Large Memory Layers with Product Keys [1]

Reading Group Slides

Presenter: Zhiping (Patricia) Xiao

[1] **NeurIPS' 19**, Authors: Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, Hervé Jégou

Outline

Introduction

Related Work

Memory Network: [2]

- Memory Networks (ICLR' 15)
- End-To-End Memory Networks (NIPS' 15)
- Learning to Transduce with Unbounded Memory (NIPS' 15)
- Hybrid computing using a neural network with dynamic external memory (Nature' 16)
- Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes (NIPS' 16)

[2] Thanks to Xingjian for helping with organizing the related works.

- MemNN: input feature map I, generalization (update memories by new input) G, output feature map O, response R. Uses $argmax$ over the memories.
- End2end Memory Networks: embedding matrices A for input and C for output, resulting in "key-value" pair, and use $softmax$; can be cast as a traditional RNN.
- Unbounded Memory: neural Stack, neural Queue, neural DeQue.
- DNC: more focus on memory management, controller [3] uses previous inputs.
- Sparse Reads and Writes: Sparse Access Memory (SAM), with r/w constrained to a sparse subset, along with a sparse memory management scheme.

[3] Generates interface parameters and output parameters.

Links

1. The paper is available on ArXiv.
2. Code: `https://github.com/facebookresearch/XLM`
3. A minimalist example: `https://github.com/facebookresearch/XLM/blob/master/PKM-layer.ipynb`
4. An attempt translating into mxnet: `https://github.com/PatriciaXiao/gluon-nlp/blob/master/scripts/bert/model/pkm.py`.

Enjoy! ☺

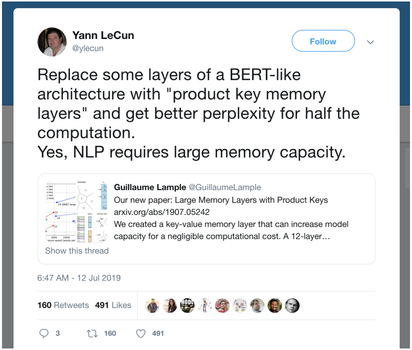

Figure: Mr. LeCun's comments on Twitter.

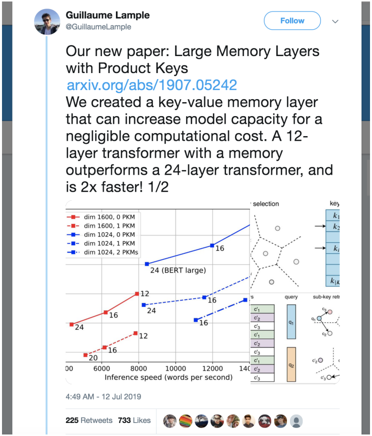

Figure: The first author's brief introduction on Twitter.

The general idea and motivation

- A function $m : \mathbb{R}^d \to \mathbb{R}^n$, acts as a layer in a neural network, offering a large capacity to it.
- Only brings slight computational overhead, in both training and testing; scaling to very large sizes while keeping exact search on the key space.
- Product-key enables fast indexing by reducing the search space dramatically.
- Inspired by the success of BERT and GPT-2, putting the memory layers into transformer.

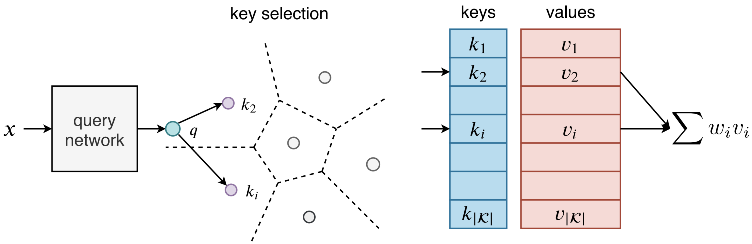

Figure: x, the input, processed through the query network, produces a query vector q, which is compared to all the $|\mathcal{K}|$ keys, and then output the **sparse** weighted sum of over the memories associated with the selected keys. All parameters of the memory are trainable, while only k selected memory slots are updated for each input.

Query network ($q : x \mapsto \mathbb{R}^{d_q}$)

- Typically linear mapping or multi-layer perceptron.
- Adding a batch normalization layer on top of the *query network* helps increasing key coverage during training. [4]
- In the paper's setting, $d_q = 512$, $d > d_q$.

[4] Confirmed by experiments in Section 4.5.

Key assignment

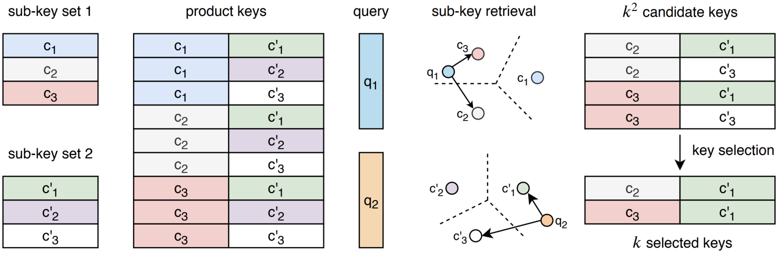

Figure: Split query q into q_1, q_2; search in sub-key set 1 and 2 for the k ($k = 2$ in the illustrated example) nearest neighbors (measured by the inner product) of q_1 and q_2 respectively, thus $k \times k$ keys are implicitly selected. The two subsets induce *product keys* $\mathcal{K}$ ($|\mathcal{K}| = 9$ in this case). The k keys nearest to q in **product keys** are **guaranteed** to be included in this $k \times k$ candidate keys.

$$
\begin{aligned}
\mathcal{I} &= \mathcal{T}_k\Big(q(x)^T k_i\Big) && \text{\# Get k nearest neighbors} \\
w &= \mathrm{Softmax}\Big((q(x)^T k_i)_{i\in\mathcal{I}}\Big) && \text{\# Normalize the top-k scores} \\
m(x) &= \sum_{i\in\mathcal{I}} w_i v_i && \text{\# Aggregate selected values}
\end{aligned}
$$

where $\mathcal{T}_k$ represents the top-k operator, selecting the top-k *indices*.

Product Key

- Having two vector codebooks $\mathcal{C}_1$ and $\mathcal{C}_2$, whose keys are the sub-key sets mentioned before. The sub-keys' dimension is $\frac{d_q}{2}$.
- $\mathcal{C}_1$ and $\mathcal{C}_2$'s outer **product** w.r.t. the vector **concatenation** operator [5] is defined as the *product key set*.

$$\mathcal{K} = \{(c_1, c_2) | c_1 \in \mathcal{C}_1, c_2 \in \mathcal{C}_2\}$$

- Get the nearest k neighbors of q_1 in $\mathcal{C}_1$ as $\mathcal{I}_{\mathcal{C}_1}$, and that of q_2 in $\mathcal{C}_2$ as $\mathcal{I}_{\mathcal{C}_2}$.
- $\{(c_{1,i}, c_{2,j}) | i \in \mathcal{I}_{\mathcal{C}_1}, j \in \mathcal{I}_{\mathcal{C}_2}\}$ is **guaranteed** to include the most similar k keys from $\mathcal{K}$.

[5] a.k.a. the Cartesian product construction.

Proof of "guaranteed" (1/2)

Statement: The candidate set $\mathcal{C} = \{(c_{1,i}, c_{2,j}) | i \in \mathcal{I}_{\mathcal{C}_1}, j \in \mathcal{I}_{\mathcal{C}_2}\}$ is **guaranteed** to include the most similar k keys from $\mathcal{K}$.

Proof: The distance is defined by the inner product between vectors, thus $\forall c_1 \in \mathcal{C}_1, c_2 \in \mathcal{C}_2$,

$$(c_1, c_2)^T q = c_1^T q_1 + c_2^T q_2$$

Assume $\exists (c_1^*, c_2^*) \notin \mathcal{C}$, but is one of the k nearest neighbors of q in $\mathcal{K}$, $\exists (c_1', c_2')$ among the top-k candidates that:

$$c_1'^T q_1 + c_2'^T q_2 \leq (c_1^*)^T q_1 + (c_2^*)^T q_2$$

$$(c_1' - c_1^*)^T q_1 \leq (c_2^* - c_2')^T q_2 \tag{1}$$

Proof of "guaranteed" (2/2)

For convenience, let's denote the set of nearest k neighbors of q_1 in $\mathcal{C}_1$ as $\mathcal{C}'_1$, and similarly $\mathcal{C}'_2$ for q_2 in $\mathcal{C}_2$.
By definition of the k nearest neighbors, $\forall c_1^* \notin \mathcal{C}'_1, \forall c_2^* \notin \mathcal{C}'_2$, and $\forall c'_1 \in \mathcal{C}'_1, \forall c'_2 \in \mathcal{C}'_2$,

$$(c'_1 - c_1^*)^T q_1 \geq 0$$
$$(c_2^* - c'_2)^T q_2 \leq 0$$

From (1) we have:

$$(c'_1 - c_1^*)^T q_1 = 0 = (c_2^* - c'_2)^T q_2$$

As long as $q_1 \neq 0$ and $q_2 \neq 0$, $c'_1 = c_1^*$ and $c'_2 = c_2^*$, which conflicts the assumption that $\exists (c_1^*, c_2^*) \notin \mathcal{C}$.
If $q_1 = 0$ or $q_2 = 0$, the distance will be always 0 thus all keys are the nearest.
The k nearest neighbors of q in $\mathcal{K}$ is guaranteed to be in $\mathcal{C}$.

- Multi-head mechanism makes the model more expressive. Increases the key usage and improves the performance.
- H heads, each has its own query network, and own set of sub-keys, but sharing the same values.
- The final output is simply the sum:

$$m(x) = \sum_{i=1}^{H} m_i(x)$$

- Different from *standard multi-head attention*: the input (query) is not split into H heads, create H queries instead.
- In practice: different heads attend to very different keys, and very different values of the memory.

Time Complexity

Given the memory with keys $\mathcal{K}$ of size $|\mathcal{K}|$, and latent space dimension d_q ($q \in \mathbb{R}^{d_q}$):

- Standard key-value memory layer:
  - Each computation of distance takes d_q operations.
  - $\mathcal{O}(|\mathcal{K}| \times d_q)$
- Product-key memory layer:
  - $|\mathcal{C}_1| = |\mathcal{C}_2| = \sqrt{|\mathcal{K}|}$
  - Finding $k \times k$ candidates from subsets: $2 \times \mathcal{O}(\sqrt{|\mathcal{K}|} \times \frac{d_q}{2}) = \mathcal{O}(\sqrt{|\mathcal{K}|} \times d_q)$
  - Finding the best k keys from $k \times k$ candidates: $\mathcal{O}(k^2 \times d_q)$, since the priority list for $\mathcal{O}(k \log k \times d_q)$ is less compliant with GPU architectures.
- The overall complexity: $\mathcal{O}\Big((\sqrt{|\mathcal{K}|} + k^2) \times d_q\Big) \approx \mathcal{O}(\sqrt{|\mathcal{K}|} \times d_q)$

The layer's role in model

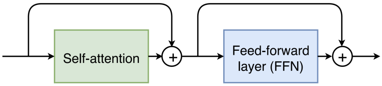

Figure: Typical transformer block with *Feed-Forward Network*. $x = x + FFN(x)$

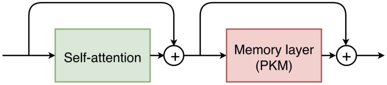

Figure: Modified transformer block with *Product-Key Memory*. $x = x + PKM(x)$

The product-key memory layer is analogous to a sparse FFN layer with a very large hidden state. In practice, they only replaced $N \in \{0, 1, 2\}$ layers' FFN layer in the transformer model.

Experiments

Dataset

- Extracted from the public Common Crawl.
- 40 million English news articles in training set, 5000 in validation and test set each.
- Did not shuffle sentences, allowing the model to learn long range dependencies.

Metrics (1/2)

To measure performance of the model:

- Perplexity on the test set (the smaller the better).

$$
\begin{aligned}
PP(S) &= \mathbf{P}(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\
&= \Big(\prod_{i=1}^{N} \frac{1}{p(w_i | w_1 w_2 \ldots w_{i-1})}\Big)^{-\frac{1}{N}}
\end{aligned}
$$

Metrics (2/2)

To evaluate memory usage:

- Fraction of accessed values: $\#\{z_i \neq 0\}$
  - Expect to use as many keys as possible, around 100%.
- KL (Kullback–Leibler) divergence between distributions of z and the *uniform distribution* $\log(|\mathcal{K}|) + \sum z_i \log(z_i)$
  - Given input x from test set, $w(x)$ is the **sparse** (at most $H \times k$ non-zero entries) of the weights of the keys accessed in the memory.
  - $z'_i = \sum_x w(x)_i$, and $z' \in \mathbb{R}^{|\mathcal{K}|}$
  - $z = \frac{z'}{\|z'\|_1}$
  - Reflects imbalance in the access patterns to the memory, the lower the better.

Main Experiments Results

- Either increasing the *dimension* or increasing *the number of layers* leads to significant perplexity improvements in all models.
- Adding memory is more beneficial than increasing the number of layers.
- In general, the more memory layers added, the better the performance would be.

Ablation Study Results

- Dominant factor for inference time is the number of accessed memory values, which is governed by the number of memory heads h, and the parameter k, **NOT** the memory size.
- Query batch-normalization helps.
- The location to insert the memory layer could be tricky. The worst position is at layer 1, right after the input token embeddings; insert right before the softmax output (at layer 6) is also not a good idea. The best position to insert at is an **intermediate** layer.
- Increasing h and / or k help reach better performance and better memory usage, but there's a trade-off between speed and performance. $h = 4$ and $k = 32$ is good in practice.
- Better than *flat keys* (standard keys) from all aspects.

Thank You

Q & A