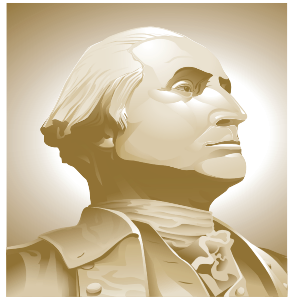
Transformer Based Neural Networks Models

Natural Language Processing
Lecture 10



THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

BERT

BERT

- BERT, which stands for Bidirectional Encoder Representations from Transformers, is a Language Model released by the Google AI Language team at the end of the year 2018.
- It has become the state-of-the-art model for many different Natural Language Undestanding tasks, including sequence and document classification.
- This is best reflected by the fact that one can only see BERT-like models on the GLUE benchmark leaderboard.
- The success of BERT can be explained in part by its novel language modeling approach, but also by the use of the Transformer
- A Neural Network architecture based solely on Attention mechanisms, which was introduced one year prior, replacing Recurrent Neural Networks (RNNs)

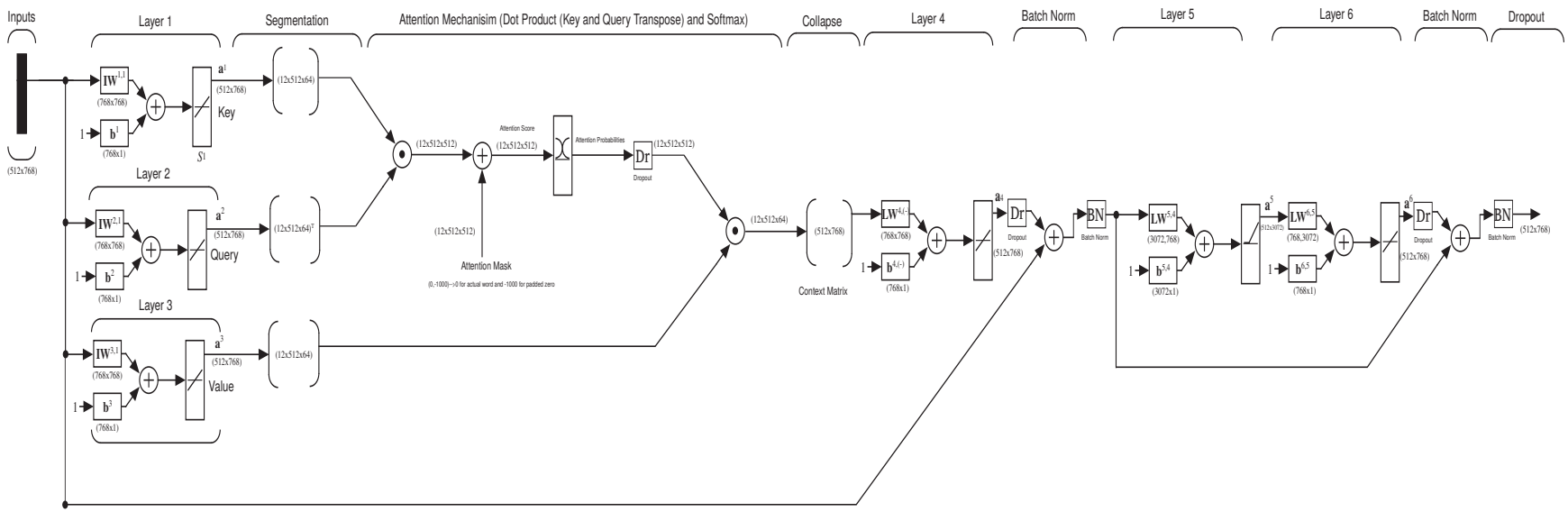
Self Attention

- Self-Attention, the kind of Attention used on the Transformer, is essentially a mechanism that allows a Neural Network to learn representations of some text sequence influenced by all the words on the sequence.
- In a RNN context, this is achieved by using the hidden states of the previous tokens as inputs to the next time step.
- However, as the Transformer is purely feed-forward, it must find some other way of combining all the words together to map any kind of function in an a NLU task.
- It does so with the following equation

$$self\ Attention = Softmax(\frac{QK^T}{\sqrt{d_k}})V$$

- Here, Q, K and V (query, key and value) are matrices which are obtained by taking the dot product of some trainable weight matrices W^Q, W^K and W^V with the embedding matrix of our input sequence X.
- That is, $Q = W^Q X^T$, $K = W^K X^T$ and $V = W^V X^T$.
- Basically, each row on these matrices corresponds to one word, meaning that each word is mapped to three different projections of its embedding space.
- These projections serve as abstractions to compute the self-attention function for each word.
- The dot product between the query for word 1 and all the keys for words $1, 2, \ldots, n$ tells us how "similar" each word is to word 1, a measure that is normalized by the softmax function across all the words.

One layer of base BERT



Multi Head Attention

- The Transformer model goes one step further than simply computing a Self-Attention function, by implementing what is called Multi-Head Attention.
- This is basically a set of L Self-Attention computations, each on different sub-vectors, obtained from the original Q, K and V by breaking them up into L different Q_l, K_l and V_l made up of R/L components each, where R is the 768 embedding dimension and L the number of 12 Attention Heads.
- Although up until this point we have only described the Encoder part of the Transformer, which is actually an Encoder-Decoder architecture.
- Now we proceed to describe BERT's architecture from input to output, and also how it is pre-trained to learn a natural language.

Embedding Layer and First Layer

- First, the actual words in the text are projected into an embedding dimension, which will be explained later in the context of Language Modeling.
- Once we have the embedding representation of each word, we input them into the first layer of BERT.
- This is done in order to avoid paying attention to padded 0s (which are necessary if one wants to do vectorized mini-batching).
- The attention mask is vector made up of 0s for the words we want the model to attend to (the actual words in the sequence), and of very small values (like -10,000) for the padded 0s.
- The sums of the keys and queries dot products with this mask will go into the softmax, making the attention scores for the masked padded 0s become practically 0.

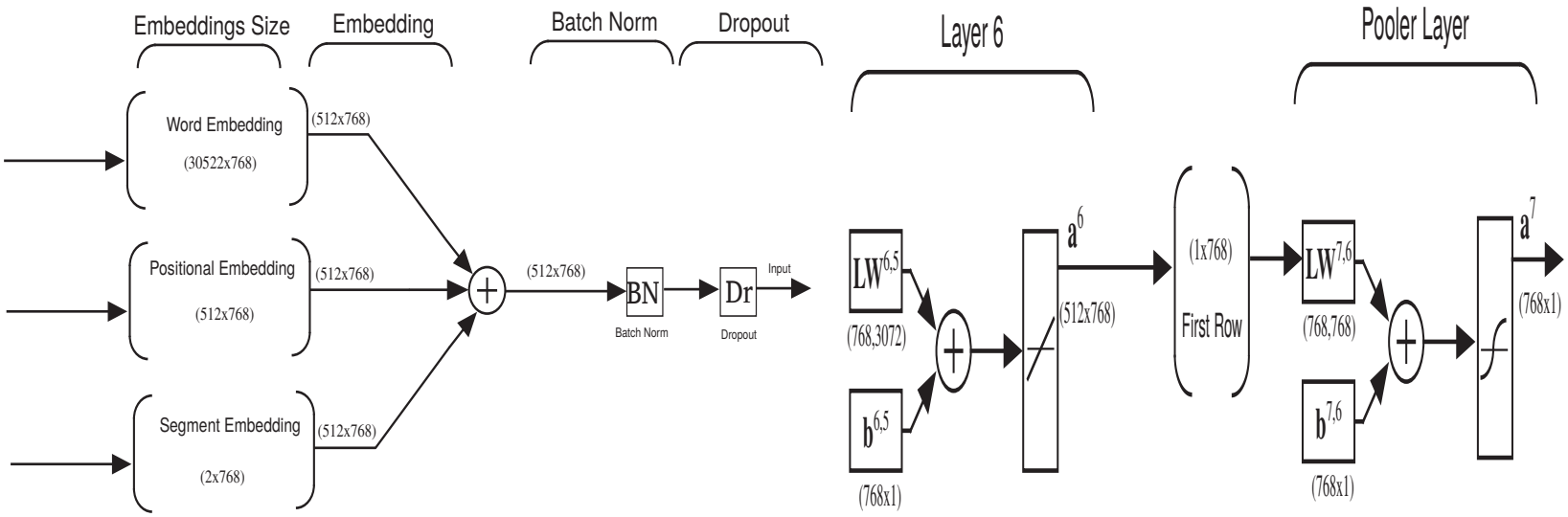
Embedding Layer and First Layer

- The output of this layer goes into a linear layer of size RxR, in order to learn a local linear combination of the Multi-Head Attention output.
- Batch Normalization is performed on the sum of the output of this layer (after a Dropout) and the input to the BERT layer.
- This is fed into yet another linear layer of size RxR', where $R' = 3072$ for the base BERT, followed by a GeLu (Gaussian Error Linear Units) transfer function and another linear layer $(R'xR)$ that maps the higher dimensions back to the embedding dimensions, with also Dropout and BatchNorm.
- This constitutes one BERT Layer, of which the base model has 12.

Layers

- The outputs of the first layer are treated as the hidden embeddings for each word, which the second layer takes as inputs and does the same kind of operations on them.
- Once we have gone through all the layers, the output for the first token (a special token [CLS] that remains the same for all input sequences) is passed onto another linear layer.
- Now that we have described BERT's architecture in detail, we will focus on the other main aspect that makes BERT so successful: BERT is, first and foremost, a Language Model.
- This means that the model is designed to learn useful knowledge about natural language from large amounts of unlabeled text

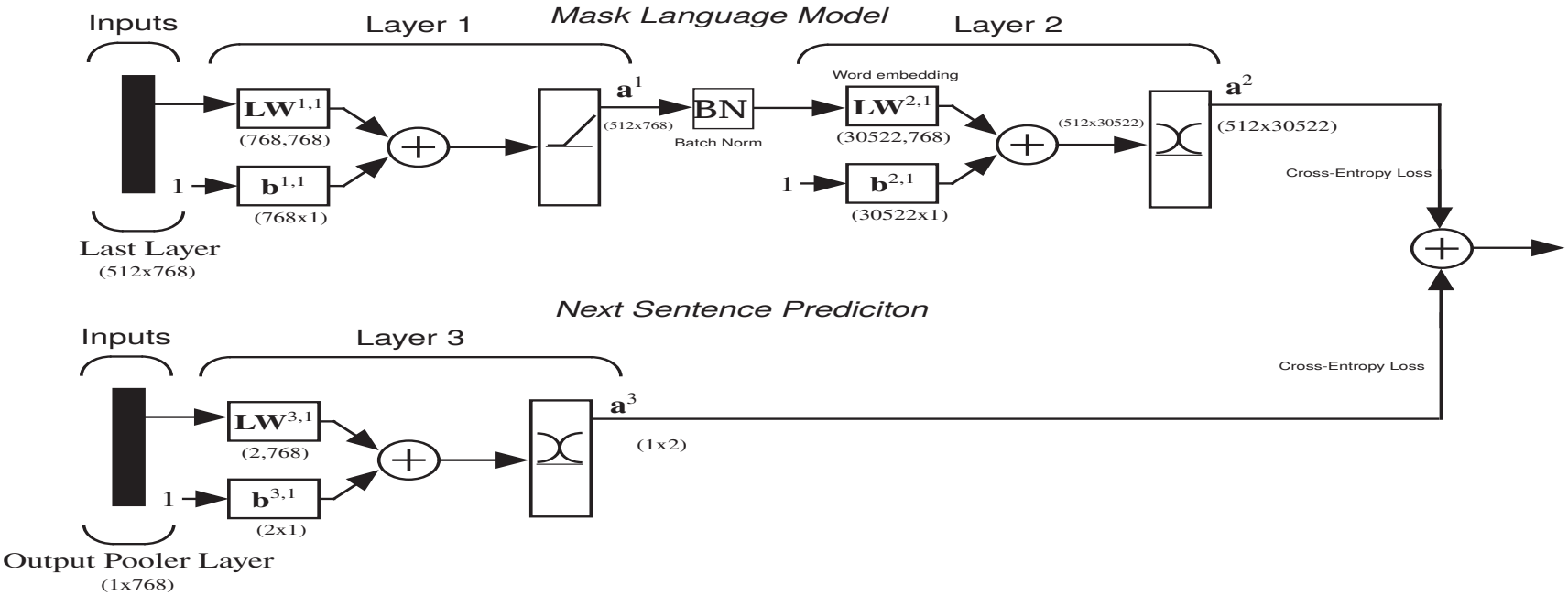
BERT Embedding



(a) Embedding Layer

(b) Pooler Layer

BERT Pre-training



Language Modeling

- The way Language Modeling usually works in a RNN scenario is just using the n previous words as inputs to predict the next word $n + 1$.
- The model cannot take as input the word n+1 or any words after it.
- However, as BERT is a feed-forward architecture that uses attention on all the words in some fixed-length sequence, if nothing is done, the model would be able to attend mainly to the very same word it is trying to predict.
- Thankfully, the attention mechanism can allow to capture both previous and future context, and one can stop the model from attending to the target word by masking it.
- In particular, for each input sequence, 15 % of the tokens are randomly masked, and then the model is trained to predict these tokens.

Masking

- The way this is done is taking the output of BERT, before the pooler, and mapping the vectors corresponding to each word to the vocabulary size with a linear layer.
- whose weights are the same as the ones from the input word embedding layer, although an additional bias is included, and then passing this to a softmax function in order to minimize a Categorical Cross-Entropy performance index.
- which is computed with the predicted labels and the true labels (the ids on the token vocabulary, but only making the masked words contribute to the loss).
- Masking words is really straightforward: just replace them with the special token “[MASK]”.
- This way, the network cannot use information from this word or any other masked words, aside from their position in the text.
- BERT was also pre-trained to predict whether a sentence B follows another sentence A (both randomly sampled from the text 50% of the time, while the rest of the time sentence B is actually the sentence that comes after sentence A).

Positional Embedding

- In addition to the usual word embeddings, positional embeddings are used to give the model information about the position of each word on the sequence
- and due to the next sentence prediction task and also for easy adaptation to downstream tasks such as question-answering, a segment embedding to represent each of the sentences is also utilized.
- The word embeddings used by BERT are WordPiece embeddings, which consist in a tokenization technique in which the words are split into sub-word units.
- This helps handling out-of-vocabulary words while keeping the actual vocabulary size small (30,522 unique word-pieces for BERT uncased).
- The positional embeddings are look-up tables of size $512xR$, which assign a different embedding vector to each token based on its position within the sequence.

XLNET

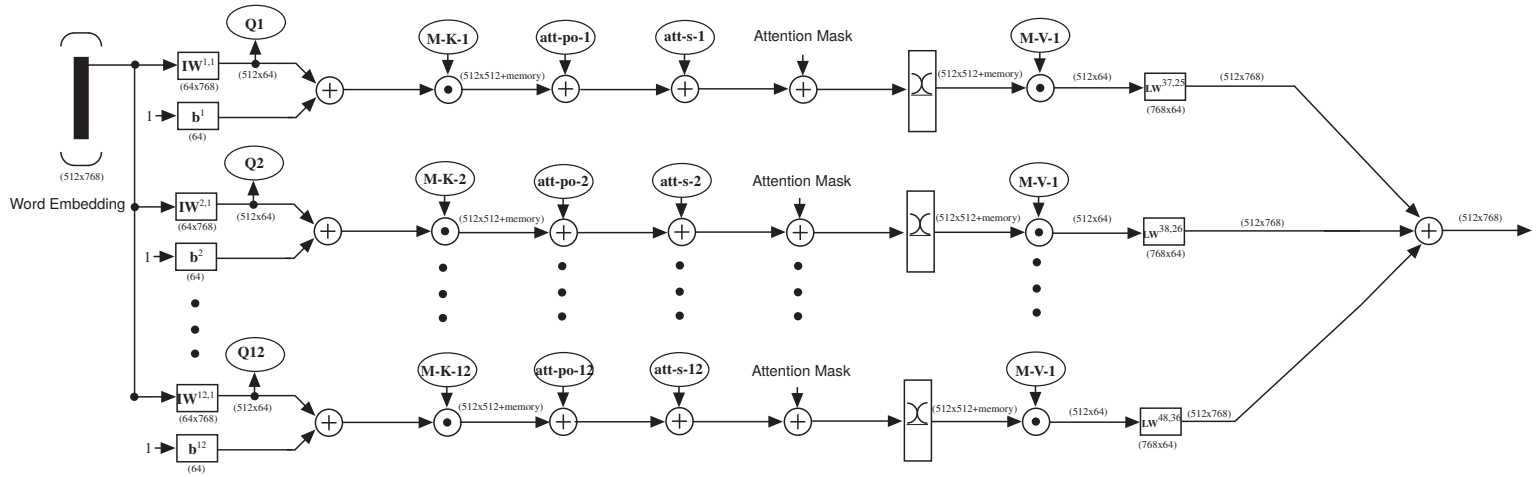
XLNET

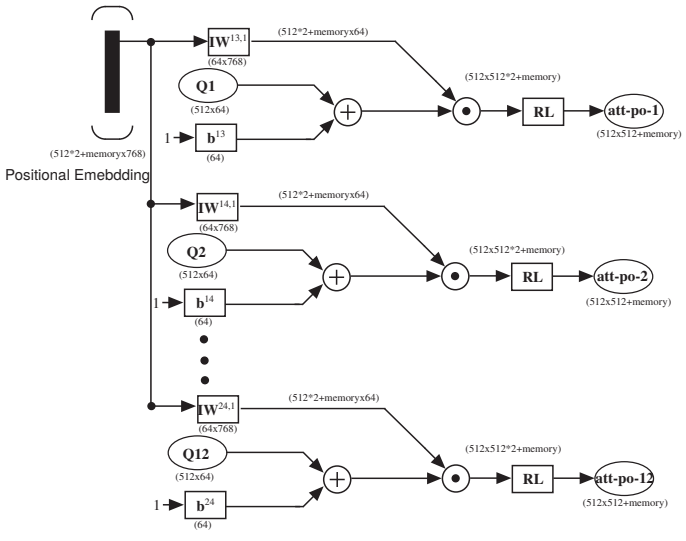
- XLNet is a language model introduced that makes use of the TransformerXL to incorporate information from previous sequences in order to process the current sequence achieving a regressive effect at the sequence level.
- To do so, it employs a relative positional encoding and a permutation language modeling approach. Although BERT and XLNet share a lot of similarities, there are some key differences that need to be explained.
- Firstly, XLNet's Multi-Head Attention's core operation is different than the one implemented in BERT and in the Transformer.
- In this case, instead of just breaking up the original Q, K and V into L different Q_l, K_l and V_l; L linear layers (for each) are used to map the input to the Multi-Head Attention layer into these different Q_l, K_l and V_l, and thus no intermediate Q, K and V are computed.

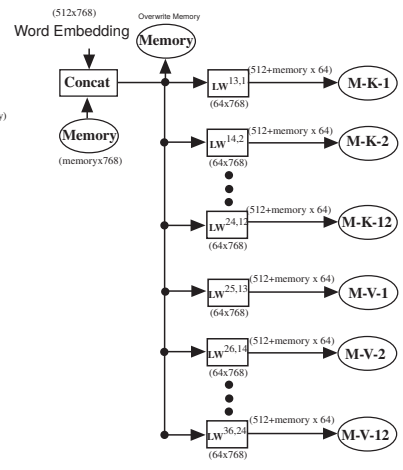
XLNET

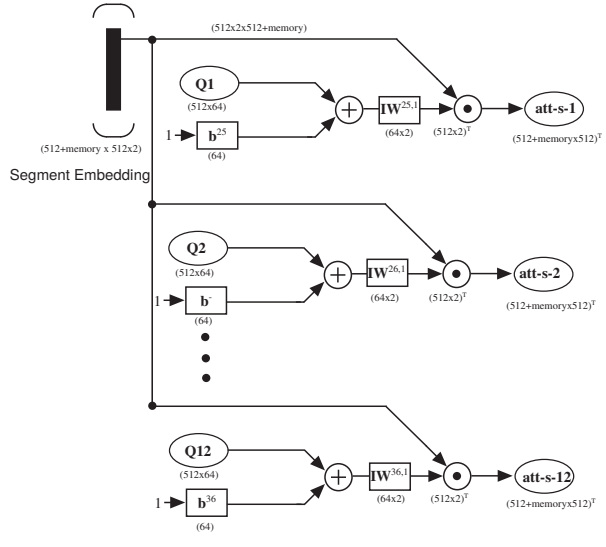
- XLNet is a language model introduced that makes use of the TransformerXL to incorporate information from previous sequences in order to process the current sequence achieving a regressive effect at the sequence level.
- To do so, it employs a relative positional encoding and a permutation language modeling approach. Although BERT and XLNet share a lot of similarities, there are some key differences that need to be explained.
- Firstly, XLNet's Multi-Head Attention's core operation is different than the one implemented in BERT and in the Transformer.
- In this case, instead of just breaking up the original Q, K and V into L different Q_l, K_l and V_l; L linear layers (for each) are used to map the input to the Multi-Head Attention layer into these different Q_l, K_l and V_l, and thus no intermediate Q, K and V are computed.

One Layer XLNET









XLNET Vs BERT

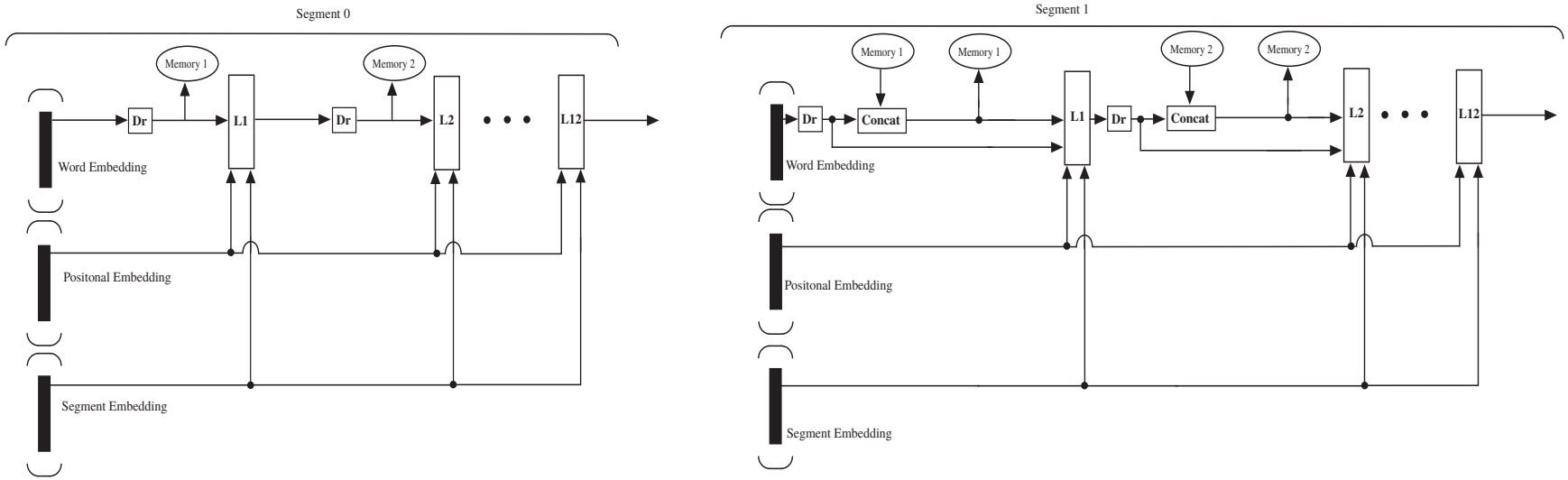
- Secondly, apart from this, XLNet's Attention is different from BERT's in two ways.
- The keys and values (but not the queries) of the current sequence and for each layer depend on the hidden states of the previous sequence/s, based on a memory length hyper-parameter.
- That is, let the hidden state (output) of layer m for the previous sequence be a matrix h^m_{t-1} and then projecting the result using the $W^{K,m+1}_l$ and $W^{V,m+1}_l$ matrices.
- This recurrence mechanism at the sequence level is illustrated.
- The operation just described is only applied to the word embeddings, and not to the sum of the three kinds of embeddings

Multi Head and Equation

$$\text{MultiHead}(X) = \sum_{l=1}^{12} \left[\text{Softmax}\left(\frac{\overbrace{XW_l^Q}^{Q_l} (\overbrace{XW_l^K}^{K_l})^T}{\sqrt{d_k}} \right) \overbrace{XW_l^V}^{V_l} \right] W_l^O$$

$$p_{\text{pos}} = \text{concat}\left[\sin\left(e_{\text{inv-inds}} \otimes p_{\text{inds}} \right), \cos\left(e_{\text{inv-inds}} \otimes p_{\text{inds}} \right) \right]$$

12 Layer XLNET

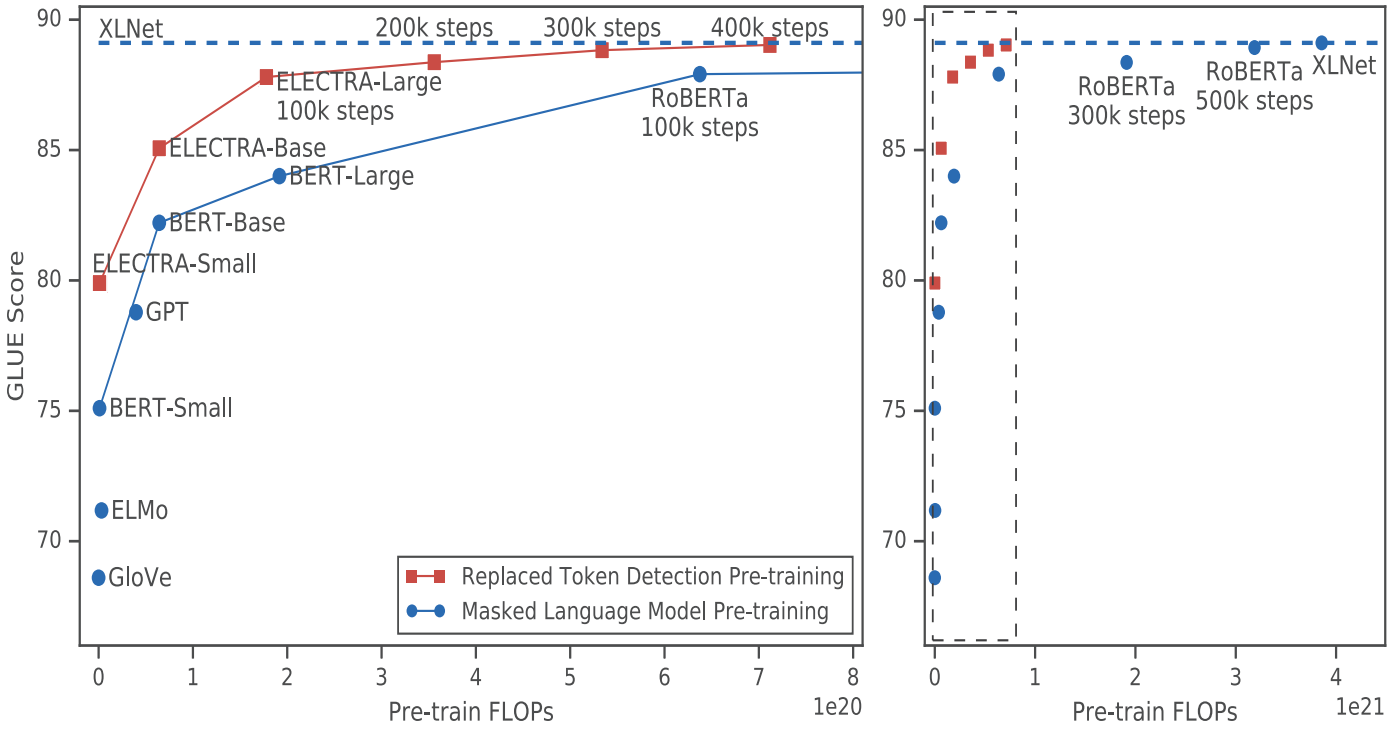


Electra

ABSTRACT

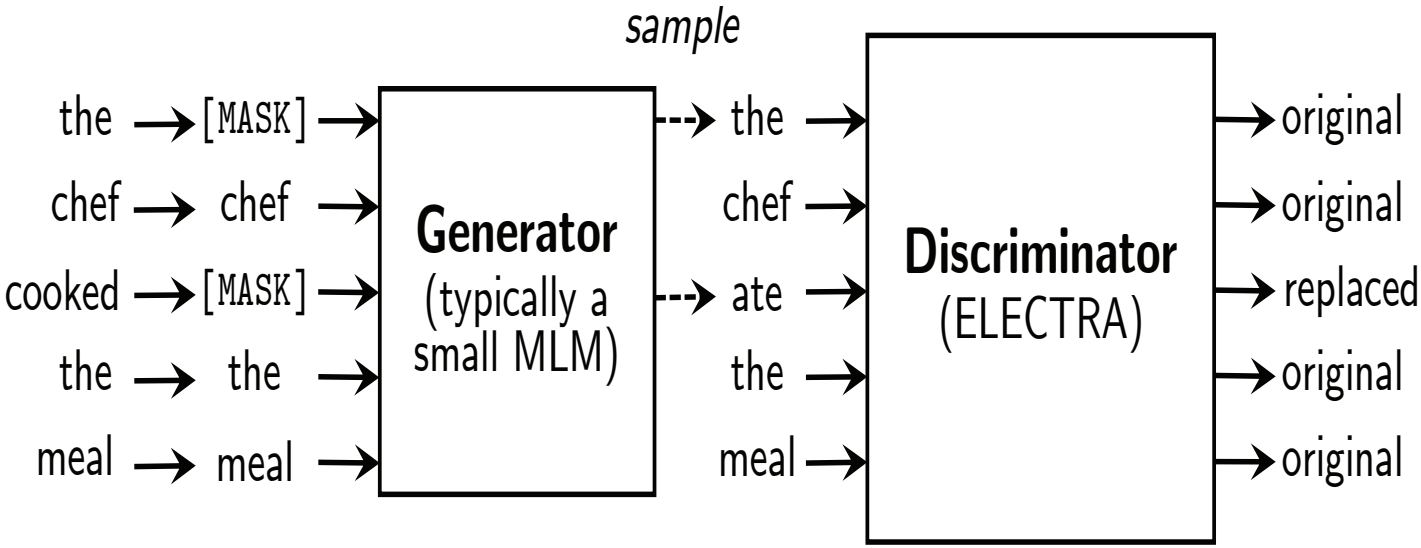
Masked language modeling (MLM) pre-training methods such as BERT corrupt the input by replacing some tokens with `[MASK]` and then train a model to reconstruct the original tokens. While they produce good results when transferred to downstream NLP tasks, they generally require large amounts of compute to be effective. As an alternative, we propose a more sample-efficient pre-training task called replaced token detection. Instead of masking the input, our approach corrupts it by replacing some tokens with plausible alternatives sampled from a small generator network. Then, instead of training a model that predicts the original identities of the corrupted tokens, we train a discriminative model that predicts whether each token in the corrupted input was replaced by a generator sample or not. Thorough experiments demonstrate this new pre-training task is more efficient than MLM because the task is defined over *all* input tokens rather than just the small subset that was masked out. As a result, the contextual representations learned by our approach substantially outperform the ones learned by BERT given the same model size, data, and compute. The gains are particularly strong for small models; for example, we train a model on one GPU for 4 days that outperforms GPT (trained using 30x more compute) on the GLUE natural language understanding benchmark. Our approach also works well at scale, where it performs comparably to RoBERTa and XLNet while using less than 1/4 of their compute and outperforms them when using the same amount of compute.

Electra Paper



Generator and Discriminator

- The generator can be any model that produces an output distribution over tokens, but we usually use a small masked language model that is trained jointly with the discriminator.
- Although the models are structured like in a GAN, we train the generator with maximum likelihood rather than adversarially due to the difficulty of applying GANs to text.
- After pre-training, we throw out the generator and only fine-tune the discriminator (the ELECTRA model) on downstream tasks.


sample
the
chef
cooked
the
meal
[MASK]
chef
[MASK]
the
meal
Generator
(typically a small MLM)
the
chef
ate
the
meal
Discriminator
(ELECTRA)
original
original
replaced
original
original

Weight Sharing

- We propose improving the efficiency of the pre-training by sharing weights between the generator and discriminator.
- If the generator and discriminator are the same size, all of the transformer weights can be tied.
- However, we found it to be more efficient to have a small generator, in which case we only share the embeddings (both the token and positional embeddings) of the generator and discriminator.

Smaller Generators

- If the generator and discriminator are the same size, training ELECTRA would take around twice as much compute per step as training only with masked language modeling.
- We suggest using a smaller generator to reduce this factor.
- Specifically, we make models smaller by decreasing the layer sizes while keeping the other hyperparameters constant

Training Algorithms

- The proposed training objective jointly trains the generator and discriminator.
- Train only the generator with L_{MLM} for n steps.
- Initialize the weights of the discriminator with the weights of the generator. Then train the discriminator with L_{Disc} for n steps, keeping the generator's weights frozen.

Bouns Exercise - Lecture 10

- Pretrain Electra model from scratch
- Class-Ex-Lecture10.py