Outline

Outline

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

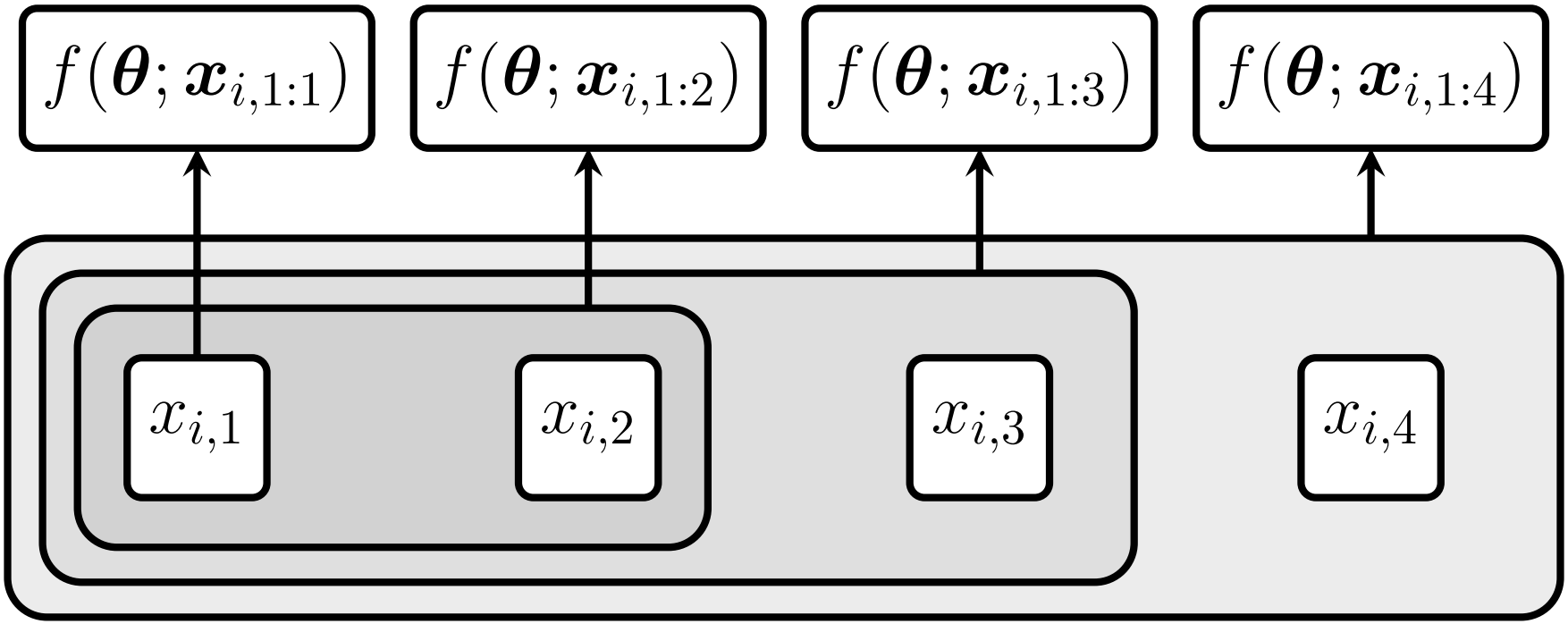


Figure 1: A sequence of next-token predictions

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`.

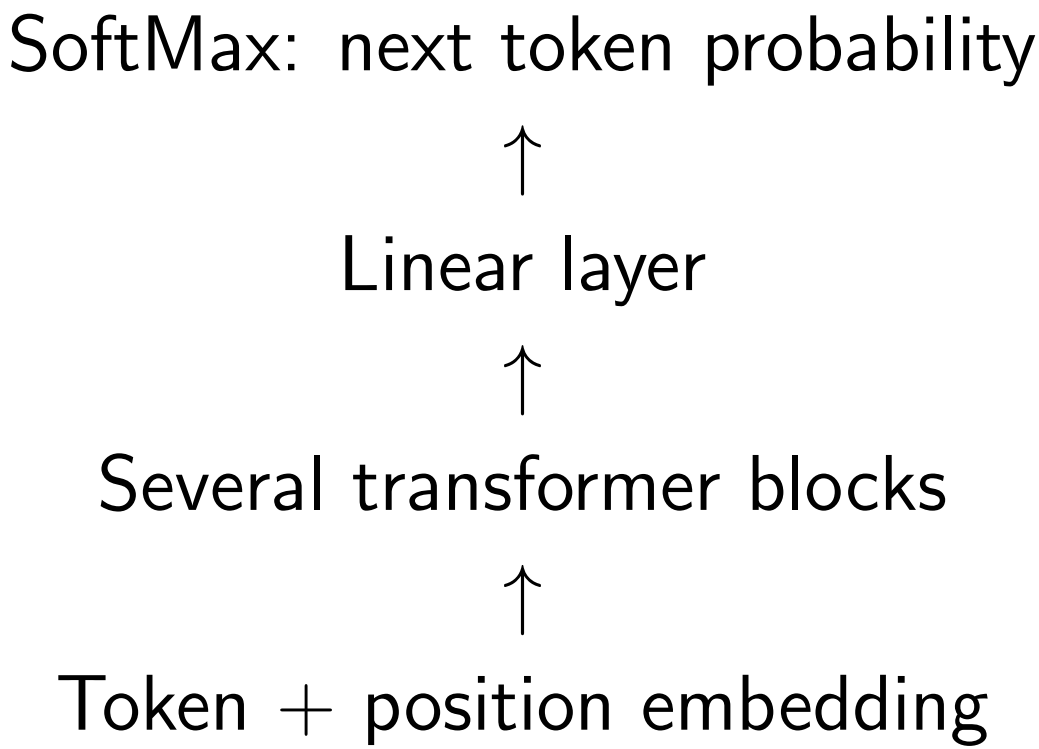
Network Design IV

- 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.

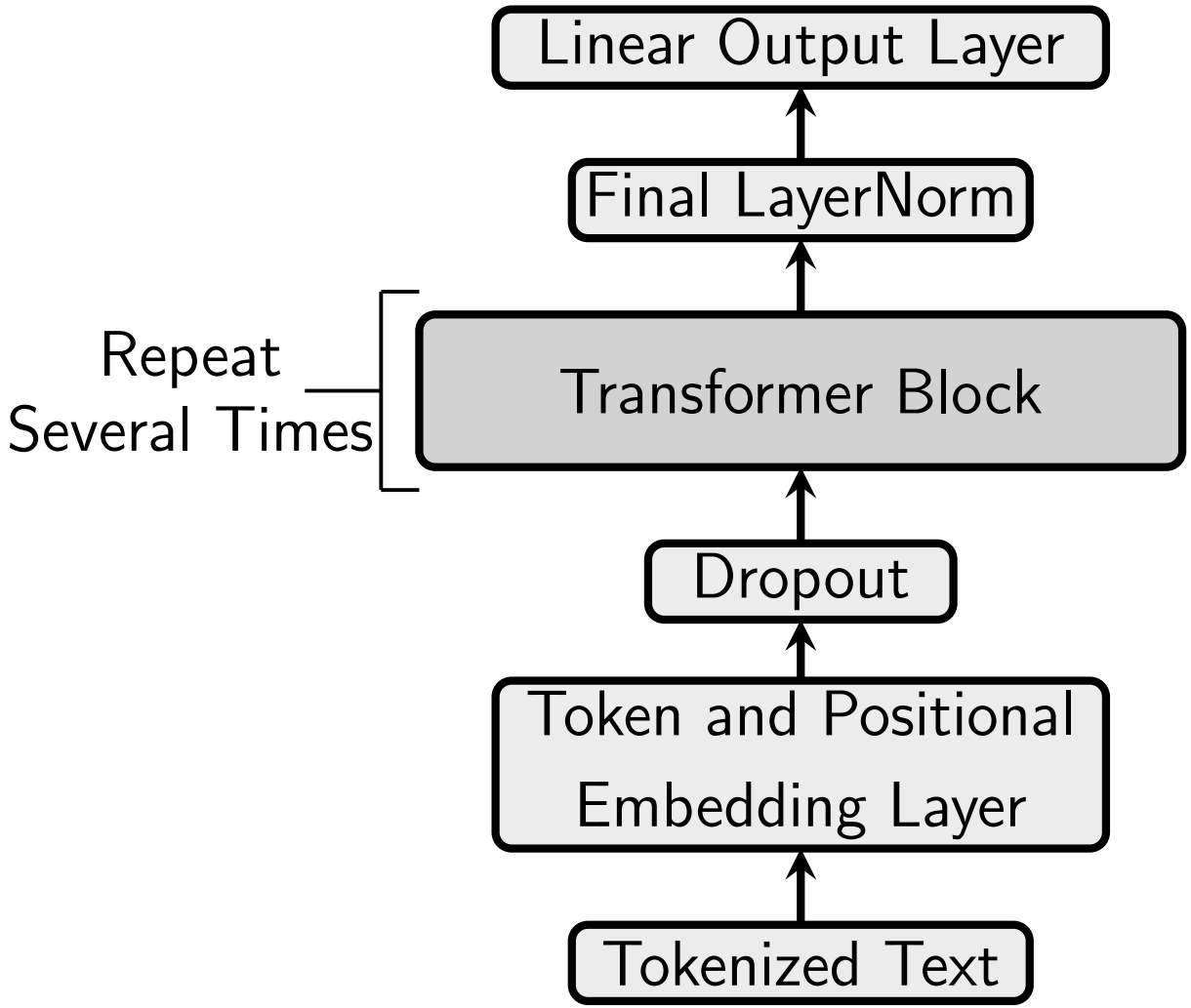
Overall Architecture II



Overall Architecture III

- We specifically discuss the architecture used in the implementation in GPT-2-small (**?**). 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 $1, \ldots, T$ 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.

$$\begin{matrix} x_1 \\ \vdots \\ x_T \end{matrix} \begin{bmatrix} \text{———} \\ \vdots \\ \text{———} \end{bmatrix} \in R^{T \times d} \tag{1}$$

  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

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

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

  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.

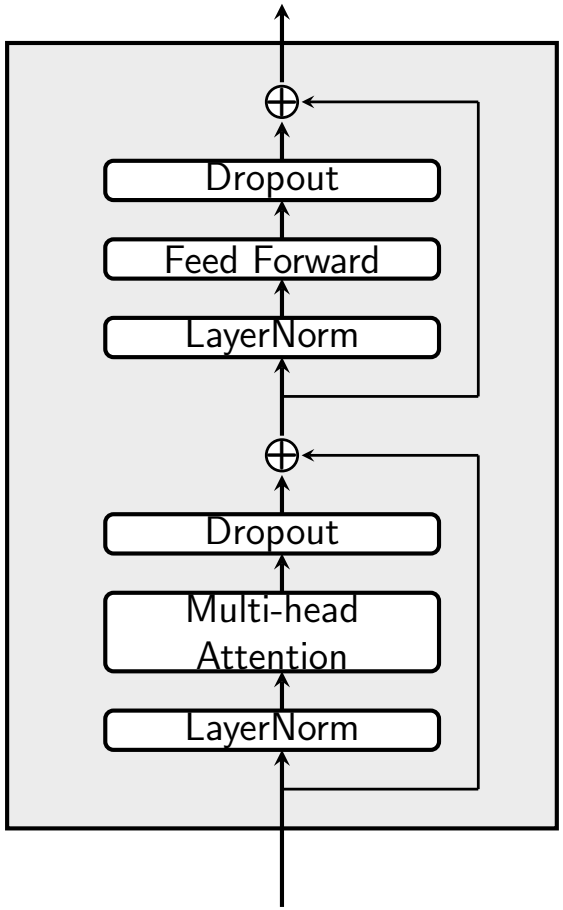
$$\begin{aligned}
\tilde{Z} &= \text{LayerNorm}(Z) && (2)\\
\tilde{Z} &\leftarrow Z + \text{DropOut}(\text{MultiHead}(\tilde{Z})) && (3)\\
\tilde{Z} &\leftarrow \text{LayerNorm}(\tilde{Z}) && (4)\\
Z^{\text{out}} &= \tilde{Z} + \text{DropOut}(\text{GELU}(\tilde{Z}W_1)W_2) && (5)
\end{aligned}$$

- Subsequently we discuss each operation in detail.

A Transformer Block IV

- Besides the attention layer in (3), what else in one transform block may slightly vary across LLM implementations.
- Our operations in (2)-(5) can be illustrated in the following figure.[1]

A Transformer Block V



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

A Transformer Block VI

[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

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

the attention operation is

$$\text{SoftMax}(\frac{\tilde{Z}W_Q W_K^T (\tilde{Z})^T}{\sqrt{d}})\tilde{Z}W_V. \quad (6)$$

Transformer and Attention II

- We consider three trainable weight matrices

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

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

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

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

Transformer and Attention III

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

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

  with $d' < d$. That is, $W_Q W_K^T$ 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 $\boldsymbol{z}$ of an input matrix in the following way.

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

- 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

$$\text{SoftMax}(\frac{\tilde{Z}W_Q W_K^T (\tilde{Z})^T}{\sqrt{d}})$$

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

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

- Due to the use of h heads, we now have

$$W_Q^i \in R^{d\times d/h}, W_K^i \in R^{d\times d/h}, W_V^i \in R^{d\times d/h}. \tag{10}$$

Transformer and Attention VIII

- Earlier we talked about if $W_Q^i(W_K^i)^T$ 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,

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

- Another advantage of using multiple heads is that the computations for head_i, $\forall i$ can be done in parallel.

Transformer and Attention IX

- We further have in (8) that

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

- 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 (2) and (4).
- For any row $\boldsymbol{z}$ of the input matrix, the normalized row is

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

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

Layer Normalization II

- In (12),

$$\boldsymbol{a} \in R^d, \boldsymbol{b} \in R^d$$

  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).

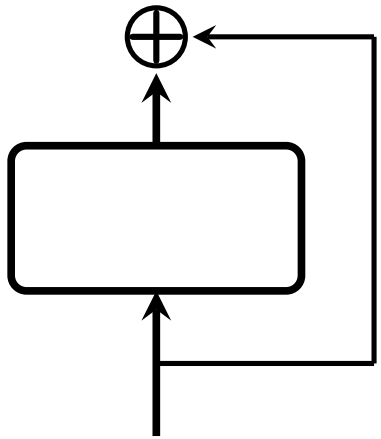
Layer Normalization III

- 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 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."
- Under each mini-batch of the stochastic gradient method, the "thinned" network is fixed so we can do the sub-gradient calculation.
- Thus, in different batches, the networks are slightly different.

Dropout Operations II

- It is like that we are doing an ensemble of several networks.
- Therefore, we need a rate p as the probability that a neuron is retained.
- In the prediction stage, we do not remove any neurons or their connections.
- Instead, we multiply every weight of the dropout layer by the rate p.
- In some places (e.g., PyTorch), p means the rate of elements being removed.

Dropout Operations III

- Thus, in the prediction stage we should multiply every weight of the dropout layer by $1 - p$. Interestingly, what PyTorch does is to scale the training output by a factor $1/(1 - p)$, so in prediction, the dropout layer does not do anything.[2]
- Dropout has been used in different types of architectures. For example, in some implementations such as Nano GPT, a attention dropout is applied.
- Specifically, after the softmax function in (7), some elements in the $R^{d \times d}$ matrix are not used.

[2] This seems to be what GPT-2 has done. They release only the prediction code, in which we do not see any dropout operation.

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.

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

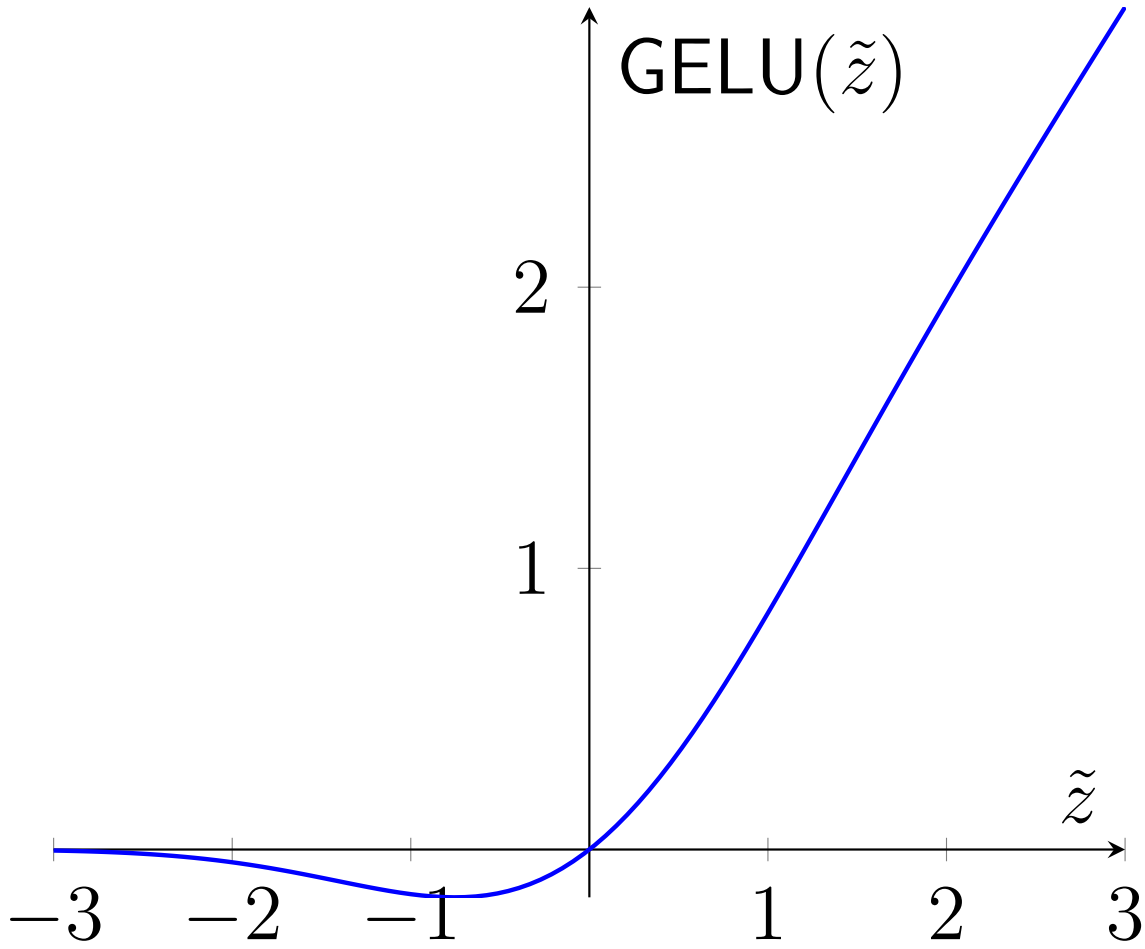
where $\text{erf}(\tilde{z})$ denotes the error function, defined by:

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

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,

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

- 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:
$$Z^{\text{out}} \in \mathbb{R}^{T \times d}.$$
- 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
$$W^{\text{final}} \in \mathbb{R}^{d \times |\text{Vocabulary}|}.$$

Final Linear Output Layer II

- Then from

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

  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

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

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

and

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

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 $= 12$,
  - $d = 768$,
  - $|\text{Vocabulary}| = 50{,}257$,[3]
  - $T = 1{,}024$.

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.

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

Outline

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

$$\text{SoftMax}(\frac{\text{“I”}\, W_Q W_K^T\, \text{“I”}^T}{\sqrt{d}})_{1\times 1}\, \text{“I”}_{1\times d}\, W_V$$

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

$$\vdots$$

Next-token Prediction III

- The reason is that to have

  input: "I"

  and

  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

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

- 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

$$\text{GELU}(\tilde{Z}W_1)W_2. \tag{15}$$

Feed-forward Layers II

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

$$\text{GELU}(\left[\text{“I”}\right]_{1\times d}(W_1)_{d\times 4d})W_2$$

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

$$\vdots$$

- The operations are precisely the same as those in (15), so we are fine.

Feed-forward Layers III

- Up to this point, we see that operations like (15) 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.

References I

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, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. 2017.