Notes: "Large Memory Layers with Product Keys"

Sun

June 1, 2020

1 Notes:

1. This paper presents a computationally efficient (though not space-efficient) method to easily increase the capacity of a given network
2. Increasing the number of params by up to (and even above) 1 billion, yet not adding a significant cost at inference or training
3. They find they can double the inference speed while also gaining performance

1.1 Medium level:

The idea behind them is quite simple: instead of iteratively nonlinearly transforming a given representation, instead, input this representation into a key-value memory layer, and your new representation is some sum of "memories" associated with each key.

There are some interesting implications here: namely, that if a network with enough memory capacity can accurately represent the input in a way that a given query can be mapped to the right key, you have a sort of weak quantization: the output of your key is a sum from a given (deterministic) set of vectors.

I believe this can lead to huge increases in performance when it comes to the model seeing "familiar" content: assuming the memory layers are trained well, the model should be able to leverage very similar samples from previous training iterations better than simply internalizing the knowledge within "normal" parameters (linear layers).

The authors apply this method to transformers, however, it's not limited to them: anything you can put in the format of a query, key, and value, works.

A basic algorithm for this would be:

1. For a given query, select the top k nearest neighbors,
2. Normalize the scores of these neighbors using softmax,
3. Take the weighted sum of the memories, weighted given by these scores

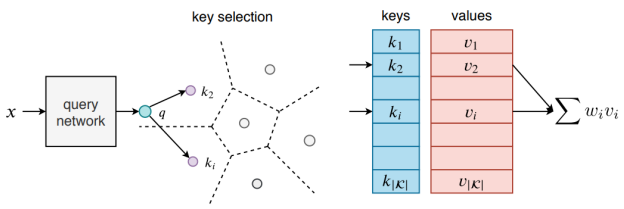

Figure 1: Overview of key-value memory layer. The input x is passed through a *query* network that produces a query vector q, which is compared to all keys. The output is the sparse weighted sum over all the memories associates with the selected keys. For a small number of keys, $\mathcal{K}$, this is extremely fast.

1.2 Low level:

1.2.1 Memory design:

High-low-level overview: The overall structure of the proposed layer is covered well in figures (1) and (2). Abstracted less, a given layer is composed of three parts: A query network, a key selection module, and a value lookup table. It first computes the query that is compared to the product keys. For each product key, it computes a score, and selects the top k product keys. These top k are then weighted by the normalized score (softmax). All the parameters of the memory are trainable, yet only k memory slots are updated each input. The sparse selection and parameter updates are where the computational efficiency lie.

Query generation: The function $q : x \to q(x) \in \mathbb{R}^{d_q}$, referred to as the query network, maps the d-dimensional input to a latent space of dimensionallity d_q. Typically, q is a linear mapping or a multi-layer ANN that reduces the dimensionality from $d = 1024$ to $d_q = 512$. The keys themselves are randomly initialized uniformly across this space.

Key assignment and weighting: Let $\mathcal{T}_k$ denote the top-k operator. Given a set of keys $\mathcal{K} = \{k_1, ..., k_{|\mathcal{K}|}\}$, composed of $|\mathcal{K}|$ d_q dimensional vectors, and an input x, we select the top k keys maximizing the inner product (dot product) of the keys with the query $q(x)$:

1. $\mathcal{I} = \mathcal{T}_k\big(q(x)^T k_i\big)$
2. $w = \text{Softmax}\big((q(x)^T k_i)_{i\in\mathcal{I}}\big)$
3. $m(x) = \sum_{i\in\mathcal{I}} w_i v_i$

where v_i is the given memory for a key.

Operations (2) and (3) only depend on the top-k indices, however, (1) is an exhaustive search. To make this more computationally efficient, they split a given query into *sub*-querys, each with half the original dimension. The codebook is also initialized in 2 different sections, the sub-queries are mapped to keys in both sections, creating "candidate keys" (figure 2), before the true top-k are selected. This trick reduces the number of operations by $\sim 10^3$ at $|\mathcal{K}| = 1024^2$.

The memory layers are optimized via traditional gradient descent.

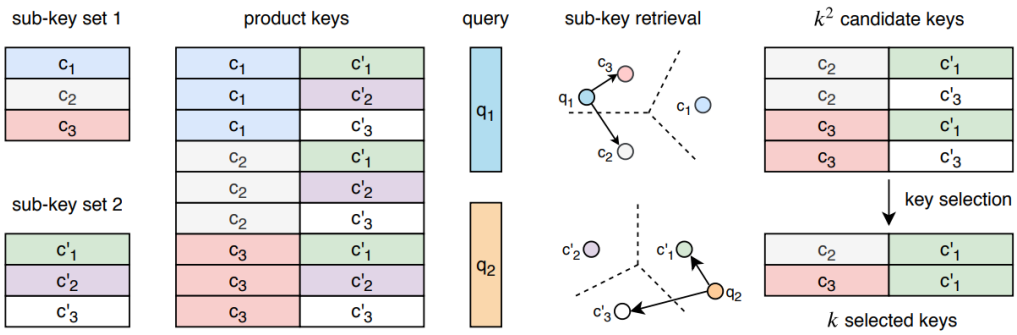

Figure 2: **Illustration of the product keys.** We define two discrete subsets of keys (sub-key set 1 and sub-key set 2). They induce a much larger set of keys, which are never made explicit (product keys). Given a query, we split it into two sub-queries (q_1 and q_2). Selecting the k closest keys ($k = 2$ in the figure) in each subset implicitly selects $k \times k$ keys. The k keys maximizing the inner product with the query are guaranteed to belong to this subset, on which the search can be done efficiently.

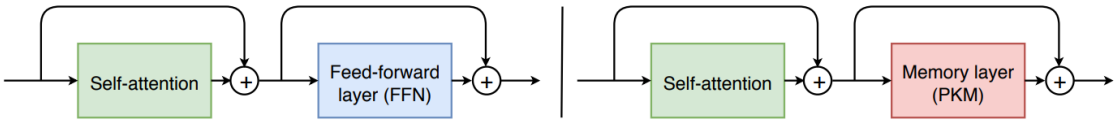

Figure 3: **Left:** A typical transformer block is composed by a self-attention layer followed by an FFN layer (a two layer network). **Right:** In our system, we replace the FFN layer with a product key memory layer, which is analogous to a sparse FFN layer with a very large hidden state. In practice, we only replace the FFN layer in N layers, where typically $N \in \{0, 1, 2\}$.

1.2.2 Uses:

The main use of this algorithm is when, by god, you want to be able to overfit. That may seem like a bad use, but for many huge datasets (mostly NLP), you actually run into problems where models are unable to overfit well due to a lack of capacity; ergo, the models most likely also are underfitting normally.

These huge memory layers let the model have more capacity, yet don't let them overfit extremely easily in normal training. Empirically, they note that even with huge memory layer sizes, on large datasets, overfitting is not noticed.

This method is sadly only usable with huge amounts of RAM available, however, in that case, it shines.

1.3 Conclusion:

If you need the ability to overfit, regardless of dataset size, look to this technique. It increases model capacity a huge amount, and assuming you have the memory available, also speeds up both inference and training, in addition to performance.