Transformers and Large Language Models

- **Transformers – Self Attention Networks**
- **Large Language Models**

Transformers – Self Attention Networks

Transformer – A Self Attention Network

Pretraining: ***learning knowledge about language and world from vast amounts of text.***

- The resulting ***pretrained language models*** are called as **large language models**.

- The **transformer** is a standard architecture for building ***large language models***.
- The transformer makes use of a novel mechanism called **self-attention**, which developed out of the idea of attention that was introduced for RNNs.

- Generally, the input to a transformer is a sequence of words, and the output is a ***prediction for what word comes next***, as well as a ***sequence of contextual embedding*** that represents the ***contextual meaning of each of the input words***.

Transformers vs. LSTMs

- Like the LSTMs, transformers **can handle distant information**
- But unlike LSTMs, transformers are **not based on recurrent connections**
  - Transformers can be more efficient to implement at scale. The training can be parallelized.

- Transformers are made up of stacks of **transformer blocks**,
- Each **transformer block** which is a multilayer network maps sequences of input vectors ($x_1,\dots,x_n$) to output vectors ($x_1,\dots,x_n$) of the same length.
- These **transformer blocks** are made by combining:
  - simple linear layers
  - feedforward networks
  - **self-attention layers**

- **Self-attention** allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs

Transformers - intuition

Intuition of a transformer: Using a series of layers, a transformer can build up richer and richer ***contextualized representations of the meanings of input words*** or tokens.

  – Although ***input*** is referred as ***a sequence of words*** for convenience, technically ***input*** is first tokenized by an algorithm like BPE, so it is ***a series of tokens*** rather than words.

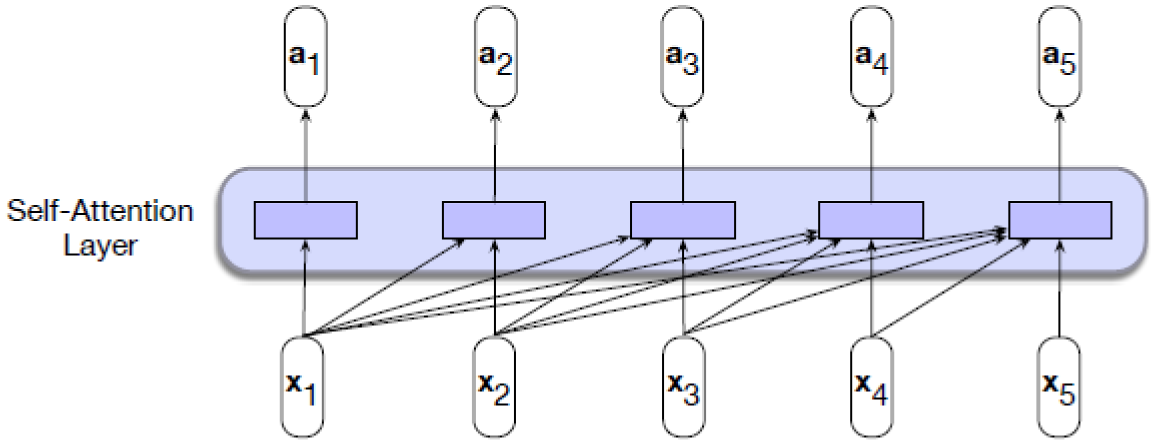
- At each layer of a transformer, to compute the representation of a word i:
  - Its representation at the previous layer is combined with information from the representations of the neighboring words.
  - The goal is to produce ***a contextualized representation for each word at each position.***

- ***Transformers produce contextualized versions of word vectors***.
- ***A contextualized representation*** represents what this word means in the particular context in which it occurs.

Self-Attention

backward-looking (causal) self-attention

- The **concept of context** can be used in two ways in **self-attention**.
  - In **backward looking (casual) self-attention**, the context is any of the prior words.
  - In **bidirectional self-attention**, the context can include future words (like in BERT).
- When processing each item in the input (in ***backward looking self-attention***):
  - The model has access to all the inputs up to and including the one under consideration
  - No access to information about inputs beyond the current one (*only left context*)
  - The computation performed for each item is *independent* of all the other computations
    - Easily parallelize both forward inference and training of such models

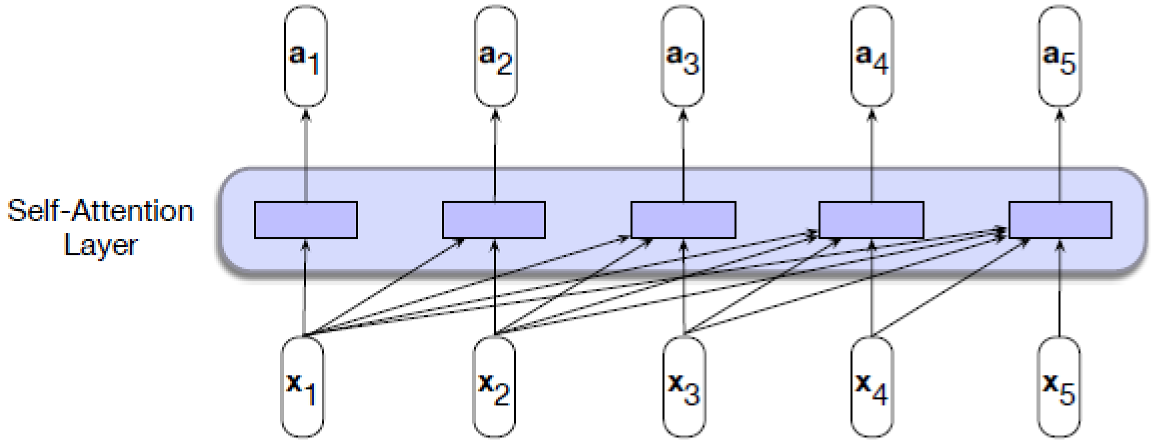


Self-Attention

Attention-based approach: *Compare an item to a collection of other items in a way that reveals their **relevance in the current context**.*

Self-Attention: *The set of **comparisons** is made to other elements within a given sequence*;

- the result of these comparisons is then used to compute an output for the current input



Self-Attention

Simple Variant

- **Computation of the output value a_i** is based on **a set of comparisons** (**dot product**) between the input x_i and its preceding elements x_1 and x_2, and to x_i itself.
  1. The first step is to compute a similarity between x_i and x_j : **score(x_i,x_j) = $x_i . x_j$**
  2. Normalize them with a ***softmax*** to create a **vector of weights**, $\boldsymbol{\alpha_{ij}}$
  3. Generate a_i

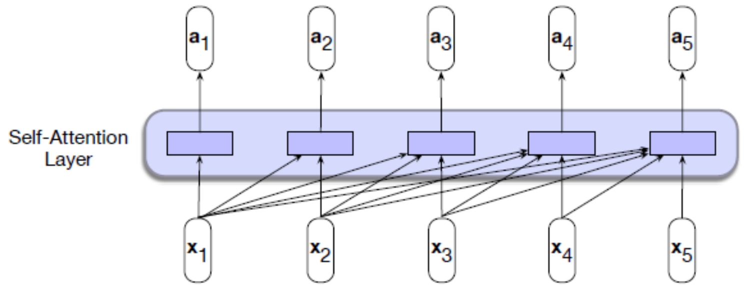
$$
\begin{aligned}
\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{x}_i,\mathbf{x}_j)) \;\; \forall j \le i \\
&= \frac{\exp(\text{score}(\mathbf{x}_i,\mathbf{x}_j))}{\sum_{k=1}^{i}\exp(\text{score}(\mathbf{x}_i,\mathbf{x}_k))} \;\; \forall j \le i
\end{aligned}
$$

α_{ij} **is a proportional relevance** of each input to input i (current focus of attention).

- weight will likely be highest for the current focus element i, since x_i is very similar to itself.
- Other context words may also be similar to i, and softmax also assign some weight to them.

$$\mathbf{a}_i = \sum_{j \le i} \alpha_{ij}\mathbf{x}_j$$

Attention Value



Self-Attention

- Transformers allow us to create a more sophisticated way of representing how words can contribute to the representation of longer inputs.
- Each input embedding plays ***three different roles during attention process*** :
  - **Query** : The **current focus** of attention when being compared to all the other preceding inputs.
  - **Key** : In its role as **a preceding input** being compared to the current focus of attention.
  - **Value** : As used to compute the **output** for the current focus of attention.

- To capture these three different roles, transformers introduce **weight matrices W^Q, W^K,** and **W^V.**
- These weights project each input vector xi into a representation of its role as a **query**, **key**, or **value**.

$$q_i = x_i W^Q \qquad k_i = x_i W^K \qquad v_i = x_i W^V$$

Self-Attention Layer

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}} \qquad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{K}} \qquad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{V}}$$

- The inputs x and outputs y of transformers, as well as the intermediate vectors after the various layers, all have the same dimensionality 1×d
- Let's assume the dimensionalities of the transform matrices are $W^Q \in R^{d \times dk}$, $W^K \in R^{d \times dk}$, and $W^V \in R^{d \times d}$
  - The dimension of W^V can be $R^{d \times dv}$. In this case, we may need projection from dv to d dimension.
- Both query q_i and key k_i are of dimensionality of $1 \times d_k$.
- Given these projections, the score between a **current focus of attention** x_i and x_j (preceding context) consists of a dot product between its query vector q_i and the preceding element's key vectors k_j.

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j$$

- The result of a dot product can be an arbitrarily large divide the dot product by the square root of the dimensionality of the query and key vectors

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

Self-Attention Layer

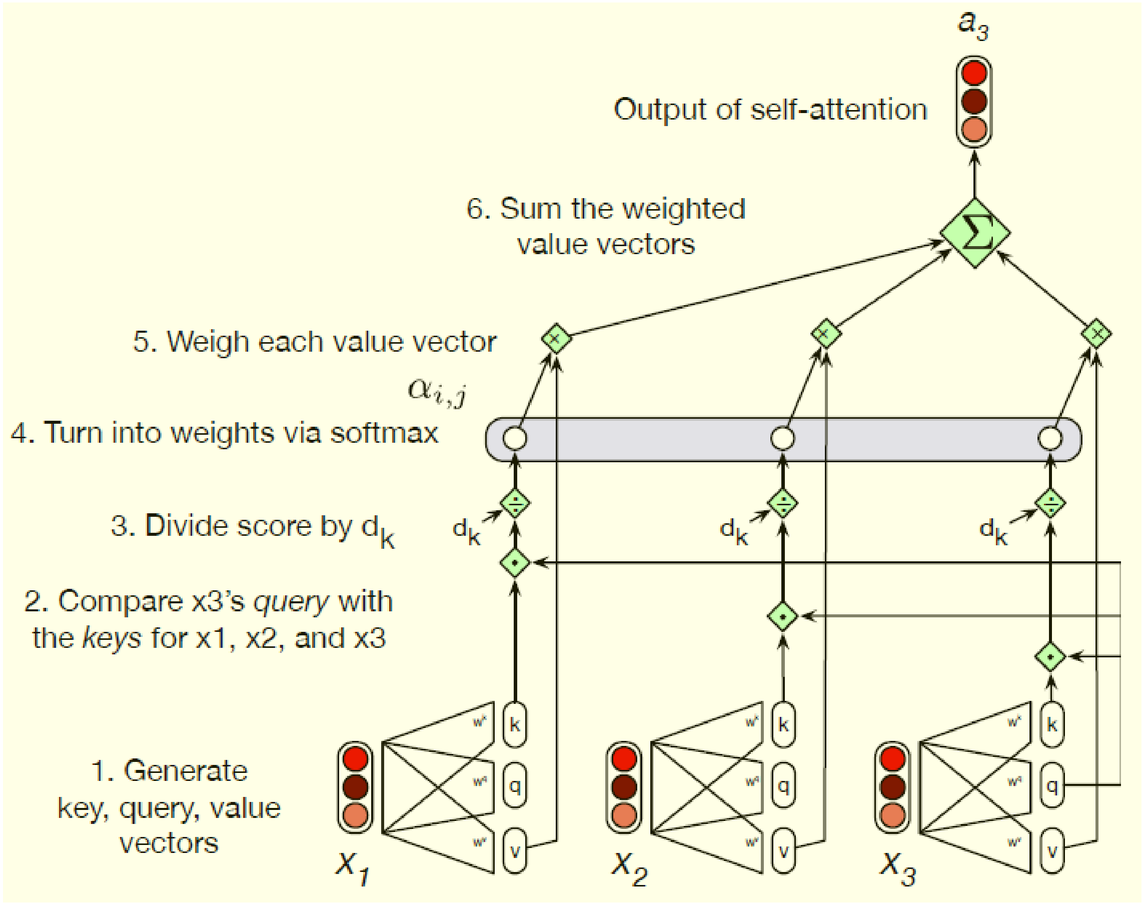
$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}};\ \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{K}};\ \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{V}}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j))\ \forall j \leq i$$

$$\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j$$

Calculating the value of a$_3$, the third element of a sequence using causal (left-to-right) self-attention.



a_3
Output of self-attention
6. Sum the weighted value vectors
5. Weigh each value vector
$\alpha_{i,j}$
4. Turn into weights via softmax
3. Divide score by d_k
d_k
2. Compare x3's *query* with the *keys* for x1, x2, and x3
1. Generate key, query, value vectors
k
q
v
x_1
x_2
x_3

Self-Attention Layer

- The input of a transformer is a sequence of embeddings of N tokens : $\mathbf{X} \in \mathbf{R}^{\mathbf{N \times d}}$
  - Transformers for large language models can have an input length N=1024,2048 or 4096.
- We can multiply X by the key, query, and value matrices to produce matrices **Q**, **K** and **V** containing all the query, key, and value vectors:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \;\; \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \;\; \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

- We can reduce the entire self-attention step for an entire sequence of N tokens to the following computation

$$\mathrm{SelfAttention}(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathsf{T}}}{\sqrt{d_k}}\right)\mathbf{V}$$

Multihead Attention

- It would be difficult for a ***single self-attention model*** to learn *to capture all of the different kinds of parallel relations among its inputs.*
- Transformers address this issue with **multihead self-attention layers.**

- Sets of self-attention layers, called **heads**, that reside in parallel layers at the same depth in a model, each with its ***own set of parameters***
- Given these distinct sets of parameters, each head can learn different aspects of the relationships that exist among inputs at the same level of abstraction.

- To implement this notion, each head i, in a self-attention layer is provided with its own set of key, query and value matrices: $\mathbf{W_i^K}$, $\mathbf{W_i^Q}$ and $\mathbf{W_i^V}$.

Multihead Attention

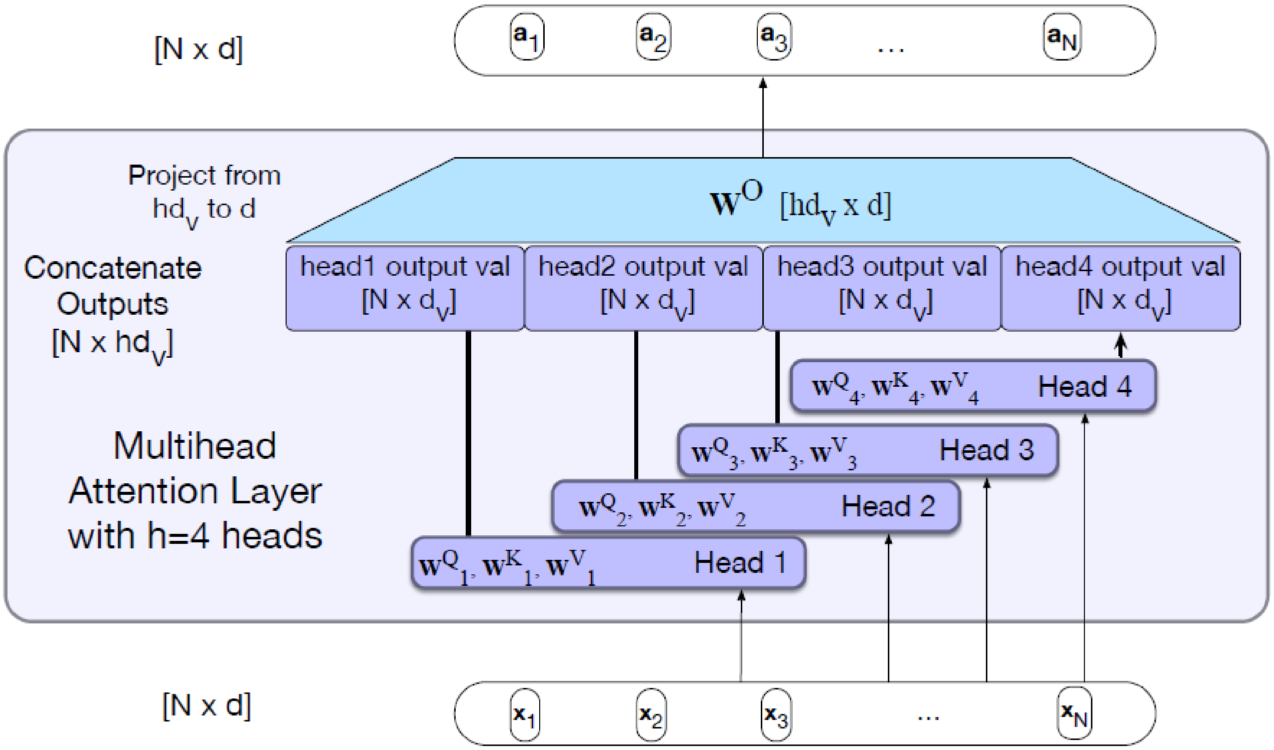
- In multi-head attention, ***model dimension*** is still **d**, ***key*** and ***query*** embeddings have dimensionality $\mathbf{d_k}$, and ***value*** embeddings are of dimensionality $\mathbf{d_v}$
  - in the original transformer paper dk = dv = 64, h = 8, and d = 512.
- Thus for each head i, we have weight layers:

$$\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k} \qquad \mathbf{W}_i^K \in \mathbb{R}^{d \times d_k} \qquad \mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$$

- These weights are multiplied by N input embeddings packed into $\mathbf{X \in R^{N \times d}}$ to produce **Q**, **K** and **V** matrices: $\mathbf{Q \in R^{N \times dk}}$, $\mathbf{K \in R^{N \times dk}}$ and $\mathbf{V \in R^{N \times dv}}$
- The output of the multi-head layer with h heads consists of h matrices of shape $\mathbf{N \times d_v}$.
- Another linear projection $\mathbf{W^O \in R^{h.dv \times d}}$, that reshape it to the original output dimension for each token.
- Multiplying the concatenated matrix output by $\mathbf{W^O}$ yields the self-attention output of shape N×d, suitable to be passed through layers.

Multihead Attention

- Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices.
- The outputs from each of the layers are concatenated and then projected to d, thus producing an output of the same size as the input so the attention can be followed by layer norm and feedforward and layers can be stacked.


[N x d]
a_1 a_2 a_3 … a_N
Project from hd_V to d
W^O $[hd_V \times d]$
Concatenate Outputs $[N \times hd_V]$
head1 output val $[N \times d_V]$
head2 output val $[N \times d_V]$
head3 output val $[N \times d_V]$
head4 output val $[N \times d_V]$
w^Q_4, w^K_4, w^V_4 Head 4
w^Q_3, w^K_3, w^V_3 Head 3
w^Q_2, w^K_2, w^V_2 Head 2
w^Q_1, w^K_1, w^V_1 Head 1
Multihead Attention Layer with h=4 heads
[N x d]
x_1 x_2 x_3 … x_N

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q \; ; \; \mathbf{K} = \mathbf{X}\mathbf{W}_i^K \; ; \; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V$$

$$\mathbf{head}_i = \mathrm{SelfAttention}(\mathbf{Q},\mathbf{K},\mathbf{V})$$

$$\mathrm{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W}^O$$

Transformer Blocks

- ***Self-attention (multihead attention) layer*** is at the core of a **transformer block.**
- In addition to the **self-attention layer**, a transformer block also includes a **feedforward layer**, **residual connections** and **normalizing layers** (***layer norm***).

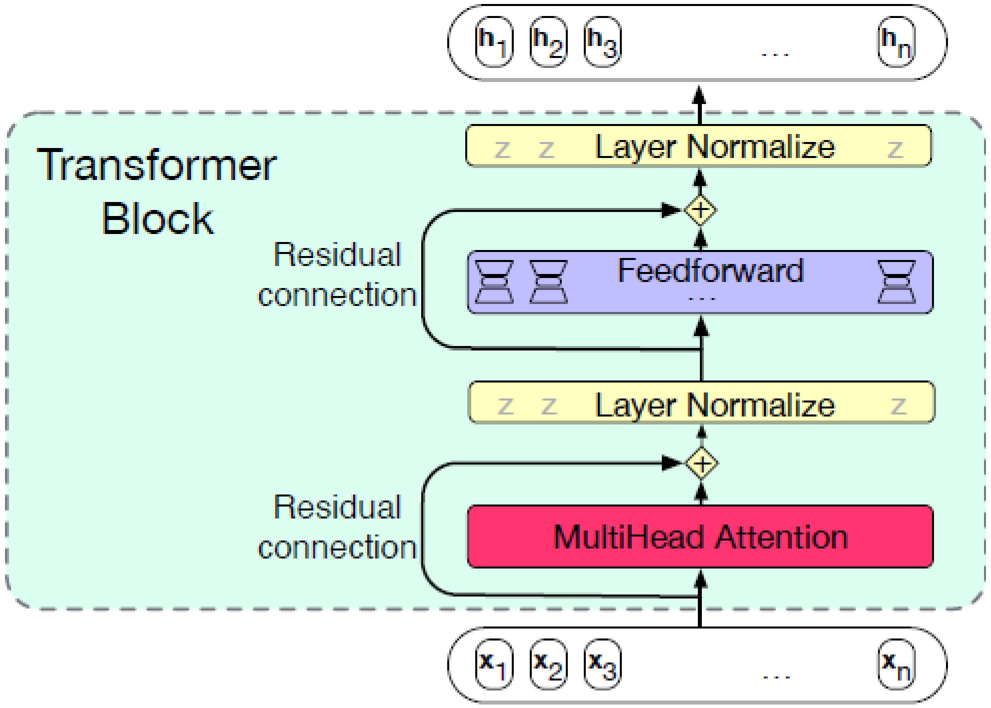
Feedforward layer contains ***N position-wise networks***, one at each position.

- Each is a fully-connected 2-layer network.

Residual connections are connections that pass information from a lower layer to a higher layer without going through the intermediate layer;

- gives higher level layers direct access to information from lower layers

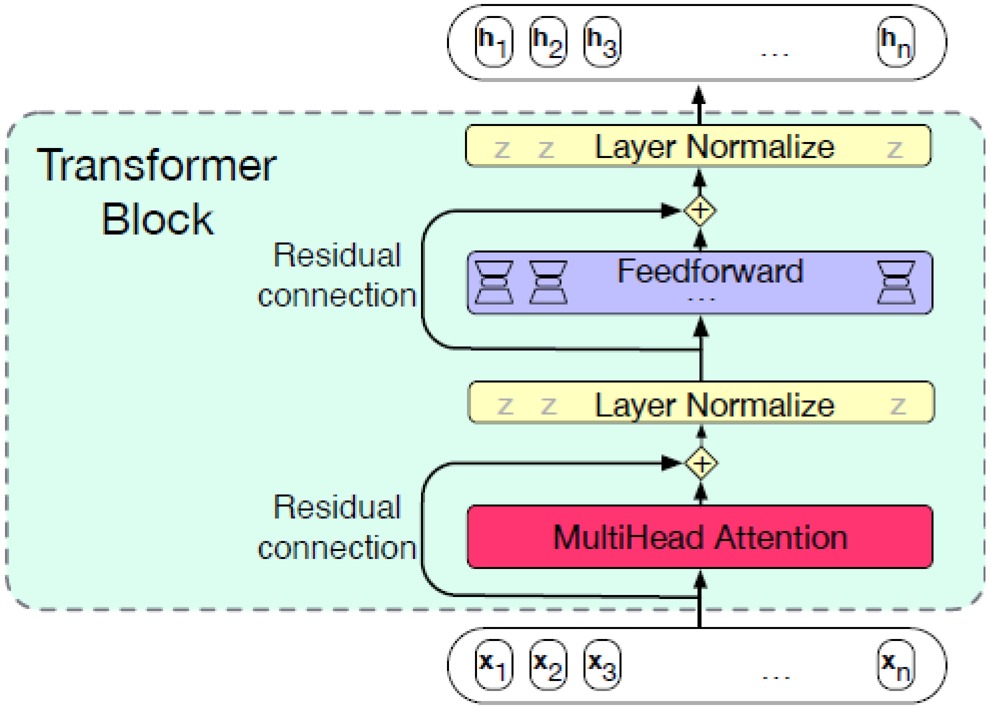
Layer Norm: Forms of normalization that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training



Transformer Blocks

The function computed by a transformer block can be expressed as:

$$\mathbf{O} = \text{LayerNorm}(\mathbf{X} + \text{SelfAttention}(\mathbf{X}))$$
$$\mathbf{H} = \text{LayerNorm}(\mathbf{O} + \text{FFN}(\mathbf{O}))$$

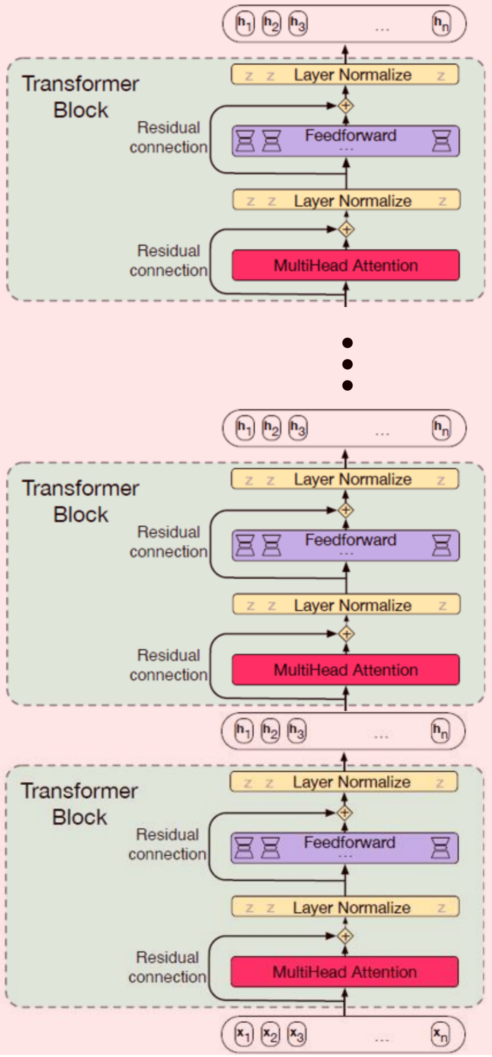


Transformer Blocks

Input and output dimensions of transformer blocks are matched so they can be stacked.

- Each token x_i at the input to the block has dimensionality d, and so input X and output H are both of shape [N×d].
- *Transformers for large language models stack many of these blocks.*
  - 12 layers are used for the T5 or GPT-3-small language models to 96 layers are used for GPT-3 large, to even more for more recent models.
- As embedding vectors pass up through the transformer layers, embedding representation will change and grow, incorporating context.

Stacked Transformer Blocks

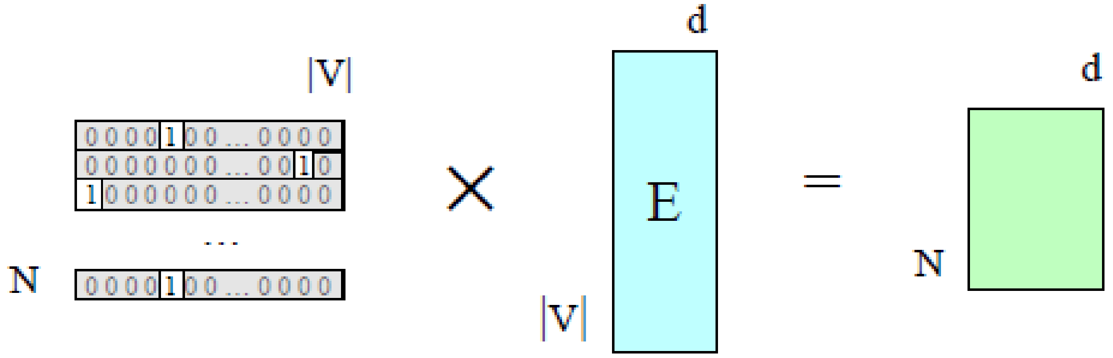


Positional Embeddings
input: embeddings for token

Input : Given a sequence of N tokens (N is the context length in tokens), the input is the matrix of shape [N×d] which has an embedding for each word in the context.

- The initial embeddings are stored in the embedding matrix E, which has a row for each of the |V| tokens in the vocabulary.
- Each each word is a row vector of d dimensions, and E has shape [|V|×d].
- For a given sequence of N tokens, the input matrix can be created using one-hot vectors for tokens.

- ***Selecting the embedding matrix for the input sequence of token ids W by multiplying a one-hot matrix corresponding to W by the embedding matrix E.***



Positional Embeddings

How does a transformer model the position of each token in the input sequence?

- With RNNs, information about the order of the inputs was built into the structure of the model, Not with Transformers

Solution: Positional Embeddings

- Modify the input embeddings by combining them with positional embeddings specific to each position in an input sequence

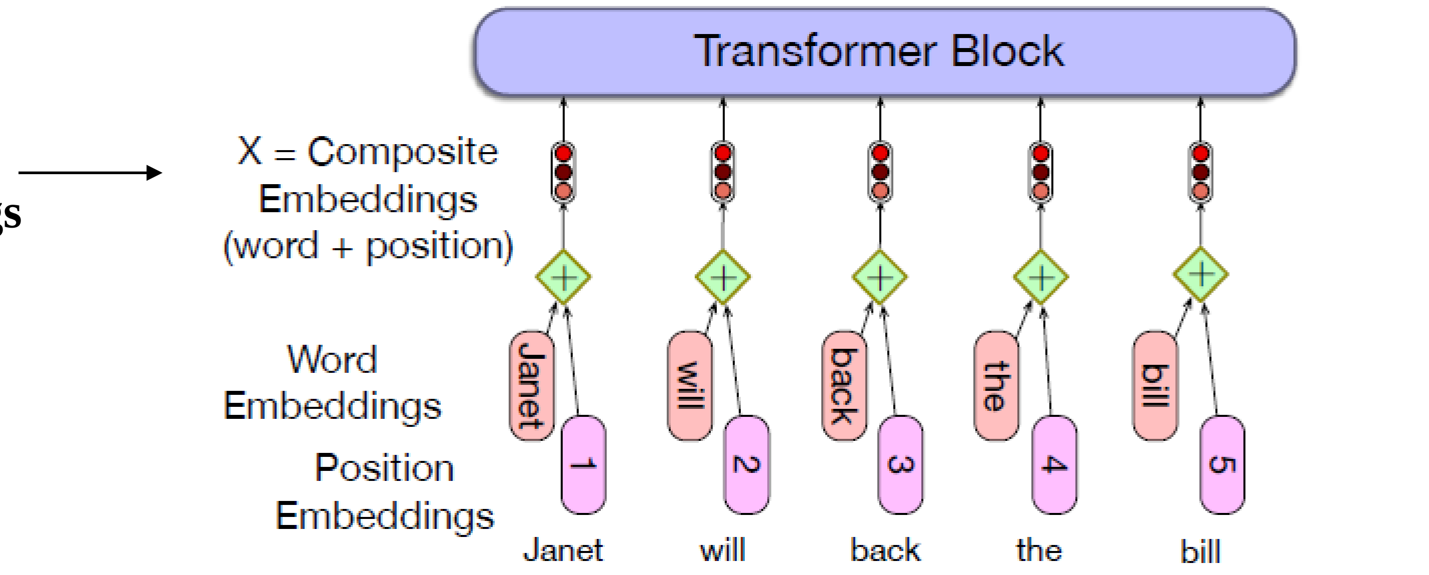
How do we get them? Start with randomly initialized embeddings corresponding to each possible input position up to some maximum length

- As with word embeddings, these positional embeddings are learned along with other parameters during training.
- To produce an input embedding that captures positional information, we just add the word embedding for each input to its corresponding positional embedding
  - The individual token and position embeddings are both of size [1×d], so their sum is also [1×d]. This new embedding serves as the input for further processing.

Positional Embeddings

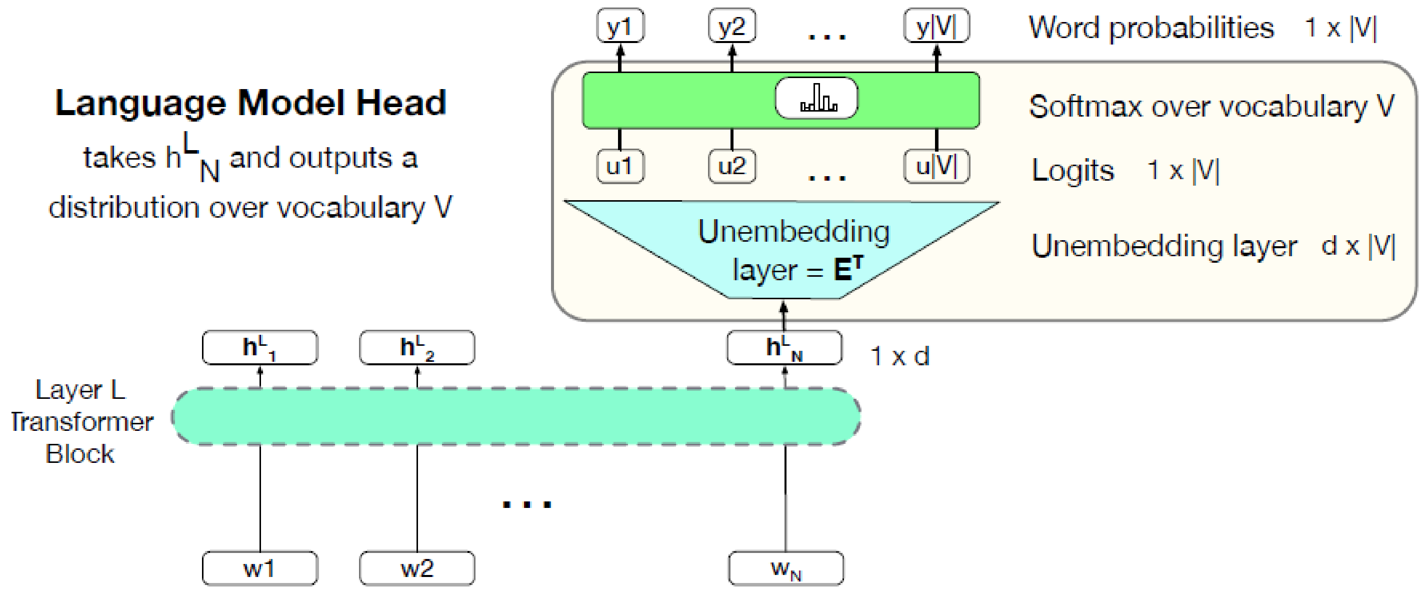
- A simple way to model position: add an embedding of the **absolute position** to the token embedding to produce a new embedding of the same dimensionality.

Positional Embeddings



Language Modeling Head

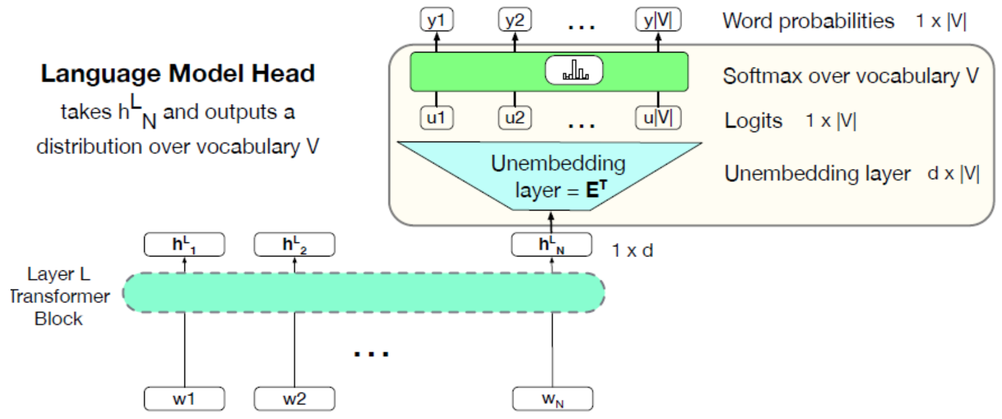
- The **language modeling head** is the *additional neural circuitry* that we add on top of the *basic transformer architecture* to enable the language modeling.
  - The ***job of the language modeling head*** is to take the output of the final transformer layer from the last token N and use it to predict the upcoming word at position N+1.


y1
y2
...
y|V|
Word probabilities 1 x |V|
Language Model Head
takes h^L_N and outputs a distribution over vocabulary V
Softmax over vocabulary V
u1
u2
...
u|V|
Logits 1 x |V|
Unembedding layer = E^T
Unembedding layer d x |V|
h^L_1
h^L_2
h^L_N
1 x d
Layer L Transformer Block
...
w1
w2
w_N

Language Modeling Head

- **Unembedding layer** is a linear layer to project from the output $\mathbf{h}^{\mathbf{L}}_{\mathbf{N}}$ which is the output token embedding at position N from the final block L, to a **logit vector** (a score vector that has a single score for each of the |V| possible words in the vocabulary V.
  - This linear layer can be learned, but more commonly this matrix is tied to (the transpose of) the embedding matrix E.
  - ➔ **weight tying**, we use the same weights for two different matrices in the model.
  - In the learning process, E will be optimized to be good at doing both of these mappings.
  - The transpose $\mathbf{E}^{\mathbf{T}}$ is called **unembedding layer** because it performs a reverse mapping.
  - A **softmax layer** turns the logits u into the probabilities y over the vocabulary.

$$\mathbf{u} = \mathbf{h}^{\mathbf{L}}_{\mathbf{N}}\ \mathbf{E}^{\mathbf{T}}$$
$$\mathbf{y} = \text{softmax}(\mathbf{u})$$

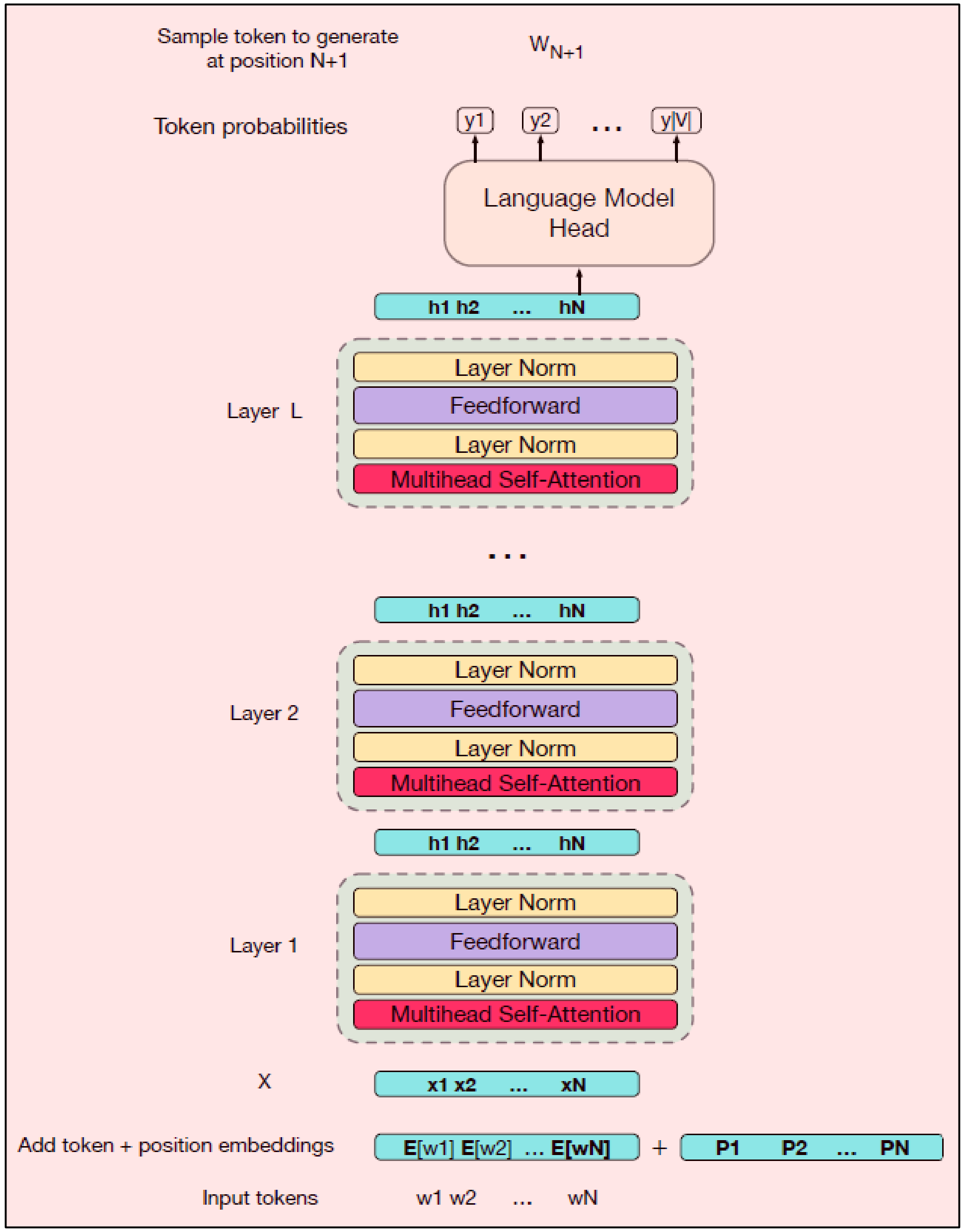

y1
y2
...
y|V|
Word probabilities 1 x |V|
Language Model Head
takes h^L_N and outputs a distribution over vocabulary V
Softmax over vocabulary V
u1
u2
...
u|V|
Logits 1 x |V|
Unembedding layer = E^T
Unembedding layer d x |V|
h^L_1
h^L_2
h^L_N
1 x d
Layer L Transformer Block
...
w1
w2
w_N

Language Modeling Head

a transformer decoder-only model

A **transformer decoder-only model**, stacking transformer blocks and mapping from input tokens w_1 to w_N to a predicted next word w_{N+1}.

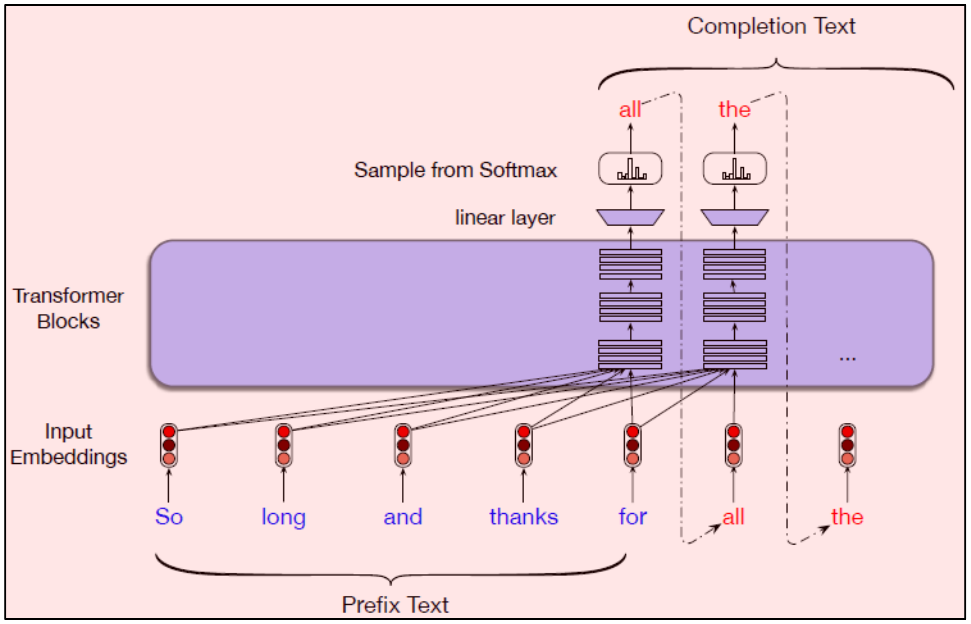
The input to the first transformer block is represented as X which is the N indexed word embeddings + position embeddings, but the input to all the other layers is the output H from the layer just below the current one.



Large Language Models with Transformers

Large Language Models with Transformers

- Many practical NLP tasks can be cast as **word prediction task** using ***transformer-based large language models***.
  - A powerful-enough language model can solve them with a high degree of accuracy.
- Many NLP tasks are ***cases of conditional generation***, the task of generating text conditioned on an input piece of text, a ***prompt***.
  - The fact that transformers have such long contexts (4096 tokens) makes them very powerful for conditional generation, because they can look back so far into the prompting text.



Autoregressive text completion with transformer-based large language models.

- A language model is given a text prefix and is asked to generate a possible completion.
- The generation process proceeds, the model has direct access to the priming context as well as to all of its own subsequently generated.
- This ability to incorporate the entirety of the earlier context and generated outputs at each time step is the key to the power of large language models built from transformers

Large Language Models with Transformers

NLP tasks as word prediction task

Sentiment Analysis as language modeling by giving a language model a context like:

The sentiment of the sentence "I like Jackie Chan" is:

and comparing the conditional probability of the words "positive" and the word "negative" to see which is higher:

P(***positive*** | *The sentiment of the sentence "I like Jackie Chan" is:*)

P(***negative*** | *The sentiment of the sentence "I like Jackie Chan" is:*)

If the word "positive" is more probable, we say the sentiment of the sentence is positive, otherwise we say the sentiment is negative.

Large Language Models with Transformers

NLP tasks as word prediction task

Question Answering as word prediction by giving a language model a question and a token like A: suggesting that an answer should come next:

Q: Who wrote the book ``The Origin of Species"? A:

If we ask a language model to compute

P(***w*** | *Q: Who wrote the book ``The Origin of Species"? A:*)

and look at which words w have high probabilities, we might expect to see that Charles is very likely, and then if we choose Charles and continue and ask

P(***w*** | *Q: Who wrote the book ``The Origin of Species"? A: Charles*)

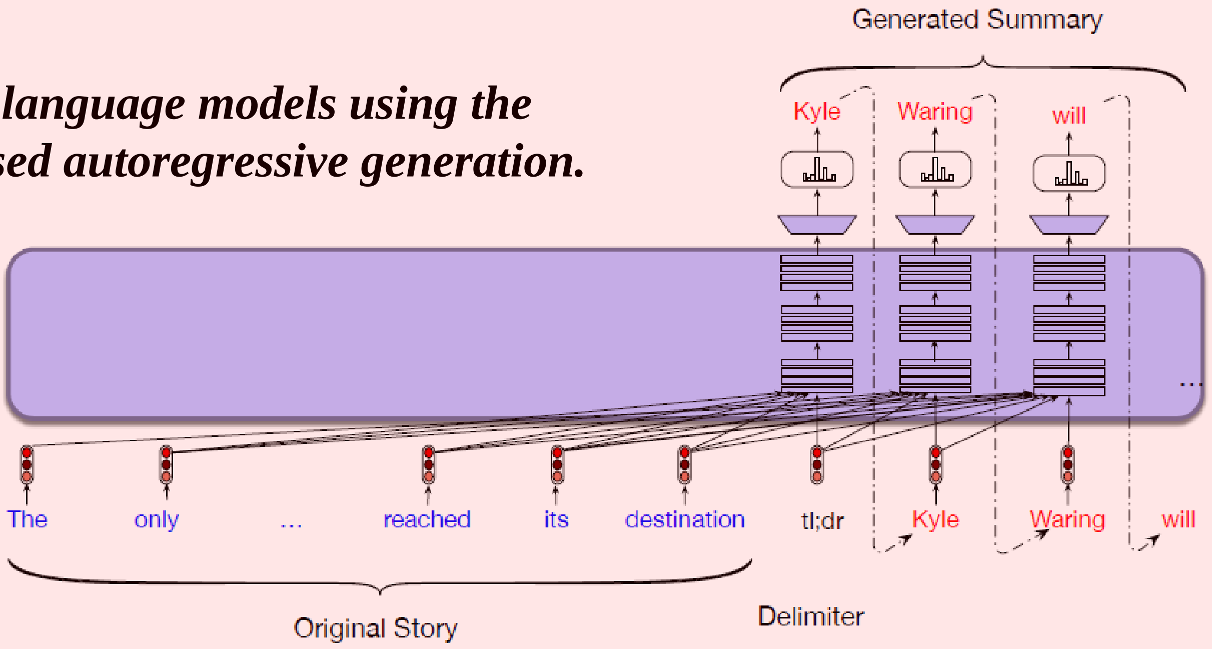
we might now see that Darwin is the most probable word, and select it.

Large Language Models with Transformers

NLP tasks as word prediction task

- **Text Summarization** takes a long text, such as a full-length article, and produce an effective shorter summary (still a long response) of it.
  - We can cast summarization as language modeling by giving a large language model a text, and follow the text by a token like tl;dr;
  - We can then do conditional generation: give the language model this prefix, and then ask it to generate the words, one by one, and take the entire response as a summary.

Summarization with large language models using the tl;dr token and context-based autoregressive generation.



Large Language Models with Transformers

NLP tasks as word prediction task - Text Summarization

- ***What makes transformers able to succeed at Text Summarization task?***
  - The ability of self-attention to incorporate information from the large context windows.
    - The model has access to the original article as well as to the newly generated text.
- ***Which words do we generate at each step?***
  - **Greedy decoding:** always generate the ***most likely word*** for a given the context.
  - At each time step in generation, the output y_t is chosen by computing the probability for each possible outputs and then choosing the highest probability word.
  - A major problem with **greedy decoding** is that because the words it chooses are extremely predictable, the resulting text is generic and often quite repetitive.

  - In most tasks, however, we prefer text which has been generated by more sophisticated methods, called **sampling methods**, that introduce a bit more diversity into the generations.

Large Language Models: Generation by Sampling

- The core of the generation process for large language models is the task of choosing the single word to generate next based on the context and based on the probabilities that the model assigns to possible words.
- Choosing a word to generate based on the model’s probabilities is called **decoding**.
- Repeatedly choosing the next word conditioned on our previous choices is called **autoregressive generation** (or **casual LM generation**).
- The most common method for ***decoding*** in large language models is **sampling**.
  - ***Sampling*** from a model’s distribution over words means to choose random words according to their probability assigned by the model.

Large Language Models: Generation by Sampling

random sampling

- **Random Sampling** generates a sequence of words $w_1, w_2, \ldots$ until it hits the end of sequence token.
  - To generate text from a trained transformer language model: at each step we'll sample words according to their probability conditioned on our previous choices, and we'll use a transformer language model as the probability model that tells us this probability.
  - Although random sampling mostly generates sensible, high-probable words, it may also generate low probability rare words:
    - Even though each rare word is low probability, if you add up all the rare words, they constitute a large enough portion of the distribution, and they can be chosen often enough to result in generating weird sentences.

- For this reason, instead of random sampling, we usually ***use sampling methods that avoid generating the very unlikely words***.

Large Language Models: Generation by Sampling

Top-k sampling

Top-k sampling is a simple generalization of ***greedy decoding***.

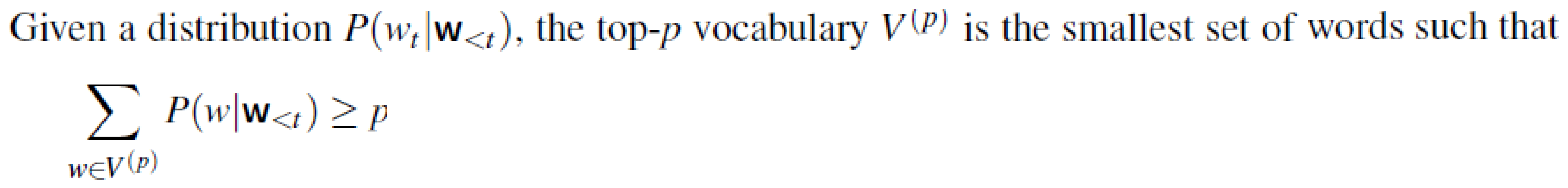
- Instead of choosing the single most probable word to generate,
  1. We first truncate the distribution to the top k most likely words,
  2. Renormalize to produce a legitimate probability distribution for top k words,
  3. Then randomly sample from within these k words according to their renormalized probabilities.

- When k = 1, **top-k sampling** is identical to ***greedy decoding***.
- Setting k to a larger number than 1 leads us to sometimes select a word which is not necessarily the most probable, but is still probable enough, and whose choice results in generating more diverse but still high-enough-quality text.

Large Language Models: Generation by Sampling

Top-p sampling

- One problem with ***top-k sampling*** is that k is fixed, but the shape of the the probability distribution over words differs in different contexts.
  - If we set k=10, sometimes the top 10 words will be very likely and include most of the probability mass, but other times the probability distribution will be flatter and the top 10 words will only include a small part of the probability mass.
- An alternative sampling method, called **top-p sampling** is to keep not the top k words, but the ***top p percent of the probability mass***.
  - The goal is same: to truncate the distribution to remove the very unlikely words.
  - But by measuring probability rather than the number of words, the hope is that the measure will be more robust in very different contexts, dynamically increasing and decreasing the pool of word candidates.



Given a distribution $P(w_t|\mathbf{w}_{<t})$, the top-p vocabulary $V^{(p)}$ is the smallest set of words such that

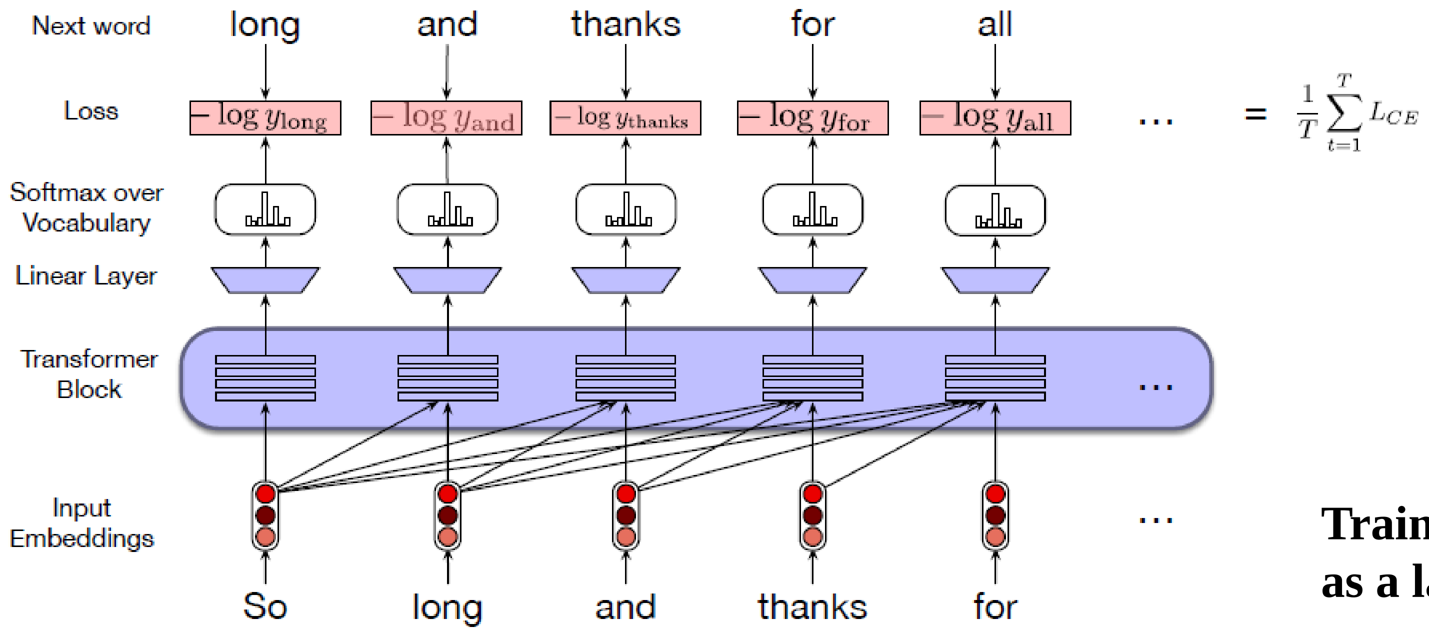
$$\sum_{w \in V^{(p)}} P(w|\mathbf{w}_{<t}) \geq p$$

Large Language Models: Training Transformers

- To ***train a transformer as a language model***, we use s**elf-supervisio**n algorithm
  - A corpus of text is training dataset and the model ***predicts next word at each time step t***.
  - A model is called **self-supervised** because we don't have to add any special gold labels to the data; the natural sequence of words is its own supervision!
  - We simply ***train the model to minimize the error in predicting the true next word*** in the training sequence, using cross-entropy as the loss function.
- The **cross-entropy loss** measures the difference between a ***predicted probability distribution*** and the ***correct distribution***.
- At each word position t of the input, the model takes as input the correct sequence of tokens $w_{1:t}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next token w_{t+1}.
  - Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct sequence of tokens $w_{1:t+1}$ to estimate the probability of token w_{t+2}.
  - This idea that we always give the model the correct history sequence to predict the next word is called **teacher forcing**.

Large Language Models: Training Transformers

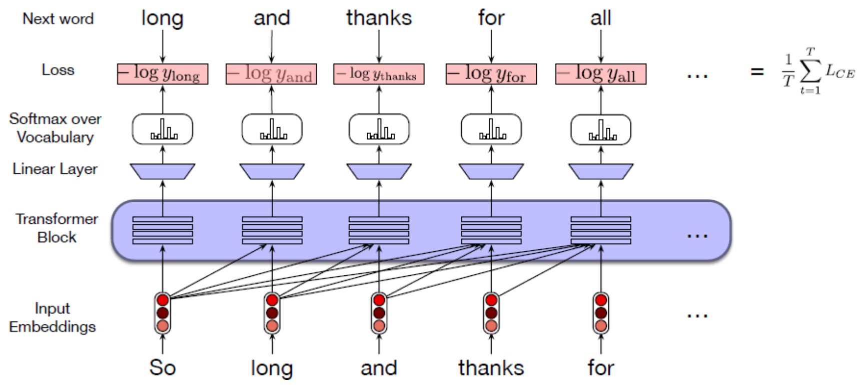
- During training, ***probability assigned to the correct word is used to calculate the cross-entropy loss for each item*** in the sequence.
- The **loss for a training sequence** is the ***average cross-entropy loss over the entire sequence***.
- The ***weights in the network are adjusted to minimize the average CE loss over the training sequence*** via **gradient descent**.



Training a transformer as a language model

Large Language Models: Training Transformers

- The key difference between ***training transformes*** and ***training RNNs*** as language models,:
  - The ***calculation of the outputs and the losses*** at each step is inherently **serial in RNNs** (given the recurrence in the calculation of the hidden states).
  - With **transformers**, each training item can be processed in **parallel** since the output for each element in the sequence is computed separately.
- Large models are generally trained by filling the full context window
  - For example 2048 or 4096 tokens for GPT3 or GPT4 with text.
  - If documents are shorter than this, multiple documents are packed into the window with a special end-of-text token between them.



Large Language Models: Training Transformers

- ***Training corpora for large language models are so huge***, they are likely to contain many natural examples that can be helpful for NLP tasks.
  - The GPT3 models, for example, are trained mostly on the web (429 billion tokens), some text from books (67 billion tokens) and Wikipedia (3 billion tokens).
- The ***performance of large language models*** are mainly determined by 3 factors:
  - **Model size** (the number of parameters not counting embeddings),
  - **Dataset size** (the amount of training data), and
  - **Amount of computer used for training**.
- The number of (non-embedding) parameters N can be roughly computed as follows (with **d** as the input and output dimensionality of the model, d_{attn} as the self-attention layer size, and d_{ff} the size of the feedforward layer):

$$N \approx 2\, d\, n_{\text{layer}}(2\, d_{\text{attn}} + d_{\text{ff}}) \approx 12\, n_{\text{layer}}\, d^2 \qquad (\text{assuming } d_{\text{attn}} = d_{\text{ff}}/4 = d)$$

  - Thus GPT-3, with n=96 layers and dimensionality d=12288, has $12 \times 96 \times 12882^2 \approx$ 175 billion parameters.

Potential Harms from Language Models

- Language models can generate texts that are false, this problem called **hallucination.**
  - Language models are trained to generate text that is predictable and coherent, but the training algorithms don't have any way to enforce that the generated text is correct or true.
- A second source of harm is that language models can generate **toxic language**.
  - Non-toxic prompts can lead large language models to output hate speech and abuse their users.
  - Language models can also generate stereotypes and negative attitudes about many demographic groups.
- Language models also present **privacy issues** since they can leak information about their training data.
  - It is thus possible for an adversary to extract training-data text from a language model such as an individual person's name, phone number, and address.
  - This is a problem if large language models are trained on private datasets such as electronic health records.

Summary

- **Transformers** are ***non-recurrent networks*** based on ***self-attention***.
  - A **self-attention** layer maps input sequences to output sequences of the same length, using ***attention heads*** that model how the surrounding words are relevant for the current word.
- A **transformer** block consists of a ***single attention layer*** followed by a **feedforward layer** with ***residual connections*** and ***layer normalizations*.**
- **Transformer blocks** can be ***stacked to make deeper and more powerful networks***.
- **Language models** can be built out of ***stacks of transformer blocks***, with a ***linear (unembedding layer)*** and ***softmax max layer*** at the top.
- ***Transformer-based language models*** have a **wide context window** (as wide as 4096 tokens for current models) allowing them to draw on enormous amounts of context to predict upcoming words.

Summary

- **Many NLP tasks** (question answering, summarization, sentiment analysis,…) can be cast as ***tasks of word prediction*** and hence addressed with large language models.

- The choice of which word to generate in large language models is generally done by using a **sampling algorithm**.

- Because of their ability to be used in so many ways, ***language models also have the potential to cause harms***.
  - Some harms include hallucinations, bias, stereotypes, misinformation and propaganda, and violations of privacy and copyright.