**A Brief Review of Transformers and Transfer Learning in NLP**

**The Breakthrough: Using Language Modeling to Learn Representation**

2018 saw many advances in transfer learning for NLP, most of them centered around language modeling.  In case you're not familiar, language modeling is a fancy word for the task of predicting the next word in a sentence given all previous words. This seemingly simple task has a surprising amount of depth and the true potential of language modeling started to be unlocked by methods using it as a pretraining method.

The forerunners in this trend were [ULMFiT](https://arxiv.org/abs/1801.06146) and [ELMo](https://arxiv.org/abs/1802.05365), both of which used LSTM-based language models. The basic idea of these methods was to train a language model on massive amounts of unlabeled data and then use the internal representations of the language model on subsequent tasks with smaller datasets such as question answering and text classification. This was a form of transfer learning, where a larger dataset was used to bootstrap a model that could then perform better on other tasks. The reason this worked so well was that language models captured general aspects of the input text that were almost universally useful. Indeed, both ULMFiT and ELMo were a massive success, producing state-of-the-art results on numerous tasks.

**BERT**

Traditional language models are trained in a left-to-right fashion to predict the next word given a sequence of words. This has the limitation of not requiring the model to model *bidirectional context*. What does "bidirectional context" mean? For some words, their meaning might only become apparent when you look at both the left and right context *simultaneously*. The simultaneous part is important: models like ELMo train two *separate* models that each take the left and right context into account but do not train a model that uses both at the same time.

BERT solves this problem by introducing a new task in the form of masked language modeling. The idea is simple: instead of predicting the next token in a sequence, BERT replaces random words in the input sentence with the special [MASK] token and attempts to predict what the original token was. In addition to this, BERT used the powerful Transformer architecture (which I will explain next) to incorporate information from the entire input sentence.

Equipped with these two approaches, BERT achieved state-of-the-art performance across numerous tasks. If you want to learn more about BERT, please refer to [this blog post](http://mlexplained.com/2019/01/07/paper-dissected-bert-pre-training-of-deep-bidirectional-transformers-for-language-understanding-explained/) I wrote in the past.

**The Transformer Architecture**

BERT is based on the powerful Transformer architecture which is one of the factors that contributed to its incredible performance. Though I've written a [blog post](http://mlexplained.com/2017/12/29/attention-is-all-you-need-explained/) in the past about the Transformer, I'll give a quick overview of the Transformer here architecture as well, since it'll be important in understanding how XLNet works.

As its [original paper title](https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf) implies, the Transformer architecture is composed of multiple attention layers and gets rid of the recurrence mechanism of RNNs (ironically, recurrence gets reintroduced in XLNet, but we'll talk about that later). The Transformer is composed of multiple "Multihead Attention Layers". Each layer applies a linear transformation to the input and then uses attention to aggregate information across the sequence.

To be a bit more concrete, Multihead Attention Layers formulate attention as an operation that takes queries, keys, and values as input. Queries, keys, and values are all just tensors which are each transformed with a different linear transformation. For each query, the Multihead Attention Layer uses the key to computes an attention score for each value vector, then sums the value vectors using the attention weights into a single representation.  Intuitively, the query represents what kind of information we are looking for, the keys represent the relevance to the query, and the values represent the actual contents of the input.

The term "multi-head" comes from the fact that the Multi-Head Attention Layer uses multiple attention heads to compute multiple different attention weighted sums for each input. If there were 64 query vectors and 8 heads, each query vector would receive 8 separate representations for a grand total of 64 \* 8 = 512 representations.

This is great, but there is one glaring flaw we haven't yet addressed. A simple attention-based model cannot handle positional information. The solution to this problem used in the Transformer is simple: add positional embeddings to each word that express information regarding the position of each word in a sequence.

Henceforth, we'll use a very simplified view of the transformer architecture to make the discussion more concrete. Expressed in code, the core Transformer loop (heavily simplified) would look something like this [1](https://mlexplained.com/2019/06/30/paper-dissected-xlnet-generalized-autoregressive-pretraining-for-language-understanding-explained/#easy-footnote-bottom-1-1244):

embeddings = word\_embs + pos\_embs

h = [embeddings] + [None for \_ in attention\_layers]

for i, attention in enumerate(attention\_layers):

    h[i+1] = attention(queries=h[i], keys=h[i], values=h[i])

Here, the embeddings contain both the word embeddings and the positional embeddings.  Make sure you understand this because this will become crucial in understanding the next architecture: the Transformer XL.

**The Transformer XL**

Transformers were a game-changer in NLP due to their incredible performance and ease of training. However, they had a major drawback compared to RNNs: they had limited context.

Suppose you had a 50000-word long piece of text that you wanted to feed to a model. Feeding this into any model all at once would be infeasible given memory constraints. For an RNN you could work around this by simply chunking the text, then feeding the RNN one chunk at a time without resetting the hidden state between chunks. This works because the RNN is recurrent and as long as you keep the hidden state, the RNN can "remember" previous chunks, giving it a theoretically infinite memory.

For a Transformer, this is impossible because Transformers take fixed-length sequences as input have no notion of "memory". All its computations are stateless (this was actually one of the major selling points of the Transformer: no state means computation can be parallelized) so there is an upper limit on the distance of relationships a vanilla Transformer can model.

50000 words might be a bit of a stretch, but there are plenty of scenarios where you would want to feed very long sequences to a model. Language modeling is a prime example of this.

The Transformer XL is a simple extension of the Transformer that seeks to resolve this problem. The idea is simple: what if we added recurrence to the Transformer? Adding recurrence at the word level would just make it an RNN. But what if we added recurrence at a "segment" level. In other words, what if we added state between consecutive *sequences* of computations? The Transformer XL accomplishes this by caching the hidden states of the previous sequence and passing them as keys/values when processing the current sequence. For example, if we had the consecutive sentences

"I went to the store. I bought some cookies."

we can feed "I went to the store." first, cache the outputs of the intermediate layers, then feed the sentence "I bought some cookies." and the cached outputs into the model.

Expressed in code, the core loop for the Transformer XL would look something like this:

memory = init\_memory()

h = [embeddings] + [None for \_ in attention\_layers]

for i, attention in enumerate(attention\_layers):

ext\_h\_i = concat([h[i], memory[i])

h[i+1] = Attention(queries=h[i], keys=ext\_h\_i, values=ext\_h\_i)

# save memory

memory = h

This idea is great, but there is one flaw: position. In the Transformer, we handled position using positional embeddings. The first word in a sentence would have the "first position" embedding added to it, the second word would have the "second position" embedding added, and so on. But with recurrence, what happens to the positional embedding of the first word in the *previous* segment? If we're caching the Transformer outputs, what happens to the positional embedding of the first word in the current segment?

To address these issues, the Transformer XL introduces the notion of **relative positional embeddings**. Instead of having an embedding represent the absolute position of a word, the Transformer XL uses an embedding to encode the *relative distance* between words. This embedding is used while computing the attention score between any two words: in other words, the relative positional embedding enables the model to learn how to compute the attention score for words that are

words before and after the current word.

Now that we've covered the necessary background for this post, let's delve into the details of XLNet.

**What's Wrong with BERT?**

BERT was already a revolutionary method with strong performance across multiple tasks, but it wasn't without its flaws. XLNet pointed out two major problems with BERT.

1. The [MASK] token used in training does not appear during fine-tuning

BERT is trained to predict tokens replaced with the special [MASK] token. The problem is that the [MASK] token - which is at the center of training BERT - never appears when fine-tuning BERT on downstream tasks.

This can cause a whole host of issues such as:

* What does BERT do for tokens that are not replaced with [MASK]?
* In most cases, BERT can simply copy non-masked tokens to the output. So would it really learn to produce meaningful representations for non-masked tokens?
* Of course, BERT still needs to accumulate information from all words in a sequence to denoise [MASK] tokens. But what happens if there are no [MASK] tokens in the input sentence?

There are no clear answers to the above problems, but it's clear that the [MASK] token is a source of **train-test skew** that can cause problems during fine-tuning. The authors of BERT were aware of this issue and tried to circumvent these problems by replacing some tokens with random real tokens during training instead of replacing them with the [MASK] token. However, this only constituted 10% of the noise. When only 15% of the tokens are noised to begin with, this only amounts to 1.5% of all the tokens, so is a lackluster solution.

2. BERT generates predictions independently

Another problem stems from the fact that BERT predicts masked tokens in parallel. Let's illustrate with an example: Suppose we have the following sentence.

*I went to [MASK] [MASK] and saw the [MASK] [MASK] [MASK].*

One possible way to fill this out is

*I went to New York and saw the Empire State building.*

Another way is

*I went to San Francisco and saw the Golden Gate bridge.*

However, the sentence

*I went to San Francisco and saw the Empire State building*

is not valid. Despite this, BERT predicts all masked positions in parallel, meaning that during training, it does not learn to handle dependencies between predicting simultaneously masked tokens. In other words, it does not learn dependencies between its own predictions. Since BERT is not actually used to unmask tokens, this is not *directly* a problem. The reason this can be a problem is that this reduces the number of dependencies BERT learns at once, making the learning signal weaker than it could be.

Note that neither of these problems is present in traditional language models. Language models have no [MASK] token and generate all words in a specified order so it learns dependencies between all the words in a sentence.

**The Best of Both Worlds: Permutation Language Modeling**

Of course, despite its flaws, BERT has one major advantage over traditional language models: it captures bidirectional context. This bidirectionality was a crucial factor in BERT's success, so going back to traditional language modeling is simply not an option. The question then becomes: can we train a model to incorporate bidirectional context while avoiding the [MASK] token and parallel independent predictions?

The answer is yes: XLNet does this by introducing a variant of language modeling called "permutation language modeling". Permutation language models are trained to predict one token given preceding context like traditional language model, but instead of predicting the tokens in sequential order, it predicts tokens in some random order. To illustrate, let's take the following sentence as an example:

*I like cats more than dogs.*

A traditional language model would predict the tokens in the order

*"I", "like", "cats", "more", "than", "dogs"*

where each token uses all previous tokens as context.

*The prediction scheme for a traditional language model. Shaded words are provided as input to the model while unshaded words are masked out.*

In permutation language modeling, the order of prediction is not necessarily left to right and is sampled randomly instead. For instance, it could be

*"cats", "than", "I", "more", "dogs", "like"*

where "*than*" would be conditioned on seeing "*cats*", "*I*" would be conditioned on seeing "*cats*, *than*" and so on. The following animation demonstrates this.

*An example of how a permutation language model would predict tokens for a certain permutation. Shaded words are provided as input to the model while unshaded words are masked out.*

Notice how the model is forced to model bidirectional dependencies with permutation language modeling. In expectation, the model should learn to model the dependencies between all combinations of inputs in contrast to traditional language models that only learn dependencies in one direction.

The difference between permutation language modeling and BERT is best illustrated below.

*The conceptual difference between BERT and XLNet. Transparent words are masked out so the model cannot rely on them. XLNet learns to predict the words in an arbitrary order but in an autoregressive, sequential manner (not necessarily left-to-right). BERT predicts all masked words simultaneously.*

As a word of caution, in permutation language modeling, we are not changing the *actual* order of words in the input sentence. We are just changing the order in which we *predict* them. If you're used to thinking of language modeling in a sequential manner, this may be hard to grasp: how can we change the order in which we predict tokens while not changing the order in which we feed them to the model? Just remember that Transformers use masking to choose which inputs to feed into the model and use positional embeddings to provide positional information. This means that we can feed input tokens in an arbitrary order simply by adjusting the mask to cover the tokens we want to hide from the model. As long as we keep the positional embeddings consistent, the model will see the tokens "in the right order".

**Using the Transformer XL**

Aside from using permutation language modeling, XLNet improves upon BERT by using the Transformer XL as its base architecture. The Transformer XL showed state-of-the-art performance in language modeling, so was a natural choice for XLNet.

XLNet uses the two key ideas from Transformer XL: relative positional embeddings and the recurrence mechanism. The hidden states from the previous segment are cached and frozen while conducting the permutation language modeling for the current segment. Since all the words from the previous segment are used as input, there is no need to know the permutation order of the previous segment.

The authors found that using the Transformer XL improved performance over BERT, *even in the absence of permutation language modeling*. This shows that better language models can lead to better representations, and thus better performance across a multitude of tasks, motivating the necessity of research into language modeling.

**XLNet Implementation: The Details**

Now that we have an overview of what makes XLNet unique, let's delve into the details. If you're just here for the overall idea, feel free to skip this section.

**Handling Position: Two-Stream Self-Attention**

For language models using Transformers, when predicting a token at position

, the entire embedding for that word is masked out **including the positional embedding**. This means that the model is cut off from knowledge regarding the position of the token it is predicting.

This can be problematic, especially for positions like the beginning of the sentence which have a considerably different distribution from other positions in the sentence. To address this problem, the authors introduce a second set of representations that incorporate positional information but mask the actual token, just for the sake of pretraining. This second set of representations is called the **query stream**. The model is trained to predict each token in the sentence using information from just the query stream.

The original set of representations that includes both the positional embedding and token embedding is called the **content stream**. This set of representations is used to incorporate all the information relevant to a certain word during pretraining. The content stream is used as input to the query stream, but not the other way around. This scheme is called "**Two-Stream Self-Attention**".

Let's make this more concrete. For the

-th prediction in a sentence, the content stream can be computed like this:

permutation = permute(range(T))

c = [embeddings] + [Tensor(T) for \_ in attention\_layers]

for i, attention in enumerate(attention\_layers):

for t, position in enumerate(permutation):

visible\_positions = permutation[:t+1] # We include the current word in the visible positions

        c[i][position] = attention(queries=q[i-1][position], keys=c[i-1][visible\_positions], values=c[i-1][visible\_positions])

(Note that this ignores mini-batching, vectorization, memory, etc. and is just here to conceptualize the Two-Stream Self Attention method).

This is just a standard forward pass for the Transformer, but expressed slightly more verbosely to make the correspondence with the notation in the original paper clearer. The same scheme can be expressed in mathematical notation as

is the representation of the embedding of the

th token and .

is the

-th word that we are predicting.

Now, the query stream can be computed like this:

permutation = permute(range(T))

c = compute\_query\_stream(permutation) # content

q = [w] + [Tensor(T) for \_ in attention\_layers] # query

for i, attention in enumerate(attention\_layers):

for t, position in enumerate(permutation):

visible\_positions = permutation[:t] # we exclude the current word from the visible positions

        q[i][position] = attention(queries=q[i-1][position], keys=c[i-1][visible\_positions], values=c[i-1][visible\_positions])

Here,

is some learnable parameter that ideally incorporates positional information. In mathematical notation, where

is the query stream, and

For each word, the query stream uses the content stream which encodes all the available contextual information for words up to the current word.

If this seems confusing, think of it this way. Suppose we are predicting the word "*like*" in the sentence

"*I like cats more than dogs*"

where the previous words in the permutation were "*more*" and "*dogs*". The content stream would encode information for the words "*more*" and "*dogs*". The query stream would encode the positional information of "*like*" and the information from the content stream which would then be used to predict the word "*like*".

During fine-tuning, the query stream is scrapped and the content stream is used as the text representation.

**Handling optimization difficulties**

Permutation language modeling is more challenging compared to traditional language modeling which apparently causes the model to converge slowly. To address this problem, the authors chose to predict the last

tokens in the permutation instead of predicting the entire sentence from scratch

**Modeling Multiple Segments**

Many downstream tasks that we would want to use XLNet to take multiple segments of text as input. For instance, in the case of question answering, we take a question and answer as input.

To enable the model to distinguish between words in different segments,  BERT learns a segment embedding. In contrast, XLNet learns an embedding that represents whether two words are from the same segment. This embedding is used during attention computation between any two words. This idea is very similar to the relative positional encoding idea introduced in the Transformer XL. Similar to BERT, XLNet also feeds special [CLS] and [SEP] tokens to delimit the input sequences. The advantage of this scheme is that XLNet can now be extended to tasks that take arbitrary numbers of sequences as input.

**Results**

You've probably already heard the news, but XLNet achieves state-of-the-art performance (beats BERT) across 18 tasks including:

* Text classification
* Question answering
* Natural language inference
* Duplicate sentence (question) detection
* Document ranking
* Coreference resolution

Below is the table for text classification.

As you can see, XLNet is incredibly good: it outperforms BERT on all datasets and often by a fairly substantial margin considering the error rates.

Through ablation studies, the authors found that both permutation language modeling and the usage of the Transformer XL architecture both contributed to the performance of XLNet.

To see all the results, [the original paper](https://arxiv.org/pdf/1906.08237.pdf) is the best source.

**Conclusion and Further Readings**

XLNet is an exciting development in NLP, not only because of its results but because it shows us that there is still room to improve upon for transfer learning in NLP. Last year was a crazy year for NLP, let's hope we get even more crazy advancements this year as well.

In future posts, I'll be covering how to actually use XLNet as well as how to build the Transformer XL from scratch, so please look forward to them!

**Further Readings**

[The original paper](https://arxiv.org/pdf/1906.08237.pdf)

[My previous blog post on the Transformer](http://mlexplained.com/2017/12/29/attention-is-all-you-need-explained/)

[My previous blog post on ELMo](http://mlexplained.com/2018/06/15/paper-dissected-deep-contextualized-word-representations-explained/)

[My previous blog post on BERT](http://mlexplained.com/2019/01/07/paper-dissected-bert-pre-training-of-deep-bidirectional-transformers-for-language-understanding-explained/)