Memory Transformer Networks

Jonas Metzger
Stanford University
metzgerj@stanford.edu

CS224N Custom Project with Prof. Manning

Abstract

We propose a novel neural architecture, the Memory Transformer Network. While the conventional Transformer encoder cannot feasibly process documents of arbitrary size due to the associated quadratic increase in computational costs, the Memory Transformer does not suffer from this limitation. This is achieved by a recurrent design, in which the model considers only chunks of the document at a time, while building up a compact representation of the complete document. The purpose of this architecture is to extend the recent superiority of transformers over previous recurrent architectures on small-scale NLP tasks such as SQuAD to tasks that involve reference documents of arbitrary size. The proposed architecture is applied to a large scale-version of the SQuAD task, in which it is shown to significantly outperform other baselines. Finally, we will extrapolate from the success of the Memory Transformer and propose a novel sparse self-attention mechanism, which achieves linearly scaling computational costs.

1 Introduction

Transformer networks, proposed by Vaswani et al. (2017), have quickly gained popularity in natural language processing relative to previous recurrent neural architectures. They are attractive from a statistical perspective as their use of self-attention greatly facilitates the learning of longer-range dependencies between words, compared to LSTMs or other RNNs for example. Computationally, they are attractive due to their increased parallelizability relative to previous recurrent architectures. In particular, they achieve state-of-the-art performance on many NLP tasks, when unsupervised pre-training as a language model is combined with consecutive task-specific fine-tuning, as demonstrated by Devlin et al. (2018)'s pretrained bidirectional transformer encoder model BERT.

However, the standard transformer architecture is clearly not suited for the processing of larger documents: the computational complexity of the self-attention mechanism scales quadratically with the input sequence length, as it involves calculating multiple dot products for every pair of words in the input sequence. As a consequence, in order to make use of a transformer on tasks that require the processing of longer documents, one would have to apply it to short, fixed-length sub-sequences of the original document, piece by piece. This obviously limits the length of dependencies that can be picked up by the model. An additional problem of the transformer encoder in this context is that it produces word-level embeddings for every word in the input document, i.e. entities or concepts that are referred to multiple times in a source document receive multiple embeddings - and words that are rather unimportant are not sorted out. This makes it significantly harder to fine tune a pretrained transformer such as BERT to process larger documents: A large part of what it means to understand a document consists of understanding which concepts in the given document are important, and to condense the information about a given concept that is scattered across the document into one representation for this concept. A model as described previously would have to learn most of this during the fine-tuning stage, which limits its potential performance drastically and requires significant amounts of memory.

In order to resolve these issues, we propose a novel neural architecture, the Memory Transformer. It is a recurrent extension of the original transformer, which successively builds up a knowledge representation by processing the documents in chunks. This representation consists of the transformed embeddings of the most important words that have previously been processed. All words in the currently considered chunk of the document can attend to these important previous words, and vice versa. The proposed memory mechanism can thus be regarded as a form of conditional computation.

We will apply the novel architecture to a large-scale version of the SQuAD (v2.0) task and show that it outperforms two different baselines, one without a memory and one with a naive implementation of a memory mechanism.

Finally, extrapolating from the memory transformer's success in learning a form of conditional computation of sparse attention, we will propose a novel *sparse* self-attention layer, which can directly replace existing self-attention layers in current transformer models. In contrast to the current self-attention layers however, its computational cost scales only linearly in document length to its use of conditional computation when calculating attention scores, while still permitting training via back-propagation due to sufficient differentiability.

2 Approach

2.1 Model Architecture

The memory transformer builds on the architecture of the original transformer encoder in Vaswani et al. (2017). In particular, the model mainly consists of sub-parts $M_\theta^n(x)$, which can be regarded as a composition of n transformer layers T_{θ_l}, identical up to their parameters θ_l, i.e.:

$$M_\theta^n(x) = (T_{\theta_n} \circ ... \circ T_{\theta_1})(x) \tag{1}$$

where $\circ$ denotes function composition. Every transformer layer T_θ consists of a self attention layer[1] S_θ and a standard point-wise fully-connected network F_θ with one hidden layer, both wrapped in individual sub-connection layers C. The model can thus be written as

$$T_\theta(x) = (C(F_\theta) \circ C(S_\theta))(x). \tag{2}$$

A sub-connection layer C contains a standard $LayerNorm$, a residual connection and a $Dropout$ layer, applied as follows:

$$C(F)(x) = LayerNorm(x + Dropout(F(x))) \tag{3}$$

Note that, for ease of notation, we omitted the dependency of $LayerNorm$ on its trainable parameters, which are not shared between different sub-connection layers.

Departing from the original architecture, we introduce a memory mechanism to the transformer, which effectively permits words in the currently considered chunk of the document x_{in} to sparsely attend to a matrix x_{mem} containing the embeddings of the m most important words that were read previously. In addition to these embeddings, the model also stored a vector x_{imp} containing a scalar-valued importance score for every word in memory, which will be key to achieving a trainable sparse attention to previous words. We will come back to this later.

First, x_{in} is passed through an embedding layer $E(x_{in})$, which is identical to the one initially proposed by Vaswani et al. (ibid.), i.e. it is the sum of their sine-based positional embedding and learned word-vectors. Note that the positional embeddings are independent across chunks, i.e. the first word in every chunk receives the same positional embedding. Subsequently, both the embeddings in memory x_{mem} and the embeddings of the chunk are independently fed trough N_1 individual transformer layers which do not share parameters. The embeddings for the memory are then transformed by another point-wise fully-connected layer F_{θ_f} which this time is not bypassed by a residual connection, after which they are concatenated with the embeddings of the current chunk. They are then jointly

[1] Identical to the one proposed in Vaswani et al. (2017). In fact, we partly built on the pytorch implementation from "The Annotated Transformer" at http://nlp.seas.harvard.edu/2018/04/03/attention.html.

fed through N_2 transformer layers, allowing memory and current input to attend to each other. This yields embeddings for both the words in memory and the words in the current chunk, denoted by e:

$$e = M_{\theta_{all}}^{N_2}\Big(\big(F_{\theta_f}(M_{\theta_{mem}}^{N_1}(x_{mem})),\ M_{\theta_{in}}^{N_1}(E(x_{in}))\big)\Big) \tag{4}$$

These embeddings can subsequently be fed into some task-specific layer to produce the desired predictions. What remains to be specified is the mechanism by which the memory is updated. The update will simply be performed by selecting the m most important words among those in the current chunk and in memory. The general idea is sketched in Figure 1. To do so, we introduce a vector V_I of the same dimension as our embeddings and a scalar b_I, which linearly transform any embedding in e onto the real line, yielding an importance score for every word in memory and in the current chunk. We will then keep only the embeddings and the importance scores of the m words with the largest importance scores. Denoting this operation by keep_m_max$(\cdot, \cdot)$, we obtain the embeddings and importance scores of the new memory via:

$$(x'_{mem},\ x'_{imp}) = \text{keep_m_max}(e,\ V_I^T e + b_I) \tag{5}$$

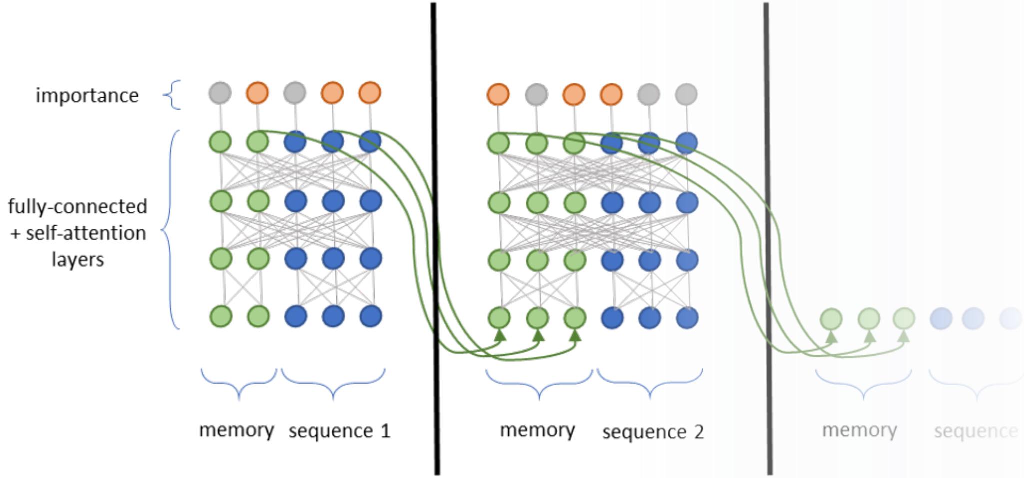

Figure 1: Sketch of general memory recurrence.

This is obviously not a differentiable step. We employ a simple trick to nevertheless allow the model to learn via back-prop which words to assign a high importance score to: We simply **add the importance scores to the attention scores within the attention layers**. More precisely, for any attention head in any self-attention layer in $M_{\theta_{all}}^{N_2}$, i.e. after the concatenation, we do the following: For a given query[2], we the obtain the attention scores for all other words by calculating the dot product with the key of those words. Before applying the soft-max to get the attention distribution however, we simply add the importance score x_{imp} (saved from the previous iteration) of a given word in memory to the key of that word. Otherwise, the self-attention mechanism proceeds as familiar. This will incentivize the model to increase the importance score for words that are worth attending to, and importantly, decrease the importance scores for words that are less important. The keep_m_max operation then simply selects the most important words according to this definition, and the bias term b_I ensures that adding the importance scores does not sub-optimally weight words in memory relative to words in the current chunk. We omitted this dependence of e on x_{imp} earlier for ease of exposition.

We show that this approach trains well and outperforms two baselines on a large scale QA task. In the last section, we will use this as the motivation to propose a novel, sparse self-attention layer based on the same principle, with computational costs that scale only linearly in document length.

[2]We use the terms queries, keys and values just like Vaswani et al. (2017).

2.2 Experiments

2.2.1 Large SQuAD Objective

We compare the performance of the novel architecture on a modified version of the SQuAD task, designed to test the large-scale question answering capabilities of the new architecture. While the original SQuAD task is posed as a set of questions about short paragraphs extracted from Wikipedia articles, we will increase the scale of the problem by directly concatenating a fixed number of successive paragraphs coming from the same article into one reference document. Due to memory limitations, we currently only considered reference documents consisting of two concatenated paragraphs. We skipped examples where this lead to reference documents of greater length than 850 tokens. Many examples in this task come close to this threshold, and are thus significantly longer than the examples considered by Devlin et al. (2018) for example. To further emulate a large-scale problem, we restrict the model capacity relative to the size of reference document by requiring the memory transformer and any baseline to only look at document chunks of 48 tokens at a time. We consider three models, the memory transformer, a naive-memory baseline and a no-memory baseline, which are described in the next subsection. For the memory transformer and the naive-memory baseline, we set the size of the memory to $m = 16$ tokens.

To facilitate a baseline without a memory module, the SQuAD task is interpreted as a token-level classification task. Every token is either part of an answer, or not. In addition to the parameters outlined in the previous section, we additionally learn a linear transformation mapping the token embeddings e onto the real line. This is number is fed into a sigmoid, yielding a distribution over the two classes. The model is trained via weighted log-likelihood, where we weight the two classes with the inverse of their frequency in the training data. This choice was made to give equal weight to type I and type II error, and to ensure the activation of the sigmoid is centered around 0.5, which yields better gradient signals. In our setting, skipping examples longer than 850 tokens and using a batch size of 16 without masking padding tokens, the share of answer tokens was 0.0058.

For the memory transformer and the naive-memory baseline, this task is performed in two phases. First, during the reading phase, the models read the whole reference document chunk by chunk, building up their memory. Next, during the inference phase, with their memory fixed to that obtained from the reading phase, the models jointly observe a SQuAD question and a given chunk of the reference document. The embeddings for the tokens in the current chunk are fed into the classifier head and contribute to the average log-likelihood across all document chunks. We maximize the log-likelihood across all questions and reference documents. The no-memory baseline simply skips the reading phase.

2.3 Baselines

In addition to the memory transformer, we consider two baselines. One simply discards the memory module all together, and is thus a simple transformer encoder with $N_1 + N_2$ transformer layers. The other baseline, the naive-memory baseline, implements a different kind of memory module. This memory does not require any importance scores, as the elements in the memory do not directly come from the embeddings of previous tokens. Rather, we initialize a sequence `<MEM>` tokens of length m, and embed them as we would embed any other sequence of tokens. Additionally, in all models, we left two dimensions of the positional embeddings "empty" (i.e. constant at zero). The naive-memory baseline sets the embedding vector in these two dimensions to one for the memory tokens, in order to be able to identify the memory as such. The embeddings in e corresponding to the `<MEM>` tokens are simply used as the memory in the next time step.

3 Experiments

3.1 Training

We consider the same hyperparameters for all three models. We used a pytorch implementation of the BERT tokenizer[3], a batch size of 16, maximum reference document length of 850 and a chunk size of 48 tokens. The model hyperparameters are an embedding size of 128 dimensions, all hidden

[3]https://github.com/huggingface/pytorch-pretrained-BERT

layers in the point-wise fully-connected layers have size 512, the number of attention heads in every self-attention layer is 8. All dropout probabilities are 0.1, $N_1 = 2$ and $N_2 = 4$, and $m = 16$. We used the Adam optimizer with a learning rate of $1e-5$ and a weight decay of 0.01.

The small chunk size of 48 was chosen to mimic a task in which reference documents are large relative to the feasible model size, which obviously puts the no-memory baseline at a disadvantage. While larger m was found to increase the performance of both models with memory, the memory-transformer drastically improves in performance relative to the naive-memory baseline with growing m, which suggests an inductive bias of the memory-transformer towards a distributed knowledge representation. All other hyperparameters were chosen to optimize the absolute performance of the no-memory baseline subject to constraints on memory and compute.

3.2 Results and Analysis

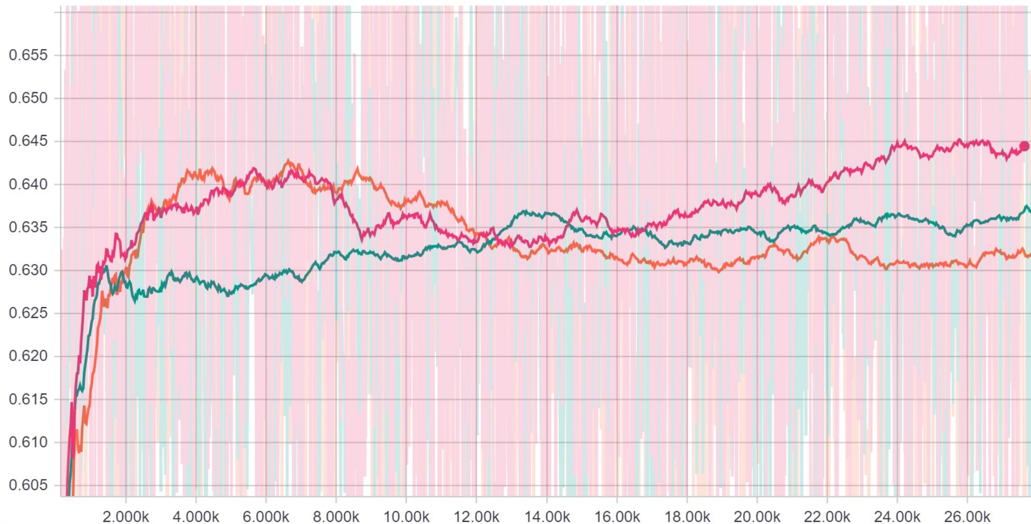

Figure 2: Pseudo F1-score (smoothed). pink: memory transformer; orange: naive-memory baseline; green: no-memory baseline

We track multiple performance metrics. The two most obvious are the training and test loss. Additionally, over the test set, we calculate the average predicted probabilities for "answer", where the averages are taken over all "answer" tokens and all "no answer" tokens respectively. These can be interpreted as a kind of true positive rate (tpr or recall) and the false positive rate (fpr). In terms of those, the precision would be defined as $\text{tpr}/(\text{tpr} + r * \text{fpr})$, where r equals the ratio of the number of the negatives relative to the positives. To reflect the fact that due to our weighted log-likelihood, the classes contribute as if we had equal shares of positives and negatives, we will instead consider a "pseudo precision" with $r = 1$. Combining recall and "pseudo precision" as if the latter were the actual precision, we get a "pseudo F1-score", which we track during training and plot in Figure 2.

We observe that the memory transformer clearly achieves the best pseudo F1-score, being the only model that clearly exhibits a trend promising continuing improvement over further training steps. In comparison, the baselines both appear to have plateaued in performance much earlier. This is in line with our general impressions that the relative performance of the memory transformer architecture starts to outperform the baselines particularly for larger batch sizes, longer training and large memory. In light of the fact that the application in the present paper only represents a very small-scale proxy for a large-scale task, this is particularly promising.

In terms of the test loss depicted in Figure 3, all methods achieve similar performance, and suggest further improvement over longer training. The train loss is plotted in Figure 6.

Further insight into the performances of the models can be obtained from considering the average predicted probability for "answer" for the "answer" tokens (Figure 4) and the "no answer" tokens (Figure 5) respectively. From the shape of these graphs, it appears that the relative performance of the models on the pseudo F1-score metric is mostly driven by their relative ability to detect

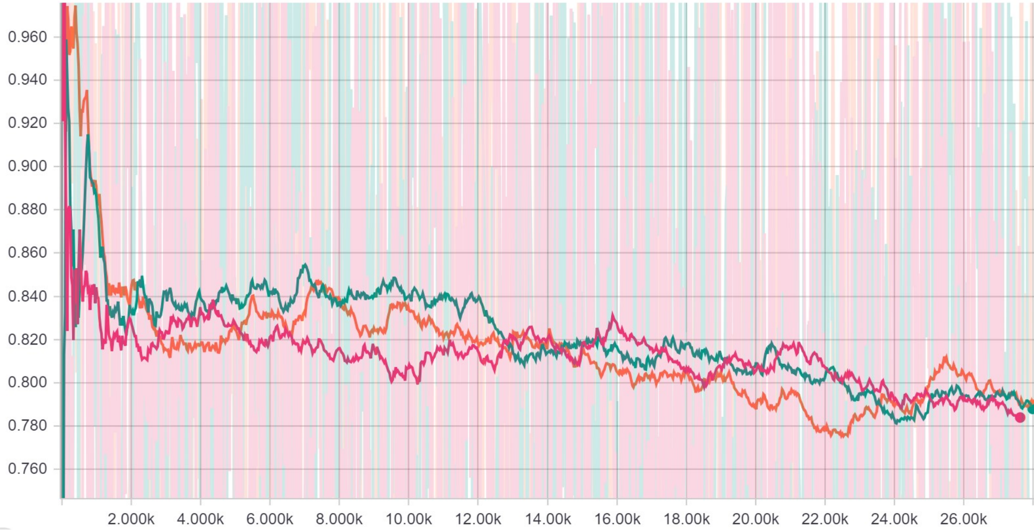

Figure 3: Test loss (smoothed). pink: memory transformer; orange: naive-memory baseline; green: no-memory baseline

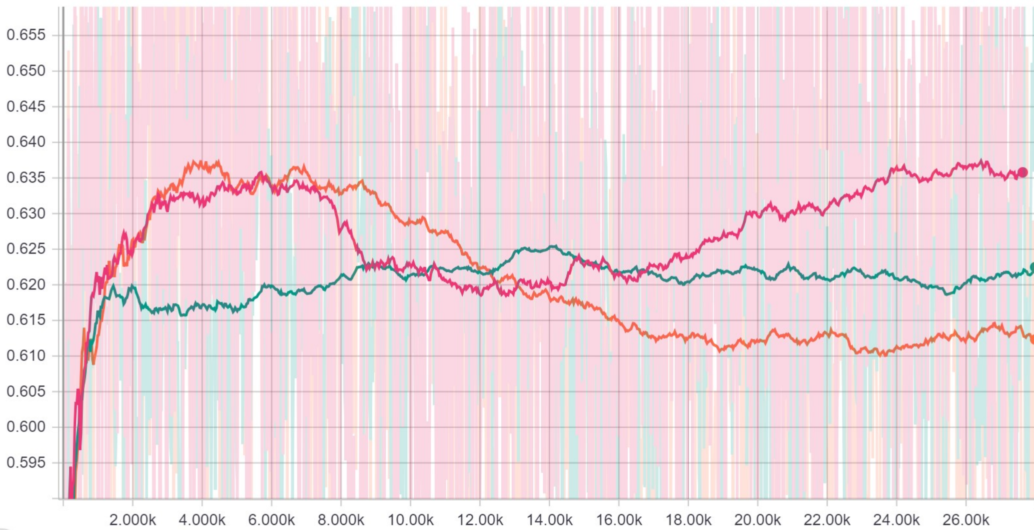

Figure 4: Average ”answer” probability for ”answer” tokens (smoothed). pink: memory transformer; orange: naive-memory baseline; green: no-memory baseline

”answer” tokens, rather than their relative performance on the ”no answer” tokens. The differences in performance in the former case are also larger (around 0.2 probability mass) than in the latter case (around to 0.1 probability mass). If we had the goal of engineering a solution which produces sharp answer predictions maximizing an exact match (EM) score, it seems reasonable to expect that doing well in detecting ”answer” tokens is more important than confidently ruling out ”no answer” tokens, which lends credibility to the ranking of the methods according to the pseudo F1-score in this case.

4 (Full) Attention Might Be More Than You Need

The successful training of the memory transformer suggests the possibility of a novel form of sparse (self-)attention mechanism, which avoids the quadratic scaling of computational costs. We will briefly propose it here and explore its performance in future work. Consider a single self-attention head, with an input sequence of L word embeddings. These word embeddings are linearly transformed into query vectors q_i and key vectors k_i $\forall i$. Conventional self-attention layers would go on

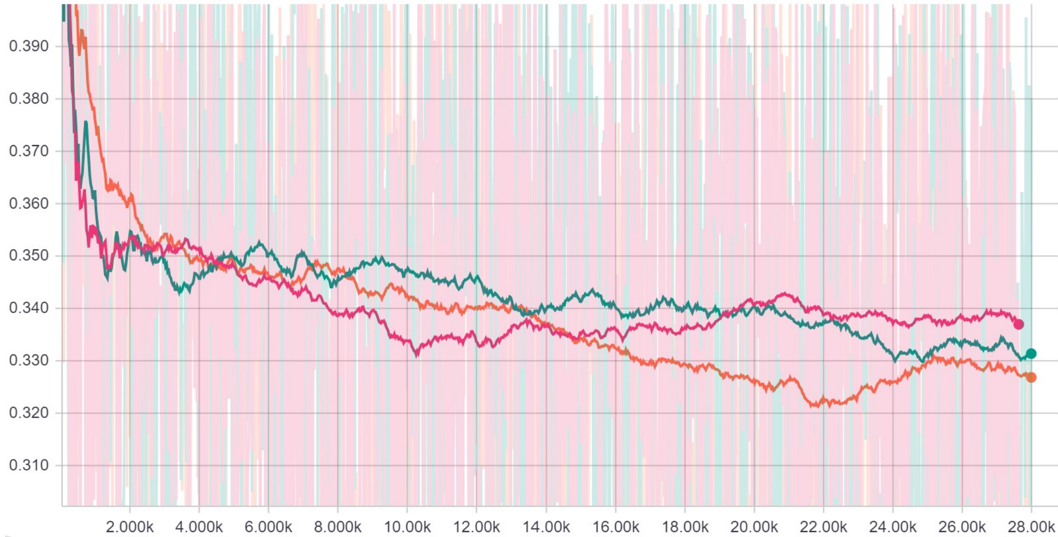

Figure 5: Average ”answer” probability for ”no answer” tokens (smoothed). pink: memory transformer; orange: naive-memory baseline; green: no-memory baseline

to compare all pairs (q_i, k_j) to assess which words j a word i should attend to, thereby incurring computational costs which scale quadratically in L. However, an individual self-attention head in a multi-head attention setting is likely to be fairly specialized on certain kinds of connections due to its low complexity. It is therefore likely that it would suffice to let any word in this head only attend to a small number of n candidate words, which have a high plausibility of being attended to by any other word in this head given their embedding. Building on the success of the memory transformer, we propose the following approach in this setting: In addition to the keys and queries, calculate plausibility scores p_i for every word i via a linear mapping from the word embeddings onto the real line, using trainable weights that are specific to this self-attention head. This additional computation can be offset by reducing the dimension of the vector of values v_i computed by this attention head by one. Let $N(p)$ denote the set of indices of the n words with the largest plausibility scores. Then compute the length n vector of attention weights w_i for a given query q_i as

$$w_i = softmax(s_i), \;\; s_{i,j} = \sum_{j \in N(p+e_i)} K(q_i, k_j) + p_j, \;\; e_i \overset{\text{iid}}{\sim} F \tag{6}$$

where $K(\cdot, \cdot)$ denotes some similarity kernel function, e.g. the scaled dot product as proposed in Vaswani et al. (2017). e_i denotes some random noise component that might be added to aid exploration during training. Multiple such attention heads can be combined into a sparse multi-head attention layer. Holding n fixed, the computational complexity of such a self-attention layer only scales linearly in L by focusing only on high plausibility candidates rather than all words in the source document. This self-attention mechanism naturally gives rise to the *Sparse Transformer Network*. In contrast to this architecture, the memory transformer can be viewed as additionally also budgeting the available memory, in case the full set of reference documents for a large-scale NLP task cannot be jointly loaded onto the GPU.

5 Conclusion

We proposed the Memory Transformer, a neural architecture alleviating the architectural limitations inherent in conventional transformer networks which prevent them from being scaled to large-scale NLP tasks. We examined the performance of the method on a large-scale version of the SQuAD task and achieved promising results. We further outlined how the core idea underlying the memory module, sparse attention, could be employed to obtain a novel, scalable self-attention mechanism, giving rise to the Sparse Transformer. It will be interesting to see how well the memory transformer

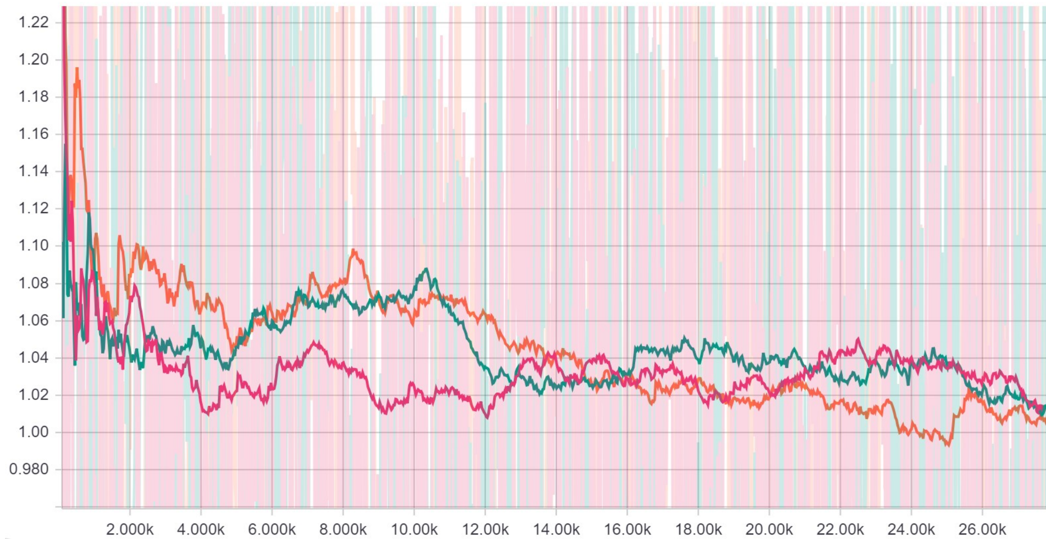

Figure 6: Train loss (smoothed). pink: memory transformer; orange: naive-memory baseline; green: no-memory baseline

networks can be pretrained to make use of their memory module via unsupervised language modeling tasks, such that they start off any task-specific fine-tuning with the ability to extract the most important concepts from a source document.

References

Devlin, Jacob et al. (2018). “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *CoRR* abs/1810.04805. arXiv: `1810.04805`. URL: `http://arxiv.org/abs/1810.04805`.

Vaswani, Ashish et al. (2017). “Attention is all you need”. In: *Advances in Neural Information Processing Systems*, pp. 5998–6008.