Deep Generative Models

Chapter 1: Text Generation via Language Models

Ali Bereyhi

`ali.bereyhi@utoronto.ca`

Department of Electrical and Computer Engineering

University of Toronto

Summer 2025

Transformer: *Revolutionary Sequence Models*

Transformer was proposed in 2017 in the paper

- Attention is All You Need!

providing a computationally-efficient was for sequence processing

Key Component of Transformers

Transformers use the Attention mechanism which enables parallel processing of sequences[1]

We use the Attention mechanism in the sequel to build context!

[1] Read more in Chapter 7 of *Applied Deep Learning* lecture notes

Key-Query-Value Tuple

Say we have *the sequence of tokens*

$$x_1, x_2, \ldots, x_t$$

and intend to build context $\mathbf{c}_t$ to find the distribution of next token x_{t+1}

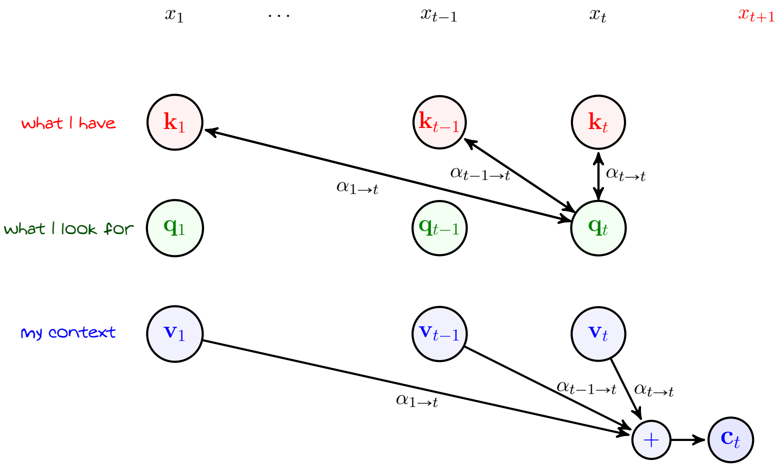
For each token in the sequence, we give three separate embeddings

- *Query $\mathbf{q}_t \in \mathbb{R}^S$ which flags what kind of context token x_t is looking for*
- *Key $\mathbf{k}_t \in \mathbb{R}^S$ which flags what kind of context token x_t has to present*
- *Value $\mathbf{v}_t \in \mathbb{R}^S$ which embeds the context of token x_t*

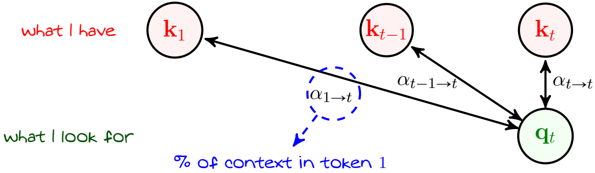
\+ *How do we compute these embeddings?*

– No worries! We'll see shortly!

Building Context by Attention



Computing Attention Weights



We can compare the query with each key using *inner product*

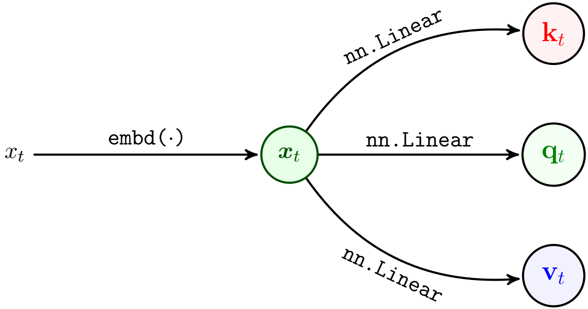
$$\xi_{j\to t} = \langle \mathbf{q}_t ; \mathbf{k}_j \rangle$$

The larger $\xi_{j\to t}$ is, the more related tokens j and t are!

But, we need weights $\in [0,1]$; so, we use Softmax

$$[\alpha_{j\to t} \text{ for } j = 1:t] = \mathit{Soft}_{\max} \left(\langle \mathbf{q}_t ; \mathbf{k}_j \rangle \text{ for } j = 1:t \right)$$

Embedding Tokens: *Key, Query and Value*



We typically compute the tuples by *linear projections of token embedding*

$$\mathbf{k}_t = \mathbf{W}_k \boldsymbol{x}_t \qquad \mathbf{q}_t = \mathbf{W}_q \boldsymbol{x}_t \qquad \mathbf{v}_t = \mathbf{W}_v \boldsymbol{x}_t$$

fore some $\mathbf{H}_k, \mathbf{W}_q, \mathbf{W}_v \in \mathbb{R}^{S \times E}$

Sequential Order

Say we have the following two sequences

$$x_{1:t} = x_1, x_2, x_3 \dots, x_t$$
$$\hat{x}_{1:t} = x_2, x_1, x_3 \dots, x_t$$

We have $\mathbf{k}_i = \mathbf{W}_k \boldsymbol{x}_j$*; thus, the keys see the same permutation*

$$\hat{\mathbf{k}}_{1:t} = \mathbf{k}_2, \mathbf{k}_1, \mathbf{k}_3 \dots, \mathbf{k}_t$$

If we compare with the query $\mathbf{q}_t$*, we get same permutation in weights*

$$\hat{\alpha}_{1\to t}, \hat{\alpha}_{2\to t}, \hat{\alpha}_{3\to t} \dots, \hat{\alpha}_{t\to t} = \alpha_{2\to t}, \alpha_{1\to t}, \alpha_{3\to t} \dots, \alpha_{t\to t}$$

Sequential Order

Weights observe the same permutation as input

$$\hat{\alpha}_{1\to t}, \hat{\alpha}_{2\to t}, \hat{\alpha}_{3\to t} \ldots, \hat{\alpha}_{t\to t} = \alpha_{2\to t}, \alpha_{1\to t}, \alpha_{3\to t} \ldots, \alpha_{t\to t}$$

We have further $\mathbf{v}_i = \mathbf{W}_v \boldsymbol{x}_j$; *thus, the values also see the same permutation*

$$\hat{\mathbf{v}}_{1:t} = \mathbf{v}_2, \mathbf{v}_1, \mathbf{v}_3 \ldots, \mathbf{v}_t$$

So, the context of permuted sequence is

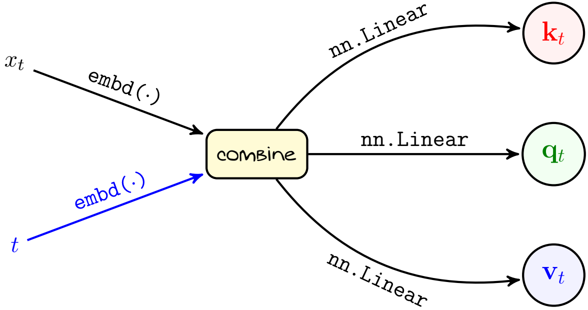
$$\begin{aligned}\hat{\mathbf{c}}_t &= \hat{\alpha}_{1\to t}\mathbf{v}_2 + \hat{\alpha}_{2\to t}\mathbf{v}_1 + \hat{\alpha}_{3\to t}\mathbf{v}_3 + \ldots + \hat{\alpha}_{t\to t}\mathbf{v}_t \\ &= \alpha_{2\to t}\mathbf{v}_2 + \alpha_{1\to t}\mathbf{v}_1 + \alpha_{3\to t}\mathbf{v}_3 + \ldots + \alpha_{t\to t}\mathbf{v}_t = \mathbf{c}_t\end{aligned}$$

Moral of Story

Using only token embedding we do not capture sequential order via attention!

Encoding Tokens *with Sequential Order*

We can capture sequential order by *positional encoding*

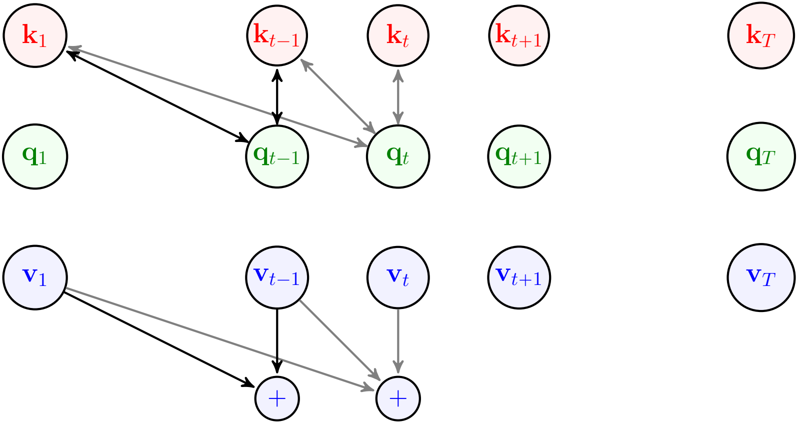


Classical combining approach is to add

$$\boldsymbol{x}_t = \mathtt{embd}(x_t) + \mathtt{embd}(t)$$

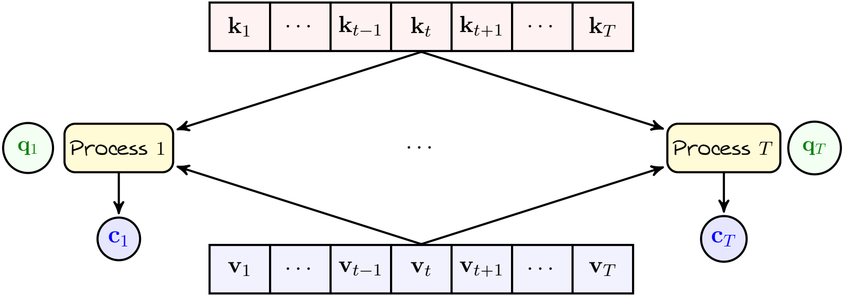
Parallel Context Computation

+ *Do we still need to process one token at a time?*
– No! *The cool part is that we can compute all contexts in parallel*



Parallel Context Computation

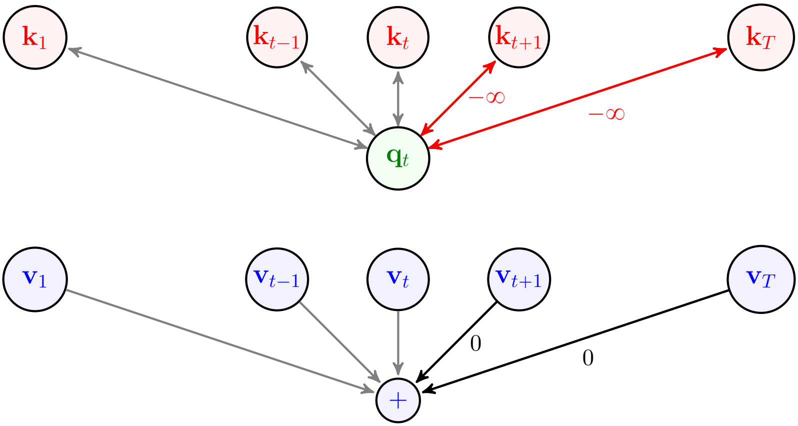
Indeed, we can break computation of $\mathbf{c}_1, \ldots, \mathbf{c}_T$ into T parallel processes



+ *But the number of weights in each process differ, right? Process 1 computes one coefficient, Process 2 computes two, $\cdots$*
– Yes! And, this can be easily unified ☺

Self-Attention with *Masked Decoding*

We can *get query from all keys in the sequence and rewrite them to keep the weights zero for future tokens* $\rightsquigarrow$ *replace future inner-products with* $-\infty$



Masked Decoding

+ *Why do we replace the inner products with $-\infty$?*
– To make the corresponding Softmax output zero

By setting $\exp\{\xi_{j\to t}\} = -\infty$ *for* $j > t$*, we have*

$$\alpha_{j\to t} = \frac{\exp\{\xi_{j\to t}\}}{\sum_{i=1}^{T} \exp\{\xi_{i\to t}\}} = \frac{\exp\{-\infty\}}{\sum_{i=1}^{T} \exp\{\xi_{i\to t}\}} = 0$$

+ *Why do we call it masked decoding?*
– Well! If we wanted *future tokens impact the current one*
  ↳ *we did not need to mask out* $\exp\{\xi_{j\to t}\}$ *for* $j > t$
  ↳ *this is used encoding, e.g., for translation machine*

Self-Attention: *Single Head*

SA_Head($x_{1:T}$:input_seq):

for $t = 1 : T$ **do**

  Set $\boldsymbol{x}_t \leftarrow \text{embd}\left(x_t\right) + \text{embd}\left(t\right)$ #positional enc

  $\mathbf{k}_t, \mathbf{q}_t, \mathbf{v}_t \leftarrow \texttt{Linear}\left(\boldsymbol{x}_t\right), \texttt{Linear}\left(\boldsymbol{x}_t\right), \texttt{Linear}\left(\boldsymbol{x}_t\right)$

  $\xi_{1:T \to t} = \langle \mathbf{k}_1 ; \mathbf{q}_t \rangle, \dots, \langle \mathbf{k}_T ; \mathbf{q}_t \rangle$ #query × keys

  if masked_decode = True **then**

    $\xi_{t+1:T \to t} = -\infty$

  end if

  $\alpha_{1:T \to t} = \mathit{Soft}_{\max}\left(\xi_{1:T \to t}\right)$ #attention weights

  $\mathbf{c}_t \leftarrow \sum_i \alpha_{i \to t} \mathbf{v}_i$ #build context

end for

return $\mathbf{c}_t$ for $t = 1 : T$

+ *Why calling it a Head?*
– Well! In practice *we can apply multiple of these heads in parallel to make a richer context*

Self-Attention: *Multi-Head*

```
SelfAttention(x_{1:T}:input_seq, H:# of heads):
 1: for h = 1 : H do
 2:     Compute c^h_{1:T} ← SA_Head(x_{1:T})           #independent heads
 3: end for
 4: Set c_t ← [c^1_{1:T}, ..., c^H_{1:T}]
 5: return c_t for t = 1 : T
```

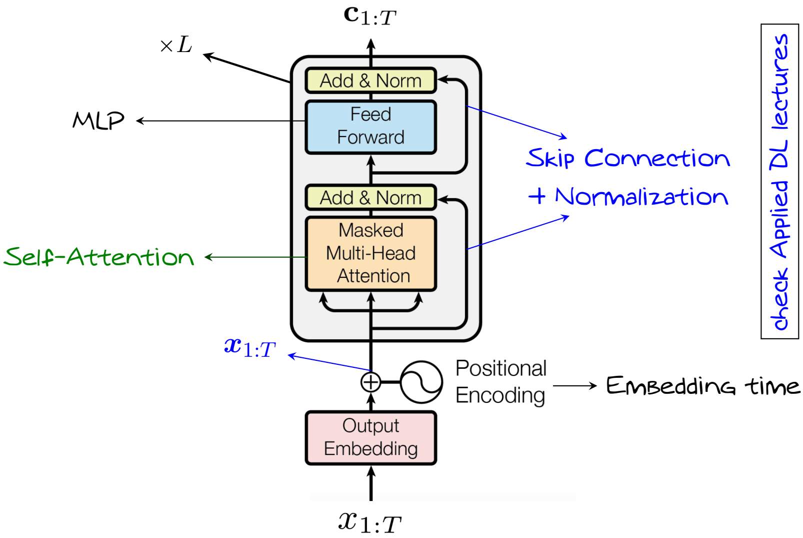
+ *Is this then a Transformer?*
– A shallow one! We can make it deep with L layers!

Transformer-Based LM: *Context Extractor*

```
TransformerDec(x_{1:T}, H:# of heads, L:depth):
1: for ℓ = 1 : L do
2:     c_{1:T} ← SelfAttention(x_{1:T}, H)
3:     c_{1:T} ← MLP(c_t) for t = 1 : T                  #extra computation
4: end for
5: return c_t for t = 1 : T
```

- + *What does the MLP do?*
- – It makes more complexity to the model
  - ↳ *This increases the model capacity*
  - ↳ *It can hence enhance the learning capability of the model*
- + *Good that we talk again "computational"!* ☺
- – We need to switch to *"statistical" talks* pretty soon ☺

Transformer-Based LM: *Overview*



check Applied DL lectures

Transformer-based LM: *Context Extractor*

+ *Wait a moment! We have not computed the distribution!*
– We can readily do it by a final MLP

```
TransformerLM(x_{1:T}, H:# of heads, L:depth):
1: c_{1:T} ← TransformerDec(x_{1:T}, H, L)
2: for t = 1 : T do
3:     p_{t+1} ← MLP(c_t)                    # p_{t+1} ∈ [0, 1]^I
4: end for
5: return p_{t+1} for t = 1 : T
```

Training Transformer-based LM

It's good to imagine how we can train this model

```
TransformerLM_Train(𝔻:train_set):
1: for multiple epochs do
2:     Sample a batch 𝔹 = {(x_{1:T+1})_b}                    #samples of T tokens
3:     for b = 1 : B do
4:         p_{2:T+1,b}, _ ← TransformerLM(x_{1:T,b}, H, L)
5:         Compute ∇_b ← −∑ ∇ log p_{t+1,b}[x_{t+1,b}]       #seq LL grad
6:     end for
7:     Update w ← w − ηOpt_avg{∇_b}                          #average batch
8: end for
```

Important

Training loop in this case can be extensively parallelized!

Generation via Transformer-based LMs

It's good to think how a this LM generates the whole new text

```
TransformerLM_Generation(x_{1:t}:input_text):
1: for i = t : T − 1 do
2:     x_{1:T+1} = x_{1:i}, 0, . . . , 0                          #add ∅ : 0
3:     p_{2:T+1} ← TransformerLM(x_{1:T−1}, H, L)                  #masked decode
4:     x_{i+1} ← multinomial(p_{i+1}, #smpl=1)                     #sample next token
5: end for
6: return {token[x_t] for t = 1 : T}                               #completed text
```

In practice though, we can improve computational efficiency by

- *We can drop number of key and query computations by caching*
- *We compute only the target token distribution in each iteration*
- . . .

Ready for LLMs!

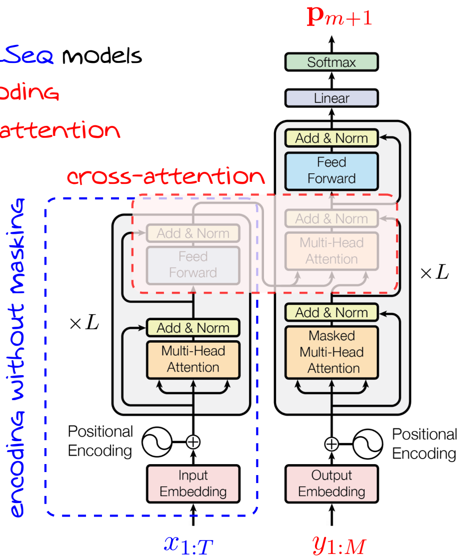
+ *Is this then an LLM?*
– Well! If scale it crazy up, Yes!
+ *How crazy should we scale?*
– Pretty crazy! ☺
+ *But, isn't this simply a text completer? How does it do what ChatGPT does?*
– Well! This is what we call *"Pre-trained LLM"*; we need to do some few extra steps to get there
+ *What are those steps?*
– No worries! We have the next section dedicated to LLMs

Final Note on Transformers

Transformers are Seq2Seq models

↳ our LM uses only decoding

↳ we don't need cross-attention



check Chapter 7 of Applied DL lectures