XLNet: Generalized Autoregressive Pretraining for Language Understanding[1]

June 21, 2019

[1] *XLNet: Generalized Autoregressive Pretraining for Language Understanding* by Yang et al.

What problems[3] BERT has?

[2] That's why we also predict non-mask tokens during pretraining.
[3] besides obesity

What problems[3] BERT has?

1. It predicts MASK tokens independently, i.e. $p(\tilde{\boldsymbol{x}}|\hat{\boldsymbol{x}}) = \prod_{i=1}^{T} p(\tilde{x}_i|\hat{\boldsymbol{x}})$, where $\tilde{\boldsymbol{x}}, \hat{\boldsymbol{x}}$ are masked and unmasked subsequences of $\boldsymbol{x}$. It's a big deal, because in reality:

$$p(\text{New}, \text{York}|\hat{\boldsymbol{x}}) = p(\text{New}|\hat{\boldsymbol{x}}) \cdot p(\text{York}|\text{New}, \hat{\boldsymbol{x}}) \neq p(\text{New}|\hat{\boldsymbol{x}}) \cdot p(\text{York}|\hat{\boldsymbol{x}})$$

[2] That's why we also predict non-mask tokens during pretraining.

[3] besides obesity

What problems[3] BERT has?

1. It predicts MASK tokens independently, i.e. $p(\tilde{\boldsymbol{x}}|\hat{\boldsymbol{x}}) = \prod_{i=1}^{T} p(\tilde{x}_i|\hat{\boldsymbol{x}})$, where $\tilde{\boldsymbol{x}}, \hat{\boldsymbol{x}}$ are masked and unmasked subsequences of $\boldsymbol{x}$. It's a big deal, because in reality:

   $$p(\text{New}, \text{York}|\hat{\boldsymbol{x}}) = p(\text{New}|\hat{\boldsymbol{x}}) \cdot p(\text{York}|\text{New}, \hat{\boldsymbol{x}}) \neq p(\text{New}|\hat{\boldsymbol{x}}) \cdot p(\text{York}|\hat{\boldsymbol{x}})$$

2. In test time we do not have MASK tokens, which is train-test discrepancy![2]

[2] That's why we also predict non-mask tokens during pretraining.

[3] besides obesity

What problems[3] BERT has?

1. It predicts MASK tokens independently, i.e. $p(\tilde{\boldsymbol{x}}|\hat{\boldsymbol{x}}) = \prod_{i=1}^{T} p(\tilde{x}_i|\hat{\boldsymbol{x}})$, where $\tilde{\boldsymbol{x}}, \hat{\boldsymbol{x}}$ are masked and unmasked subsequences of $\boldsymbol{x}$. It's a big deal, because in reality:

   $$p(\text{New}, \text{York}|\hat{\boldsymbol{x}}) = p(\text{New}|\hat{\boldsymbol{x}}) \cdot p(\text{York}|\text{New}, \hat{\boldsymbol{x}}) \neq p(\text{New}|\hat{\boldsymbol{x}}) \cdot p(\text{York}|\hat{\boldsymbol{x}})$$

2. In test time we do not have MASK tokens, which is train-test discrepancy![2]
3. Using large contexts is very costly $O(n^2)$.

[2] That's why we also predict non-mask tokens during pretraining.
[3] besides obesity

What problems language models have?

[4] or bidirectionally with simple concatenation, like ELMO

What problems language models have?

- We can train them either forward or backward[4]

[4] or bidirectionally with simple concatenation, like ELMO

What problems language models have?

- We can train them either forward or backward[4]
- $\Longrightarrow$ model is not trained to use the whole context

[4] or bidirectionally with simple concatenation, like ELMO

What problems language models have?

- We can train them either forward or backward[4]
- $\Longrightarrow$ model is not trained to use the whole context
- $\Longrightarrow$ it's bad, because such an information is very useful for some tasks

[4] or bidirectionally with simple concatenation, like ELMO

XLNet to the rescue

XLNet to the rescue

- Let $\mathcal{Z}_T$ be a set of all possible permutations of a sequence of length T (so it has $T!$ elements).

XLNet to the rescue

- Let $\mathcal{Z}_T$ be a set of all possible permutations of a sequence of length T (so it has $T!$ elements).
- Note that we can decompose $p(\boldsymbol{x})$ via chain rule in an arbitrary order:

$$\begin{aligned} p(\boldsymbol{x}) &= p(x_1, x_2, x_3) \\ &= p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \\ &= p(x_2)p(x_3|x_2)p(x_1|x_2, x_3) \\ &= \dots \end{aligned}$$

XLNet to the rescue

- Let $\mathcal{Z}_T$ be a set of all possible permutations of a sequence of length T (so it has $T!$ elements).
- Note that we can decompose $p(\boldsymbol{x})$ via chain rule in an arbitrary order:

$$
\begin{aligned}
p(\boldsymbol{x}) &= p(x_1, x_2, x_3) \\
&= p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \\
&= p(x_2)p(x_3|x_2)p(x_1|x_2, x_3) \\
&= \ldots
\end{aligned}
$$

- Which order is the best for our task? Forward? Backward?

XLNet to the rescue

- Let $\mathcal{Z}_T$ be a set of all possible permutations of a sequence of length T (so it has $T!$ elements).
- Note that we can decompose $p(\boldsymbol{x})$ via chain rule in an arbitrary order:

$$\begin{aligned} p(\boldsymbol{x}) &= p(x_1, x_2, x_3) \\ &= p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \\ &= p(x_2)p(x_3|x_2)p(x_1|x_2, x_3) \\ &= \ldots \end{aligned}$$

- Which order is the best for our task? Forward? Backward?
- All of them:

$$\mathcal{L}_{\text{XLNet}}(\theta) = -\mathop{\mathbb{E}}_{\boldsymbol{z}\sim\mathcal{Z}_T}[\log p_\theta(\boldsymbol{x})] = -\mathop{\mathbb{E}}_{\boldsymbol{z}\sim\mathcal{Z}_T}\left[\sum_{t=1}^{T} \log p_\theta(x_{z_t}|\boldsymbol{x}_{\boldsymbol{z}_{<t}})\right]$$

XLNet to the rescue

- Let $\mathcal{Z}_T$ be a set of all possible permutations of a sequence of length T (so it has $T!$ elements).
- Note that we can decompose $p(\boldsymbol{x})$ via chain rule in an arbitrary order:

$$\begin{aligned} p(\boldsymbol{x}) &= p(x_1, x_2, x_3) \\ &= p(x_1)p(x_2|x_1)p(x_3|x_1, x_2) \\ &= p(x_2)p(x_3|x_2)p(x_1|x_2, x_3) \\ &= \ldots \end{aligned}$$

- Which order is the best for our task? Forward? Backward?
- All of them:

$$\mathcal{L}_{\text{XLNet}}(\theta) = -\mathop{\mathbb{E}}_{\boldsymbol{z}\sim\mathcal{Z}_T}[\log p_\theta(\boldsymbol{x})] = -\mathop{\mathbb{E}}_{\boldsymbol{z}\sim\mathcal{Z}_T}\left[\sum_{t=1}^{T} \log p_\theta(x_{z_t}|\boldsymbol{x}_{\boldsymbol{z}_{<t}})\right]$$

- Is this an upper bound for a true loss? Yes:

$$\mathop{\mathbb{E}}_{\boldsymbol{z}}[\log p(\boldsymbol{x}|\boldsymbol{z})] \leq \log \mathop{\mathbb{E}}_{\boldsymbol{z}}[p(\boldsymbol{x}|\boldsymbol{z})] = \log p(\boldsymbol{x})$$

How does it look like in practice (1/3)?

How does it look like in practice (1/3)?

- It's straight-forward how to decode the sequence in the required order $\boldsymbol{z}$: just use proper masks

How does it look like in practice (1/3)?

- It's straight-forward how to decode the sequence in the required order $\boldsymbol{z}$: just use proper masks

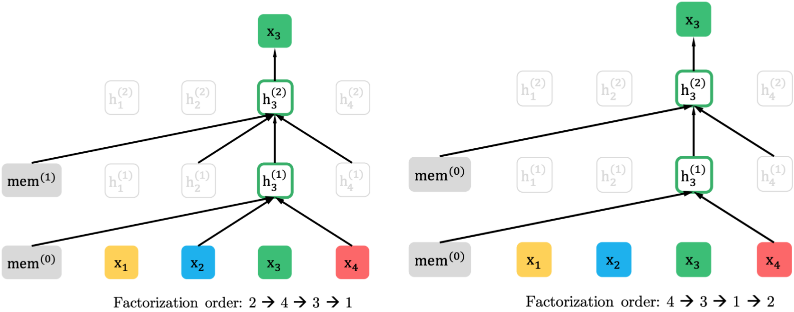

Figure: Example of prediction token x_3 for different permutation orders $\boldsymbol{z}$.

How does it look like in practice (2/3)?

How does it look like in practice (2/3)?

- But we cannot predict “next” token like we do in language models!

How does it look like in practice (2/3)?

- But we cannot predict "next" token like we do in language models!
- Because we have many possible permutations, the same hidden state will be used to predict different tokens:

$$\left.\begin{aligned} \boldsymbol{z}^{(1)} &= ({\color{blue}3, 4, 2}, {\color{red}1, 5}) \\ \boldsymbol{z}^{(2)} &= ({\color{blue}3, 4, 2}, {\color{red}5, 1}) \end{aligned}\right\} \Longrightarrow \begin{cases} p(x_1 = s | x_3, x_4, x_2) = \dfrac{\exp\Big(e(x)^\top h_\theta(\mathbf{x_z}{<}t)\Big)}{\sum_{x'} \exp\Big(e(x')^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\Big)} \\ p(x_5 = s | x_3, x_4, x_2) = \dfrac{\exp\Big(e(x)^\top h_\theta(\mathbf{x_z}{<}t)\Big)}{\sum_{x'} \exp\Big(e(x')^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\Big)} \end{cases}$$

How does it look like in practice (2/3)?

- But we cannot predict "next" token like we do in language models!
- Because we have many possible permutations, the same hidden state will be used to predict different tokens:

$$\left.\begin{matrix} \boldsymbol{z}^{(1)} = (3, 4, 2, 1, 5) \\ \boldsymbol{z}^{(2)} = (3, 4, 2, 5, 1) \end{matrix}\right\} \Longrightarrow \begin{cases} p(x_1 = s | x_3, x_4, x_2) = \dfrac{\exp\left(e(x)^\top h_\theta(\mathbf{x}_{\mathbf{z}} < t)\right)}{\sum_{x'} \exp\left(e(x')^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\right)} \\ p(x_5 = s | x_3, x_4, x_2) = \dfrac{\exp\left(e(x)^\top h_\theta(\mathbf{x}_{\mathbf{z}} < t)\right)}{\sum_{x'} \exp\left(e(x')^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\right)} \end{cases}$$

- Solution?

How does it look like in practice (2/3)?

- But we cannot predict "next" token like we do in language models!
- Because we have many possible permutations, the same hidden state will be used to predict different tokens:

$$\left.\begin{matrix} \boldsymbol{z}^{(1)} = (3, 4, 2, 1, 5) \\ \boldsymbol{z}^{(2)} = (3, 4, 2, 5, 1) \end{matrix}\right\} \Longrightarrow \begin{cases} p(x_1 = s | x_3, x_4, x_2) = \dfrac{\exp\big(e(x)^\top h_\theta(\mathbf{x}_{\mathbf{z}} < t)\big)}{\sum_{x'} \exp\big(e(x')^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\big)} \\ p(x_5 = s | x_3, x_4, x_2) = \dfrac{\exp\big(e(x)^\top h_\theta(\mathbf{x}_{\mathbf{z}} < t)\big)}{\sum_{x'} \exp\big(e(x')^\top h_\theta(\mathbf{x}_{\mathbf{z}<t})\big)} \end{cases}$$

- Solution? Predict token from additional [MASK] embedding.

How does it look like in practice (3/3)?

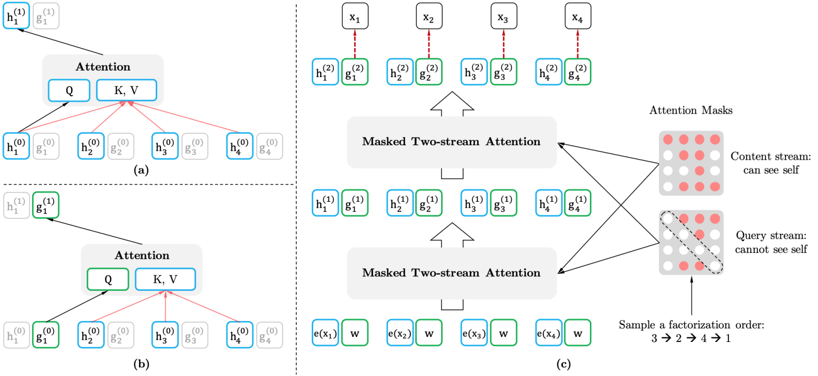

Figure: Two-stream self-attention (copy pasted for intensive hand waving).

Training details

Training details

- For each sentence in a batch take a random permutation $\boldsymbol{z} \in \mathcal{Z}_T$ for decoding.

Training details

- For each sentence in a batch take a random permutation $\boldsymbol{z} \in \mathcal{Z}_T$ for decoding.
- Initialize [MASK] tokens with trained embeddings.

Training details

- For each sentence in a batch take a random permutation $\boldsymbol{z} \in \mathcal{Z}_T$ for decoding.
- Initialize [MASK] tokens with trained embeddings.
- Use forward/backward pass for each half of the batch (explained later).

Training details

- For each sentence in a batch take a random permutation $\boldsymbol{z} \in \mathcal{Z}_T$ for decoding.
- Initialize [MASK] tokens with trained embeddings.
- Use forward/backward pass for each half of the batch (explained later).
- Mask around 15% of tokens (because otherwise it will be hard to train).

Training details

- For each sentence in a batch take a random permutation $\boldsymbol{z} \in \mathcal{Z}_T$ for decoding.
- Initialize [MASK] tokens with trained embeddings.
- Use forward/backward pass for each half of the batch (explained later).
- Mask around 15% of tokens (because otherwise it will be hard to train).
- We use [A, SEP, B, SEP, CLS] like in BERT.

Training details

- For each sentence in a batch take a random permutation $\boldsymbol{z} \in \mathcal{Z}_T$ for decoding.
- Initialize [MASK] tokens with trained embeddings.
- Use forward/backward pass for each half of the batch (explained later).
- Mask around 15% of tokens (because otherwise it will be hard to train).
- We use [A, SEP, B, SEP, CLS] like in BERT.
- We use relative segment encoding: $a_{ij}^{\text{final}} = a_{ij} + (\boldsymbol{q}_i + \boldsymbol{b})^\top \boldsymbol{s}_{ij}$, where $\boldsymbol{q}_i$ is our query, $\boldsymbol{b}$ is a learned head-specific vector, $\boldsymbol{s}_{ij} = \boldsymbol{s}_-$ or $\boldsymbol{s}_+$ if i is in the same context as j or not.

Relative positional encoding

(from Transformer-XL)

Relative positional encoding

(from Transformer-XL)

- Key idea: make information about position be dependent only on relative positions between objects, regardless where they are placed in the sequence
- Why do we need this?

Relative positional encoding

(from Transformer-XL)

- Key idea: make information about position be dependent only on relative positions between objects, regardless where they are placed in the sequence
- Why do we need this? To attend on memory without much trouble (memory usually has the same positional embeddings).

Relative positional encoding

(from Transformer-XL)

- Key idea: make information about position be dependent only on relative positions between objects, regardless where they are placed in the sequence
- Why do we need this? To attend on memory without much trouble (memory usually has the same positional embeddings).
- Imagine we have hidden states $\boldsymbol{e}_i, \boldsymbol{e}_j$. We add positional embeddings to them: $\boldsymbol{h}_i = \boldsymbol{e}_i + \boldsymbol{p}_i$ and $\boldsymbol{h}_j = \boldsymbol{e}_j + \boldsymbol{p}_j$

Relative positional encoding

(from Transformer-XL)

- Key idea: make information about position be dependent only on relative positions between objects, regardless where they are placed in the sequence
- Why do we need this? To attend on memory without much trouble (memory usually has the same positional embeddings).
- Imagine we have hidden states $\boldsymbol{e}_i, \boldsymbol{e}_j$. We add positional embeddings to them: $\boldsymbol{h}_i = \boldsymbol{e}_i + \boldsymbol{p}_i$ and $\boldsymbol{h}_j = \boldsymbol{e}_j + \boldsymbol{p}_j$

Then normal attention of $\boldsymbol{h}_i$ on $\boldsymbol{h}_j$ is computed as

$$\begin{aligned}\mathrm{Attn}(\boldsymbol{h}_i, \boldsymbol{h}_j) &= \langle W_q \boldsymbol{h}_i, W_k \boldsymbol{h}_j \rangle = \langle W_q(\boldsymbol{e}_i + \boldsymbol{p}_i), W_k(\boldsymbol{e}_j + \boldsymbol{p}_j) \rangle \\ &= \boldsymbol{e}_i^\top W_q^\top W_k \boldsymbol{e}_j + \boldsymbol{e}_i^\top W_q^\top W_k \boldsymbol{p}_j + \boldsymbol{p}_i^\top W_q^\top W_k \boldsymbol{e}_j + \boldsymbol{p}_i^\top W_q^\top W_k \boldsymbol{p}_j\end{aligned}$$

Relative positional encoding just changes attention mechanism to:

$$\mathrm{Attn}^{\mathrm{rel}}(\boldsymbol{h}_i, \boldsymbol{h}_j) = \boldsymbol{e}_i^\top W_q^\top W_k^{e} \boldsymbol{e}_j + \boldsymbol{e}_i^\top W_q^\top W_k^{r} \boldsymbol{p}_{i-j} + \boldsymbol{u}^\top W_k^{e} \boldsymbol{e}_j + \boldsymbol{v}^\top W_k^{r} \boldsymbol{p}_{i-j}$$

Adding memory (aka “recurrence mechanism”)

(from Transformer-XL)

- When we have really large sequence, we can process it segment by segment
- When previous segment is processed, put it into cache and attend as embeddings like we do in machine translation

Quiz time!

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used?

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used? — 512

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used? — 512
- How long did the thing trained?

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used? — 512
- How long did the thing trained? — 2.5 days[5]

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used? — 512
- How long did the thing trained? — 2.5 days[5]
- What was the batch size?

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used? — 512
- How long did the thing trained? — 2.5 days[5]
- What was the batch size? — 2048

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used? — 512
- How long did the thing trained? — 2.5 days[5]
- What was the batch size? — 2048
- What optimizer did they use?

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

Quiz time!

- How many TPUs v3 were used? — 512
- How long did the thing trained? — 2.5 days[5]
- What was the batch size? — 2048
- What optimizer did they use? — Adam

[5] And authors said that the model still underfit after 500K steps (but performance on downstream tasks didn't improve)

TODO

I didn't have enough time to copy-paste tables from scores and ablation study, so let's open the paper and see them manually.

Things I didn't get

- Span-based prediction?