Chapter 6

Transformers

6.1 Model

In this section, we define transformers and relevant variants, and how transformers are used to describe formal languages. This section is adapted from Section 4 of the survey by Strobl, Merrill, et al. (2024).

Transformers are composed of an input layer (Section 6.1.1), one or more hidden layers (Section 6.1.4), and an output layer. The inputs and outputs of the layers are sequences of vectors, which we treat as members of \((\mathbb{R}^d)^*\).

6.1.1 Input layer

Strings are initially mapped to sequences of vectors by\( \text{emb}: \Sigma^* \rightarrow (\mathbb{R}^d)^* \), which is the sum of a \textit{word embedding} \(\text{WE}: \Sigma \rightarrow \mathbb{R}^d\) and a \textit{position(al) embedding} or \textit{encoding} \(\text{PE}_n: [n] \rightarrow \mathbb{R}^d\) for \(n \in \mathbb{N}_{>0}\):

\[
\text{emb}(w_0 \cdots w_{n-1})[i] = \text{WE}(w_i) + \text{PE}_n(i). \tag{6.1}
\]

In theoretical constructions, the word embedding can be any computable function.

The original transformer paper (Vaswani et al., 2017) introduced the following position embedding:

\[
\text{PE}_n(i)[j] = \begin{cases} 
10000^{-j/d} \sin i & \text{if } j \text{ even} \\
10000^{-(j-1)/d} \cos i & \text{if } j \text{ odd}.
\end{cases} \tag{6.2}
\]

Theoretical papers have explored other position embeddings, including \(i\) itself (Pérez et al., 2021), \(i/n\) (Yao et al., 2021; Chiang and Cholak, 2022), and \(1/i\) or \(1/i^2\) (Pérez et al., 2021).
6.1.2 Attention

Scaled dot-product self-attention with $d$ input/output dimensions and $d_{hid}$ key/value dimensions is a function

$$\text{att}: (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$$

$$X \mapsto Y$$

$$s[i, j] = \frac{\mathbf{W}_Q^T(X[i]) \cdot \mathbf{W}_K^T(X[j])}{\sqrt{d_{hid}}}$$

$$\alpha[i, :] = \text{softmax}(s[i, :])$$

$$= \frac{\exp s[i, :]}{\sum_{j \in [d]} \exp s[i, j]}$$

$$Y[i] = \sum_{j \in [n]} \alpha[i, j] \mathbf{W}_V^T(X[j])$$

with parameters

$$\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d_{hid} \times d}$$

$$\mathbf{W}_V \in \mathbb{R}^{d \times d}$$

We call the $s[i, j]$ the attention scores, and we call the $\alpha[i, j]$ the attention weights.

Real transformers use multi-head self-attention, but we don’t use it because it can be emulated using single-head self-attention.

Attention masking In future-masked (also known as causally-masked) self-attention, a term $m(i, j)$ is added to Eq. (6.4) to force every position to attend only to preceding positions:

$$m(i, j) = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{otherwise.} \end{cases}$$

(We define $\exp(-\infty) = 0$.) Some papers use strict future-masking, that is, $m(i, j) = 0$ iff $j < i$, and occasionally past-masking ($j \geq i$) and strict past-masking ($j > i$).

Hard attention Some theoretical analyses simplify attention by replacing the softmax with variants that focus attention only on the position(s) with the maximum value, breaking ties in various ways.

For any vector $x \in \mathbb{R}^d$, define $M(x) = \{i \in [n] \mid \forall j \in [n], x[j] \leq x[i]\}$ to be the set of indices of the maximal elements of $x$. In leftmost-hard attention, the leftmost maximal element is used, replacing Eq. (6.5) with:

$$\alpha[i, j] = \mathbb{I}[j = \min M(s[i, :])]$$

\[6.9\]
whereas in average-hard attention, the maximal elements share weight equally:

\[
\alpha[i, j] = \frac{[j \in M(s[i,:])]}{|M(s[i,:])|}.
\]  

(6.10)

Leftmost-hard attention was previously called hard attention by Hahn (2020) and unique-hard attention by Hao et al. (2022). One may also consider rightmost-hard attention, in which the rightmost maximal element is used. Average-hard attention was also called hard attention by Pérez et al. (2021) and saturated attention by Merrill, Sabharwal, and Smith (2022), and has been argued to be a realistic approximation to how trained transformers behave in practice (Merrill, Ramanujan, et al., 2021). Neither type of hard attention should be confused with the concept of hard attention used in computer vision (e.g., Xu et al., 2015).

### 6.1.3 Layer normalization

A \( d \)-dimensional layer normalization (Ba et al., 2016), or layernorm for short, is a function

\[
\text{norm}: \mathbb{R}^d \rightarrow \mathbb{R}^d \\
\mathbf{x} \mapsto \gamma \odot \frac{\mathbf{x} - \bar{x}}{\sqrt{\text{Var}(\mathbf{x}) + \varepsilon}} + \beta
\]  

(6.11)

where \( \odot \) is component-wise multiplication,

\[
\bar{x} = \frac{1}{d} \sum_{i \in [d]} x_i 
\]  

(6.12)

\[
\text{Var}(\mathbf{x}) = \frac{1}{d} \sum_{i \in [d]} (x_i - \bar{x})^2
\]  

(6.13)

and the parameters are

\[
\gamma, \beta \in \mathbb{R}^d \\
\varepsilon \geq 0.
\]

The original definition of layernorm (Ba et al., 2016) sets \( \varepsilon = 0 \), but, for numerical stability, and to avoid division by zero, all implementations we are aware of set \( \varepsilon > 0 \). Observe that \( \text{norm} \) is Lipschitz-continuous iff \( \varepsilon > 0 \).

Some transformer analyses omit layernorm for simplicity (e.g., Pérez et al., 2021).
### 6.1.4 Hidden layers

A transformer layer (also known as a block) comes in two variants. The post-norm variant (Vaswani et al., 2017) is

\[
\text{layer}: (\mathbb{R}^d)^* \xrightarrow{\text{h}} (\mathbb{R}^d)^* \\
X \mapsto Y \text{ where }
H = \text{norm}_1(X + \text{att}(X)) \\
Y = \text{norm}_2(H + \text{ffn}(H))
\]

(6.14)

and the pre-norm variant (Wang et al., 2019) has

\[
H = X + \text{att}(\text{norm}_1(X)) \\
Y = H + \text{ffn}(\text{norm}_2(H))
\]

(6.15)

where

- \(\text{att}\) is a self-attention with \(d\) input/output dimensions and \(d_{\text{hid}}\) key/value dimensions

- \(\text{ffn}\) is a feed-forward network with \(d\) input/output dimensions and two layers, one ReLU and one linear. In applications, the hidden layer usually has \(4d\) dimensions, but theoretical constructions use as many or as few dimensions as needed.

- \(\text{norm}_1\) and \(\text{norm}_2\) are layernorms with \(d\) dimensions.

In both variants, the \(X+\) and \(H+\) terms are called residual connections (He et al., 2015), also known as skip connections.

### 6.1.5 Transformer encoders

A post-norm transformer encoder is a length-preserving function

\[
tfr: \Sigma^* \xrightarrow{w} (\mathbb{R}^d)^* \\
w \mapsto H^{(L)} \text{ where }
H^{(0)} = \text{emb}(w) \\
H^{(1)} = \text{layer}_1(H^{(0)}) \\
\vdots \\
H^{(L)} = \text{layer}_L(H^{(L-1)})
\]

(6.16)

where
Chapter 6. Transformers

- $e$ is an input layer
- $L$ is called the depth
- each $layer_\ell$ for $\ell \in [L]$ is a post-norm transformer layer (6.14).

A pre-norm transformer encoder is additionally parameterized by the weights of a final layernorm $norm$ and is defined as:

$$
tfr : \Sigma^* \xrightarrow{b} (\mathbb{R}^d)^* \\
\quad w \mapsto norm(H^{(L)}) \text{ where}
\quad H^{(0)} = emb(w) \\
\quad H^{(1)} = layer_1(H^{(0)}) \\
\quad \vdots \\
\quad H^{(L)} = layer_L(H^{(L-1)})
$$

(6.17)

where
- $e$ is an input layer
- $L$ is called the depth
- each $layer_\ell$ for $\ell \in [L]$ is a pre-norm transformer layer (6.15)
- $norm$ is a layernorm.

The encoder’s output is a sequence of vectors in $(\mathbb{R}^d)^*$. To use it as a language recognizer, we add a linear output layer

$$
out : \mathbb{R}^d \rightarrow \mathbb{R} \\
\quad h \mapsto w \cdot h + b \quad i \in [n]
$$

(6.18)

where parameters $w \in \mathbb{R}^d$, and $b \in \mathbb{R}$. The encoder accepts iff $out(tfr(w)[n]) \geq 0$.

### 6.2 Expressivity

#### 6.2.1 Decoders with intermediate steps

Transformer decoders A transformer decoder is a transformer encoder $tfr$ with future masking in its attention, typically used to generate rather than recognize strings. GPT and its competitor LLMs are all transformer decoders.
We assume that $\Sigma$ contains a special symbol BOS that does not occur anywhere else; later, we will add several other special symbols. The input string is the prefix of previously-generated symbols, $w_{<t} = w_0 \cdots w_{t-1}$, where $w_0 =$ BOS. The output is a probability distribution $\hat{p}(w_t | w_{<t})$ over the next symbol,

$$\text{out}: \mathbb{R}^d \to \mathbb{R}$$

$$x \mapsto Wx + b$$

$$\hat{p}(\cdot | w_{<t}) = \text{softmax(out}(tfr(w_{<t})[t-1]))$$

where parameters $W \in \mathbb{R}^{[K] \times d}$ and $b \in \mathbb{R}^{[K]}$.

To sample a string, we first sample $w_1$ from $\hat{p}(w_1 | \text{BOS})$, then, for each time step $t > 1$, sample $w_t$ from $\hat{p}(w_t | w_{<t})$. The process stops when $w_t = \text{EOS}$. Because each sampled output symbol becomes part of the input at the next time step, this kind of model is called autoregressive. See Fig. 6.1a.

In most (not all) theoretical papers about transformer decoders, we want the decoder to output a single next symbol instead of a probability distribution over next symbols. To do this, we can select the argmax of out$(tfr(w_{<t})[t-1])$ instead. Warning: In general, selecting the argmax at each step does not give you the highest-probability string.

We can also provide a prompt $x$ to the decoder, which the decoder can see as part of its input but doesn’t have to output. In that case, for $t \geq 1$, the input string is $x \cdot \text{BOS} \cdot y_{<t}$, and the output is $y_t$. See Fig. 6.1b.

**Intermediate steps** In many applications of transformer decoders, the prompt $x$ is some kind of question, and the desired output $y$ is the answer. For example, $x = 101 \cdot 101$ and $y = 10201$. Researchers have found in practice that sometimes a transformer decoder isn’t very good at answering certain kinds of questions, but if one allows the decoder to insert a number of intermediate time steps between the prompt and the final output, it sometimes performs much better. This is known as a scratchpad (Nye et al., 2022) or chain of thought (Wei et al., 2022).
Here, we’re only interested in the case where the final output is a single symbol, so we have the following definition.

**Definition 6.1.** Let \( f \) be a transformer decoder. For any string \( x \in \Sigma^* \), we say that \( f \), on prompt \( x \), outputs \( y_T \) after \( T \) intermediate steps if there is a string \( y = y_1 \cdots y_T \) such that for all \( t = 1, \ldots, T \), we have \( f(x \cdot \text{BOS} \cdot y_1 \cdots y_{t-1}) = y_t \).

**Turing machines** We assume that you are familiar with Turing machines. We use the definition of Turing machine in the textbook by Sipser (2013), with one small modification. Just to make sure we’re on the same page, we give the barest of definitions here.

**Definition 6.2.** A Turing machine is a tuple \( M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}) \), where

- \( Q \) is a finite set of states
- \( \Sigma \) is a finite input alphabet, where \( \_ \not\in \Sigma \)
- \( \Gamma \) is a finite tape alphabet, where \( \Sigma \cup \{\_\} \subseteq \Gamma \)
- \( \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{-1, +1\} \) is the transition function.

The tape has a left end and extends infinitely to the right. On input \( w \in \Sigma^* \), the tape is initialized to \( w_\_ \_ \_ \cdots \). If the current state is \( q \), the current tape symbol is \( a \), and \( \delta(q, a) = (r, b, m) \), then the machine enters state \( r \), writes a \( b \), and moves left if \( m = -1 \), right if \( m = +1 \). If the machine enters state \( q_{\text{accept}} \), it halts and accepts \( w \); if it enters state \( q_{\text{reject}} \), it halts and rejects \( w \).

**Example 6.3.** Here’s an example Turing machine (Sipser, 2013), with \( q_{\text{start}} = q_1 \), \( q_{\text{accept}} = q_6 \), \( q_{\text{reject}} = q_7 \). It decides the language \( \{1^{2n} \mid n \geq 0\} \).
The reject state $q_7$ appears twice to reduce clutter.

The (not very exciting) run of this machine on string $11$ is:

- $q_1 \xrightarrow{1} \cdot \cdot \cdot$
- $q_2 \xrightarrow{1} \cdot \cdot \cdot$
- $q_3 \xrightarrow{x} \cdot \cdot \cdot$
- $q_4 \xrightarrow{x} \cdot \cdot \cdot$
- $q_5 \xrightarrow{x} \cdot \cdot \cdot$
- $q_6 \xrightarrow{x} \cdot \cdot \cdot$
- accept

### Simulating Turing machines

**Theorem 6.4.** For any Turing machine $M$ with input alphabet $\Sigma$, there is a transformer decoder $f$ with average-hard attention that is equivalent to $M$ in the following sense. For any string $w \in \Sigma^*$:

- If $M$ halts and accepts on input $w$, then there is a $T$ such that $f$, on prompt $w$, outputs ACC after $T$ intermediate steps.
- If $M$ halts and rejects on input $w$, then there is a $T$ such that $f$, on prompt $w$, outputs REJ after $T$ intermediate steps.
- If $M$ does not halt on input $w$, then there does not exist a $T$ such that $f$, on prompt $w$, outputs either ACC or REJ.

The rest of this section proves the above theorem. There are several related proofs in the literature (Pérez et al., 2021; Bhattamishra et al., 2020; Merrill and Sabharwal, 2024); ours is an amalgam of these.

Let $M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}})$. The alphabet of $f.tfr$ is $\Sigma \cup \{\text{BOS}, \text{ACC}, \text{REJ}\} \cup (Q \times \Gamma \times \{-1, +1\})$. At each time step starting with BOS, the network outputs a triple $(r, b, m) \in (Q \times \Gamma \times \{-1, +1\})$ indicating what the next simulated action of $M$ is.

Each time step $i = 1, 2, \ldots$ of the transformer proceeds as follows.

1. Unpack the current input symbol, $x^{(i)}$:
   - If $x^{(i)} \in \Sigma \cup \text{BOS}$, let $q^{(i)} = \bot$ and $m^{(i-1)} = 0$.
   - Else, let $(q^{(i)}, b^{(i-1)}, m^{(i-1)}) = x^{(i)}$.

2. Compute the head position: $h^{(i)} = \sum_{j=0}^{i-1} m^{(j)}$.

3. Compute the symbol under the head, $a^{(i)}$;
Chapter 6. Transformers

4. Compute the next transition:

- If \( x^{(i)} \in \Sigma \), just output \( y^{(i)} = x^{(i)} \). (It will be ignored anyway)
- Else, if \( x^{(i)} = \text{BOS} \), let \( q^{(i+1)} = q_{\text{start}}, b^{(i)} = a^{(i)}, \) and \( m^{(i)} = 0 \).
- Else, let \( (q^{(i+1)}, b^{(i)}, m^{(i)}) = \delta(q^{(i)}, a^{(i)}) \).
- If \( q^{(i+1)} = q_{\text{accept}} \) or \( q_{\text{reject}} \), output \( y^{(i)} = \text{ACC} \) or \( y^{(i)} = \text{REJ} \), respectively.
- Else, output \( y^{(i)} = (q^{(i+1)}, b^{(i)}, m^{(i)}) \).

For example, the run from Example 6.3 is simulated by:

<table>
<thead>
<tr>
<th>( i )</th>
<th>tape</th>
<th>( x^{(i)} )</th>
<th>( q^{(i)} )</th>
<th>( b^{(i-1)} )</th>
<th>( m^{(i-1)} )</th>
<th>( h^{(i)} )</th>
<th>( a^{(i)} )</th>
<th>( y^{(i)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \hat{1} )</td>
<td>( 1 )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( \hat{1} )</td>
<td>( 1 )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>3</td>
<td>BOS</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( (q_1, 1, 0) )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( \hat{1} \hat{1} \cdots )</td>
<td>( (q_1, 1, 0) )</td>
<td>( q_1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( (q_2, \bot, +1) )</td>
</tr>
<tr>
<td>5</td>
<td>( \hat{1} \hat{1} \cdots )</td>
<td>( (q_2, \bot, +1) )</td>
<td>( q_2 )</td>
<td>( \bot )</td>
<td>( +1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( (q_3, x, +1) )</td>
</tr>
<tr>
<td>6</td>
<td>( \hat{1} \hat{1} \cdots )</td>
<td>( (q_3, x, +1) )</td>
<td>( q_3 )</td>
<td>( x )</td>
<td>( +1 )</td>
<td>( 2 )</td>
<td>( \bot )</td>
<td>( (q_5, \bot, -1) )</td>
</tr>
<tr>
<td>7</td>
<td>( \hat{1} \hat{1} \cdots )</td>
<td>( (q_5, x, -1) )</td>
<td>( q_5 )</td>
<td>( \bot )</td>
<td>( -1 )</td>
<td>( 1 )</td>
<td>( x )</td>
<td>( (q_5, x, -1) )</td>
</tr>
<tr>
<td>8</td>
<td>( \hat{1} \hat{1} \cdots )</td>
<td>( (q_5, x, -1) )</td>
<td>( q_5 )</td>
<td>( x )</td>
<td>( -1 )</td>
<td>( 0 )</td>
<td>( \bot )</td>
<td>( (q_2, \bot, +1) )</td>
</tr>
<tr>
<td>9</td>
<td>( \hat{1} \hat{1} \cdots )</td>
<td>( (q_2, \bot, +1) )</td>
<td>( q_2 )</td>
<td>( \bot )</td>
<td>( +1 )</td>
<td>( 1 )</td>
<td>( x )</td>
<td>( \text{ACC} )</td>
</tr>
</tbody>
</table>

Now we have to construct transformer layers to perform the above steps. Each input vector contains a word embedding, \( e_{x^{(i)}} \), and a position embedding with 4 components:

\[
H^{(0)}[i] = \begin{bmatrix}
e_{x^{(i)}} \\
1/i \\
1 \\
i \\
i^2
\end{bmatrix}.
\] (6.21)
Step 1 is piecewise linear, so it can be computed by a FFNN (Theorem 3.6). Afterwards, the activation vector at position $i$ is:

$$H^{(1)}[i] = \begin{bmatrix} e_{x(i)} \\ 1/i \\ 1 \\ i \\ i^2 \\ e_{q(i)} \\ e_{h(i-1)} \\ m_{(i-1)}^{(i-1)} \end{bmatrix}.$$  \hspace{1cm} (6.22)

Step 2 can be computed by a uniform self-attention layer:

$$q^{(i)} = 0 \quad k^{(j)} = 0 \quad v^{(j)} = \begin{bmatrix} 0 \\ m_{(i-1)}^{(i-1)} \end{bmatrix}. \hspace{1cm} (6.23)$$

Uniform self-attention computes an average, not a sum. Since at time step $i$, there are $i$ positions to average over, the result is $h^{(i)}/i$, not $h^{(i)}$. We’ll correct this in the next step.

$$H^{(2)}[i] = \begin{bmatrix} e_{x(i)} \\ 1/i \\ 1 \\ i \\ i^2 \\ e_{q(i)} \\ e_{h(i-1)} \\ m_{(i-1)}^{(i-1)} \end{bmatrix}. \hspace{1cm} (6.24)$$

Step 3 is the most difficult step. There are several schemes that have been proposed for this. All of them use average-hard attention, and all of them further modify the transformer in some way: changing dot-product to something else (Pérez et al., 2021), adding components to the position embedding (Pérez et al., 2021; Barceló et al., 2024; Strobl, Angluin, et al., 2024), or applying layer normalization only to selected components (Merrill and Sabharwal, 2024). The construction here is closest to that of Strobl, Angluin, et al. (2024).

It uses two self-attention layers. The first layer changes $h^{(i)}/i$ to $h^{(i)}$ by treating the position embeddings as a lookup table (Barceló et al., 2024).

$$q^{(i)} = \begin{bmatrix} 2h^{(i)}/i \\ -1/i \end{bmatrix} \quad k^{(j)} = \begin{bmatrix} j \\ j^2 \end{bmatrix} \quad v^{(j)} = \begin{bmatrix} 0 \\ j \\ j^2 \\ e_{x(j)} \end{bmatrix}. \hspace{1cm} (6.25)$$
\[ q^{(i)} \cdot k^{(j)} = \frac{1}{i} j(2h^{(i)} - j). \] (6.26)

Then the attention scores are uniquely maximized when \( j = h^{(i)} \):

\[
\begin{align*}
q^{(i)} \cdot k^{(j)} &\quad h^{(j)} \\
0 &\quad 0 \\
2h^{(i)} &\quad h^{(i)} \\
\end{align*}
\]

So the attention layer outputs \( h^{(i)} \), and also some other quantities which we’ll need shortly. The activation vector at position \( i \) is:

\[
H^{3.5}[i] = \begin{bmatrix}
    e_{\hat{v}^{(i)}} \\
e_{\hat{v}^{(i)}} \\
1/i \\
1 \\
i \\
i^2 \\
\end{bmatrix}
\]

Recall that we need to search positions \( j < i \) such that \( h^{(j)} = h^{(i)} \). But future-masked attention looks at positions \( j \leq i \). To get around this, we search positions \( j \leq i \) such that \( h^{(j-1)} = h^{(i)} \). So we will need

\[
h^{(j-1)} = h^{(i)} - m^{(i-1)}
\]

(6.28)

\[
(h^{(j-1)})^2 = (h^{(i)})^2 - 2h^{(i)}m^{(i-1)} + (m^{(i-1)})^2
\]

(6.29)

\[
= \begin{cases} 
    (h^{(i)})^2 + 2h^{(i)} + 1 & \text{if } m^{(i-1)} = -1 \\
    (h^{(i)})^2 - 2h^{(i)} + 1 & \text{if } m^{(i-1)} = +1 
\end{cases}
\]

(6.30)
which can both be computed using a FFN. So

\[ H^{(4)}[i] = \begin{bmatrix} e_{q(i)} \\ 1/i \\ 1 \\ i \\ i^2 \\ e_{q(i)} \\ e_{p(i-1)} \\ m_{(i-1)} \\ h(i)/i \\ h(i) \\ (h(i))^2 \\ e_{r(h(i))} \\ h_{(i-1)}^2 \end{bmatrix}. \] (6.31)

The second self-attention layer uses another variation of the lookup trick:

\[
\begin{align*}
q^{(i)} &= \begin{bmatrix} 2h^{(i)} \\ -1 \\ \frac{1}{2i} \end{bmatrix} & k^{(j)} &= \begin{bmatrix} h^{(j-1)} \\ (h^{(j-1)})^2 \\ j \end{bmatrix} & v^{(j)} &= \begin{bmatrix} 0 \\ h^{(j-1)} \\ e_{v(j-1)} \end{bmatrix} & e_{q(i)} \\ e_{p(i-1)} \\ m_{(i-1)} \\ h(i)/i \\ h(i) \\ (h(i))^2 \\ e_{r(h(i))} \\ h_{(i-1)}^2 \end{bmatrix}
\end{align*}
\] (6.32)

\[
q^{(i)} \cdot k^{(j)} = h^{(j-1)}(2h^{(i)} - h^{(j-1)}) + \frac{j}{2i}
\]

(6.33)

The first term is maximized when \( h^{(j-1)} = h^{(i)} \). If there is more than one such position \( j \), then because of the second term, \( j^* \) is the rightmost such \( j \). If there’s
no such position \( j \), then \( h^{(j'-1)} \neq h^{(i)} \).

\[
H^{(4,5)}[i] = \begin{bmatrix}
    e_x^{(i)} \\
    1/i \\
    1 \\
    i \\
    i^2 \\
    e_g^{(i)} \\
    e_{h^{(i-1)}}^{(i-1)} \\
    m^{(i-1)} \\
    h^{(i-1)}/i \\
    h^{(i)}/i \\
    (h^{(i)})^2 \\
    e_{x(h^{(i)})} \\
    h^{(i-1)} \\
    (h^{(i-1)})^2 \\
    h^{(i-1)}/2 \\
    h^{(i-1)} \\
    e_{h^{(i-1)}}^{(i-1)} \\
    e_{h^{(j'-1)}}^{(j'-1)} \\
    e_{d^{(i)}}^{(i)}
\end{bmatrix}.
\]  

Finally, we can use the FFNN to set

\[
a^{(i)} = \begin{cases} 
    b^{(j'-1)} & \text{if } h^{(j'-1)} = h^{(i)} \text{ and } b^{(j'-1)} \neq \bot \\
    \chi^{(h^{(i)})} & \text{otherwise.}
\end{cases}
\]  

(6.35)

So the activation vectors are:

\[
H^{(5)}[i] = \begin{bmatrix}
    e_x^{(i)} \\
    1/i \\
    1 \\
    i \\
    i^2 \\
    e_g^{(i)} \\
    e_{h^{(i-1)}}^{(i-1)} \\
    m^{(i-1)} \\
    h^{(i-1)}/i \\
    h^{(i)}/i \\
    (h^{(i)})^2 \\
    e_{x(h^{(i)})} \\
    h^{(i-1)} \\
    (h^{(i-1)})^2 \\
    h^{(i-1)}/2 \\
    h^{(i-1)} \\
    e_{h^{(i-1)}}^{(i-1)} \\
    e_{h^{(j'-1)}}^{(j'-1)} \\
    e_{d^{(i)}}^{(i)}
\end{bmatrix}.
\]  

(6.36)
Step 4 is piecewise linear, so it can be computed by a FFNN (Theorem 3.6).

6.2.2 Encoders with unique-hard attention

Unique-hard attention has been studied in several papers (Hahn, 2020; Hao et al., 2022; Barceló et al., 2024). Here, we look at the results of Angluin et al. (2023), which exactly characterize transformers with rightmost hard attention and strict future masking.

Strict future masking means that each position \( i \) attends to positions \( j < i \). If \( i \) is the leftmost position, all positions are masked out, so (following Merrill and Sabharwal (2024)) the attention output is just the zero vector.

**Theorem 6.5.** For any transformer encoder \( T \) with rightmost hard attention and strict future masking, there is a closed formula of FO that defines the same language that \( T \) recognizes.

The proof hinges on the fact that in a unique hard attention transformer, each activation vector depends on at most two vectors from the layer below. Because the network has fixed, finite depth, there is a fixed, finite number of possible activation vectors that it can compute.

**Lemma 6.6.** Let \( T \) be a unique (leftmost or rightmost) hard attention transformer. There is a finite set \( \mathcal{F} \subseteq \mathbb{R} \) such that for any input string \( w \), all the attention scores and activation values computed by \( T(w) \) belong to \( \mathcal{F} \).

**Proof.** We prove that the self-attention at layer \( \ell \) has at most \( (|\Sigma| + 1)^{2\ell} - 1 \) different possible output vectors, by induction on \( \ell \).

Base case \( (\ell = 0) \): Since there are no position embeddings, the embedding at position \( i \) is determined entirely by \( w_i \), so there are at most \( |\Sigma| \) possible activation vectors.

Inductive step \( (\ell > 0) \): Assume that the output of the layer \( \ell \) has at most \( (|\Sigma| + 1)^{2\ell} - 1 \) possible activation vectors, and consider layer \( (\ell + 1) \):

- The attention output at position \( i \) depends only on \( H^{(\ell)}[i] \) (because of the residual connection) and \( H^{(\ell)}[j_i] \) (where \( j_i \) is the position that \( i \) attends to). So the number of possible output activation vectors is at most
  \[
  ((|\Sigma| + 1)^{2\ell} - 1)(|\Sigma| + 1)^{2(\ell + 1)} \leq (|\Sigma| + 1)^{2\ell} - 1 \left((|\Sigma| + 1)^{2\ell} + 1\right)
  = \left((|\Sigma| + 1)^{2\ell}\right)^2 - 1
  = (|\Sigma| + 1)^{2^{\ell+1}} - 1.
  \]
- Because the FFNN and layernorms operate position-wise, they also have at most \( (|\Sigma| + 1)^{2^{\ell+1}} - 1 \) possible activation vectors.
Therefore, the number of possible output vectors from a self-attention or FFNN at layer $\ell$ is at most $(|\Sigma| + 1)^{2^{\ell}} - 1$.

Additionally, the attention score from position $i$ to $j$ depends only on $H^{(\ell)}[i]$ and $H^{(\ell)}[j]$, so there are at most $((|\Sigma| + 1)^{2^\ell})^2 \leq (|\Sigma| + 1)^{2^{\ell+1}}$ possible attention scores.

Then $F$ is the union over all layers of the possible attention scores and components of the possible activation vectors.

Although it’s not the most efficient way to do it, we can represent an activation value $X(w)$ as a set of FO formulas $(\phi_u)_{u \in F}$, such that $w \models \phi_u$ iff $X(w) = v$. Angluin et al. (2023) show how to do this with $O(\log |F|)$ formulas instead.

**Definition 6.7.** Let $k \geq 0$. A function $X : \Sigma^* \times \mathbb{N}^k \rightarrow F$ is definable by FO formulas $(\chi_a(x_1, \ldots, x_k))_{a \in F}$ if for all $w \in \Sigma^*$ and $v \in F$, we have $X(w, i_1, \ldots, i_k) = v$ iff $w \models \chi_a(i_1, \ldots, i_k)$.

**Lemma 6.8.** Let $F \subseteq \mathbb{R}$ be a finite set. If $X : \Sigma^* \rightarrow F$ is definable by FO formulas $\phi_a$, and $f : F \rightarrow F$, then there are FO formulas $\phi'_a$ that define $f(X(w))$.

**Proof.**
\begin{equation}
\phi'_a = \bigvee_{w \models f(u)} \phi_u.
\end{equation}

Hopefully, it is easy to see how the above result generalizes to the cases where $X$ maps to a sequence of activations and/or $f$ is a function of two or more arguments.

**Proof of Theorem 6.5.** For every activation value $H^{(\ell)}[i, k](w)$, we will construct a formula $\phi_{\ell, k, a}(x)$ such that $w \models \phi_{\ell, k, a}(i)$ iff $H^{(\ell)}[i, k](w) = v$. We do this by induction on $\ell$.

Case $\ell = 0$:
\begin{equation}
\phi_{0, k, a}(x) = \bigvee_{a \in E, \text{emb}(a)[k] = v} Q_a(x).
\end{equation}

Case $\ell > 0$: By Lemma 6.8, there are formulas $\text{score}_{\ell, a}(i, j)$ that define $s^{(\ell)}[i, j]$, the attention score at layer $\ell$ from position $i$ to $j$. Furthermore, there is a formula $\text{score}_{\ell, \leq}(i, j_1, j_2)$ that holds iff $s^{(\ell)}[i, j_1] \leq s^{(\ell)}[i, j_2]$ and formulas $\text{value}_{\ell, a}(j)$ that define $W^y(X[j])$, the value at position $j$. Then we can define a formula that tests whether position $x$ attends to position $y$:

\begin{equation}
\text{weight}_{\ell}(x, y) = y < x \land \forall y' \left[(y' < y \rightarrow \text{score}_{\ell, \leq}(x, y', y)) \land (y < y' < x \rightarrow \neg \text{score}_{\ell, \leq}(x, y, y'))\right].
\end{equation}
Chapter 6. Transformers 63

and formulas that define $Y[x][k]$, the $k$-th component of the attention output at position $x$:

$$\text{att}_{i,k,o}(x) = \begin{cases} \exists y \left[ \text{weight}_{i}(x, y) \land \text{value}_{i,k,o}(y) \right] & \text{if } v \neq 0 \\ \exists y \left[ \text{weight}_{i}(x, y) \land \text{value}_{i,k,o}(y) \right] \lor \neg \exists y [y < x] & \text{if } v = 0. \end{cases} \quad (6.40)$$

By Lemma 6.8 again, there are formulas $\phi_{i,k,o}$ that define the output of the FFNN.

Finally, the activation vector at the last position is defined by $s_{G} = \phi_{i,k,o}$.

To go in the other direction, we use the fact that PTL is equivalent to FO (Kamp, 1968) and do an easier conversion from PTL.

**Theorem 6.9.** For any formula $\phi$ of PTL, there is a transformer encoder with right-most hard attention and strict future masking that recognizes the same language that $\phi$ defines.

The proof will be by induction on the structure of $\phi$, so we need the following lemma for combining the translations of sister subformulas.

**Lemma 6.10 (Parallel composition).** Given transformer encoders without layer normalization

$$tfr_{1} : \Sigma^{*} \xrightarrow{\phi} (\mathbb{R}^{d_{1}})^{*}$$

$$tfr_{2} : \Sigma^{*} \xrightarrow{\phi} (\mathbb{R}^{d_{2}})^{*}$$

there is a transformer

$$tfr_{1} \oplus tfr_{2} : \Sigma^{*} \xrightarrow{\phi} (\mathbb{R}^{d_{1}+d_{2}})^{*}$$

such that for all strings $w \in \Sigma^{*}$,

$$(tfr_{1} \oplus tfr_{2})(w[1] \cdots w[n]) = \begin{bmatrix} tfr_{1}(w)[1] \\ \vdots \\ tfr_{1}(w)[n] \end{bmatrix} \cdot \begin{bmatrix} tfr_{2}(w)[1] \\ \vdots \\ tfr_{2}(w)[n] \end{bmatrix}. \quad (6.41)$$

**Proof.** Let $d_{1}$ and $d_{2}$ be the width of $tfr_{1}$ and $tfr_{2}$, and let $d = d_{1} + d_{2}$. If one of $tfr_{1}$ and $tfr_{2}$ has fewer layers than the other, add trivial layers until they have the same number of layers $L$.

The new transformer has embedding layer

$$(tfr_{1} \oplus tfr_{2}).\text{emb}(w[1] \cdots w[n]) = \begin{bmatrix} tfr_{1}.\text{emb}(w)[1] \\ \vdots \\ tfr_{1}.\text{emb}(w)[n] \end{bmatrix} \cdot \begin{bmatrix} tfr_{2}.\text{emb}(w)[1] \\ \vdots \\ tfr_{2}.\text{emb}(w)[n] \end{bmatrix}.$$
For each layer $\ell \in [L]$, let $f_1 = \text{tfr}_\ell.\text{layer}_\ell$ and $f_2 = \text{tfr}_{\ell+1}.\text{layer}_{\ell+1}$. Widen $f_1$ into a layer $f'_1$ with width $d$ as follows:

$$f'_1.\text{W}^Q = \begin{bmatrix} f_1.\text{W}^Q & 0 \end{bmatrix} \quad f'_1.\text{W}^K = \begin{bmatrix} f_1.\text{W}^K & 0 \end{bmatrix}$$

$$f'_1.\text{W}^V = \begin{bmatrix} f_1.\text{W}^V & 0 & 0 \end{bmatrix}$$

$$f'_1.\text{lin}_1.\text{W} = \begin{bmatrix} f_1.\text{lin}_1.\text{W} & 0 \\ 0 & 0 \end{bmatrix} \quad f'_1.\text{lin}_1.\text{b} = \begin{bmatrix} f_1.\text{lin}_1.\text{b} \end{bmatrix}$$

$$f'_1.\text{lin}_2.\text{W} = \begin{bmatrix} f_1.\text{lin}_2.\text{W} & 0 \\ 0 & 0 \end{bmatrix} \quad f'_1.\text{lin}_2.\text{b} = \begin{bmatrix} f_1.\text{lin}_2.\text{b} \end{bmatrix}$$

Similarly, widen $f_2$ into a layer $f'_2$ with width $d$, but using the bottom half of the activation vectors:

$$f'_2.\text{W}^Q = \begin{bmatrix} 0 & f_2.\text{W}^Q \end{bmatrix} \quad f'_2.\text{W}^K = \begin{bmatrix} 0 & f_2.\text{W}^K \end{bmatrix}$$

$$f'_2.\text{W}^V = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

$$f'_2.\text{lin}_1.\text{W} = \begin{bmatrix} 0 & 0 \\ f_2.\text{lin}_1.\text{W} & 0 \end{bmatrix} \quad f'_2.\text{lin}_1.\text{b} = \begin{bmatrix} 0 \end{bmatrix}$$

$$f'_2.\text{lin}_2.\text{W} = \begin{bmatrix} 0 & 0 \\ f_2.\text{lin}_2.\text{W} & 0 \end{bmatrix} \quad f'_2.\text{lin}_2.\text{b} = \begin{bmatrix} 0 \end{bmatrix}$$

Then stack $f'_1$ on top of $f'_2$, or the other way around. (If we had multi-head attention, we could have combined $f_1$ and $f_2$ into a single layer.)

Proof of Theorem 6.9. For any PTL formula $\phi$, there is a transformer $\text{tfr}_\phi$ and an index $k$ such that $\text{tfr}_\phi(F) \cup 8 = I \cup F \cup 8 \models \phi$. We show this by induction on the structure of $\phi$.

Base case $\phi = Q_a$ for some $a \in \Sigma$: Then $\text{tfr}_\phi$ is just an embedding function

$$tfr_\phi(w)[i] = \llbracket w[i] = a \rrbracket . \quad (6.42)$$

If $\phi = \neg \phi_1$: By the induction hypothesis, there is a transformer $\text{tfr}_{\phi_1}$ simulating $\phi_1$. For simplicity, we write activation vectors with just the components we’re interested in:

$$tfr_{\phi_1}(w)[i] = \llbracket w[i] \models \phi_1 \rrbracket . \quad (6.43)$$

We can add a trivial self-attention layer and, by Theorem 3.5, a FFNN to simulate $\phi = \neg \phi_1$:

$$fnn_{\neg}(tfr_{\phi_1}(w))[i] = \begin{bmatrix} 0 \\ \llbracket w[i] \models \neg \phi_1 \rrbracket \end{bmatrix} . \quad (6.44)$$
If $\phi = \phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$: By the induction hypothesis, there are transformers $tfr_1$ and $tfr_2$ simulating $\phi_1$ and $\phi_2$, respectively. Combine these using Lemma 6.10, add a trivial self-attention layer and, by Theorem 3.5, a FFNN to simulate $\phi$.

If $\phi = \phi_1$ since $\phi_2$: By the induction hypothesis, there are transformers $tfr_1$ and $tfr_2$ simulating $\phi_1$ and $\phi_2$, respectively. Combine these using Lemma 6.10. For simplicity, we write activation vectors as

$$H_i(w) = \begin{bmatrix} \mathbb{I}[w, i \models \phi_1] \\ \mathbb{I}[w, i \models \phi_2] \\ 0 \end{bmatrix}$$

(6.45)

suppressing all other coordinates.

Then we add a self-attention layer:

$$q^{(i)} = [1] \quad k^{(j)} = [I - (w, j \models \phi_1) \vee (w, j \models \phi_2)]$$

$$v^{(j)} = \begin{bmatrix} 0 \\ 0 \\ w, j \models \phi_2 \end{bmatrix}$$

(6.46)

(6.47)

For any position $i$, the position $j_i$ that receives attention is the rightmost one left of $i$ that either satisfies $\phi_2$ (in which case $\phi_1$ must be satisfied from $j_i$ to $i$ exclusive) or does not satisfy $\phi_1$ (in which case $\phi_2$ must not be satisfied from $j_i$ to $i$ exclusive). These two possibilities are pictured below:

Then $i$ satisfies ($\phi_1$ since $\phi_2$) if and only if $j_i$ satisfies $\phi_2$. So the value tests for whether $\phi_2$ is satisfied.

At the leftmost position ($i = 1$), ($\phi_1$ since $\phi_2$) is false, so it’s correct that the attention outputs the zero vector.

Since this construction uses only position-independent queries (0 or 1), a perhaps surprising consequence is that every transformer encoder with rightmost hard attention and strict future masking is equivalent to one that uses only position-independent queries.
6.2.3 Encoders with soft attention

In this section, we consider transformers that don’t have the added power of intermediate steps, and don’t have the restriction of unique-hard attention. This seems to be the most difficult case to pin down. One key issue is that while unique-hard and average-hard attention only produce rational numbers, soft attention produces real numbers. So far, attempts to obtain upper bounds on the expressivity of soft-attention transformers involve limiting the precision of the numbers involved.

Actual computers, of course, use floating-point numbers with a constant (that is, independent of \( n \)) bits. But Merrill and Sabharwal (2023a) argue that in \( O(1) \) precision, attention cannot attend uniformly to a string of length \( n \), because for large enough \( n \), the attention weights \((a)\) would all round down to zero. Instead, they use \( O(\log n) \) bits of precision.

**Theorem 6.11** (Merrill and Sabharwal, 2023a). For any \( O(\log n) \)-precision transformer encoder \( T \) that recognizes a language \( L \), there is a formula of \( \text{FOM}[\text{BIT}] \) that defines \( L \).

Merrill and Sabharwal (2023a) use floating-point numbers, of the form \( m \cdot 2^e \), where the mantissa \( m \) has \( O(\log n) \) bits including a sign bit, and the exponent \( e \) has \( O(\log n) \) bits including a sign bit. In theoretical papers, I think it’s more common to use a fixed-point representation, in which \( e \) is constant. With fixed-point numbers, it appears to be possible to improve the result to \( O(n) \) bits of precision. We will consider both possibilities.

**Proof.** Merrill and Sabharwal (2023a)’s proof converted \( T \) to a family of threshold circuits, but we show how to go straight to \( \text{FOM}[\text{BIT}] \).

Transformers only use a handful of operations: addition, multiplication, division, \( \max \), \( \exp \), and iterated addition. It suffices to show that these operations can be defined in \( \text{FO}[\text{BIT}] \) on \( O(\log n) \)-bit fixed- or floating-point numbers.

**Addition** and **multiplication**, already defined on integers in Theorem 5.19, are generalized to \( c \log n \) bit integers (where \( c > 1 \)) by Schweikardt (2005, Theorem 3.4bd). Then fixed-point addition and multiplication can be defined using the following facts:

\[
(m_1 \cdot 2^{e_1}) + (m_2 \cdot 2^{e_2}) = (m_1 + m_2) \cdot 2^{e_1} \quad (6.48)
\]

\[
(m_1 \cdot 2^{e_1}) \cdot (m_2 \cdot 2^{e_2}) = (m_1 m_2) \cdot 2^{e_1+e_2} \quad (6.49)
\]

Similarly for floating-point:

\[
(m_1 \cdot 2^{e_1}) + (m_2 \cdot 2^{e_2}) = \begin{cases} 
(2^{e_1} + 2^{e_2}) & e_1 \geq e_2 \\
(m_1 \cdot 2^{e_1-e_2} + m_2) \cdot 2^{e_2} & e_1 < e_2
\end{cases} \quad (6.50)
\]

\[
(m_1 \cdot 2^{e_1}) \cdot (m_2 \cdot 2^{e_2}) = m_1 m_2 \cdot 2^{e_1+e_2} \quad (6.51)
\]
In all of the above, to get mantissas to be integers with the right number of bits, some rounding may be necessary. Division can be defined in terms of multiplication.

Iterated addition on fixed-point numbers reduces to iterated addition on integers:

$$\sum_{i=1}^{n} (m_i \cdot 2^e) = \left( \sum_{i=1}^{n} m_i \right) \cdot 2^e. \tag{6.52}$$

The sum of the $m_i$ is only a mild extension of Theorem 5.26. After counting the bits in each column, we end up with $c \log n$ numbers with $\log n$ bits each. We sum each block of $\log n$ numbers and then add the resulting $c$ block-sums.

On floating-point numbers, iterated addition is more difficult:

$$\sum_{i=1}^{n} (m_i \cdot 2^e_i) = \left( \sum_{i=1}^{n} (m_i \cdot 2^{e_i}) \right) \cdot 2^e \tag{6.53}$$

where $(*)$ is rounded off to the nearest integer. The problem is that if some of the $m_i$ are negative, the sum could end up much smaller than the largest summand. For example, suppose mantissas have 50 bits, and we want to compute

$$1 \cdot 2^0 + -1 \cdot 2^0 + 1 \cdot 2^{-100} = 1 \cdot 2^{-100}.$$

If we choose $e$ to be the maximum of the $e_i$, then $1 \cdot 2^{-100}$ would round off to 0, giving a sum of 0. (This is known as catastrophic cancellation.) Instead, to make the sum exact, Merrill and Sabharwal (2023b) choose $e$ to be the minimum of the $e_i$, which makes each $(*)$ into a $O(\text{poly}(n))$-bit integer. Iterated addition of these so-called long integers is still possible in FOM[BIT] (Barrington and Maciel, 2000, Lecture 7). But if we had started with $O(n)$ bits, we would at this point have an exponential number of bits.

For the exponential function ($\exp x$), first observe that

$$\exp x = \exp_z((x/\log 2)) \exp_z(x/\log 2 - [x/\log 2]) \tag{6.55}$$

$$= \exp_z((x/\log 2)) \exp_z(x - [x/\log 2] \log 2). \tag{6.56}$$

The first factor can be computed by shifting. The second factor can be approximated by a Taylor series (Merrill, p.c.; Hesse et al., 2002, Corollary 6.5):

$$\exp r = \sum_{i=0}^{\infty} \frac{1}{i!} r^i = \sum_{i=0}^{k} \frac{1}{i!} r^i + R_k. \tag{6.57}$$
where the Lagrange remainder term $R_k$ is, for some $z$ in $(0, r)$,

$$R_k = \frac{\exp z}{(k+1)!} \leq \frac{\exp r}{(k+1)!} < \frac{2}{2^{k+1}} r^{k+1} < \frac{1}{2^k}. \quad (6.58)$$

Letting $k = c \log n + 1$ ensures that the approximation is good to $c \log n$ bits.

So we compute Eq. (6.57) minus the remainder term $R_k$. Each term is an iterated product of $O(\log n)$ numbers, which can be expressed in $\text{FO}[\text{BIT}]$ (Hesse et al., 2002, Theorem 5.1), and the summation of $c \log n + 2$ terms can also be expressed in $\text{FO}[\text{BIT}]$. \qed
Bibliography


Barrington, David Mix and Alexis Maciel (2000). Advanced course on computational complexity. CMI-PCMI Undergraduate Program.


Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou (2022). Chain-of-thought prompt-
