash s>$‘, the end-of-sentence (EOS) token.
- In the official implementation of GPT-2, the maximum sequence length is used to stop the generation.

Termination II

- As a result, GPT-2 might stop generating text midway through a sentence, so post-processing is required to handle these incomplete sentences.
- In contrast, in the official implementation of Llama, the generation is ended when the EOS token is detected.
- In fact, the training of GPT-2 makes use of the special token '$< |\text{endoftext}| >$'. Consequently, text generation in GPT-2 can be terminated by detecting this token.

References I

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners, 2019.

A. Rashed, J. Grabocka, and L. Schmidt-Thieme. Multi-label network classification via weighted personalized factorizations, 2019.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008, 2017.