Lecture 9: The Transformer, GPT1

Andrei Arsene Simion

Columbia University

March 29, 2023

Outline

- The Transformer
- GPT1

Outline

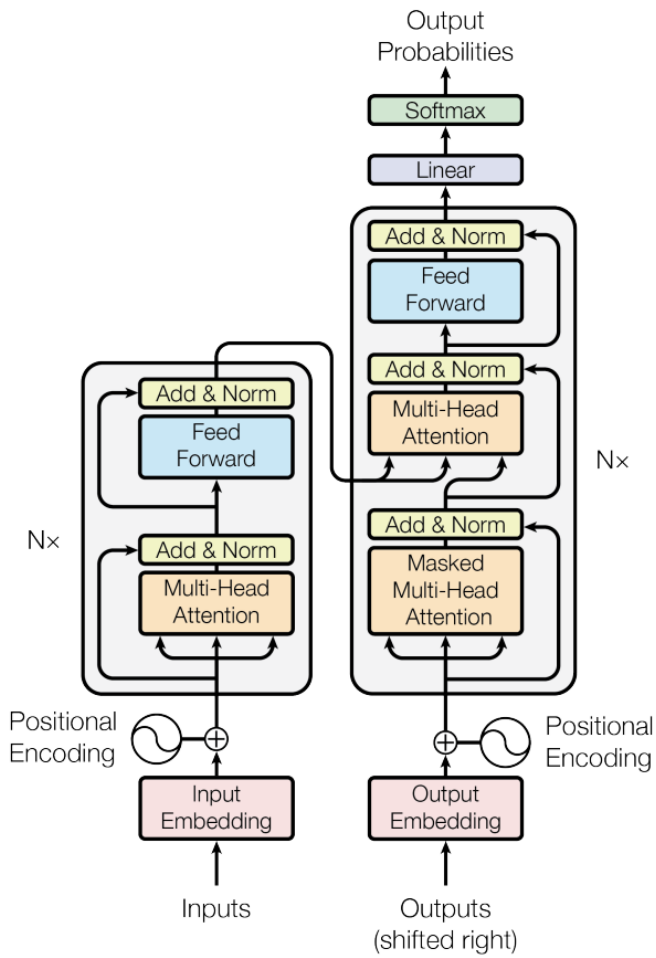
- The Transformer
- GPT1

The Transformer

- Up until 2017 researchers were mainly interested in LSTM / RNN models and variants of these models
- Basically, you take a task and then use the building blocks you have to get a model to work well on it
  - Remember ELMo?
  - Or BiDAF? Lots of LSTMs, different types of Attention
  - Or COVE? Lots of LSTMs and Attention mechanisms
- At a high level, they used LSTM models, Attention, and insights so that a model can be fit well to the task

The Transformer: The Model

- The Transformer represents a massive paradigm shift in NLP, where a new model took over
- Almost all of the models currently "hot" (Alpaca, ChatGPT, GPT4, Bard) are Transformer-based models



The Transformer: Why?

- What's wrong with RNN models?
- Main issue: people started observing that you can train large models and then fine tune them to different tasks
  - Think ULM-Fit: train an LSTM and then fine-tune
  - Or ELMo: train an LSTM and then for each token get a mixture of the hidden layer representations,

$$ELMo(x_t) = \gamma^{task} \sum_{l=0}^{L} s_l^{task} h_t^l,$$

  where
  - You optimize for γ^{task} and $\{s_l^{task}\}_{l=0}^{L}$
  - You might fine-tune the entire of $ELMo(x_t)$ as well (h_t^L)
- Generally, bigger models were seen to do better, but RNNs can't be parallelized!
- Why? For each element in a batch, you need to traverse its full length T sequentially, you cannot parallelize

The Transformer: We Want Parallelism

- In an RNN, a sequence $\{x_1, \ldots, x_T\}$ we need to process (h_0, x_1), then (h_1, x_2), etc
- Contrast this with a CNN: if we run a filter over the data, we can parallelize the application of that filter to different chunks of the data
- We can also parallelize across filters

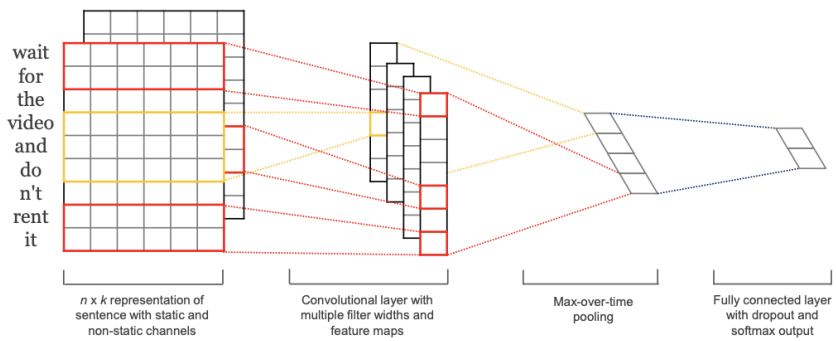


Figure 1: Model architecture with two channels for an example sentence.

- The transformer replaced the RNN/LSTM idea with an Attention based substitute that allows for parallelism
- The effective batch size of a Transformer/CNN is the number of words, not sequences, which is the effective batch size of an LSTM

Self Attention

- Big idea of the Transformer: Self-Attention
- Imagine a database analogy ...
- Assume you have a query q, a vector
- Assume you have a set of key vectors $\{k_1, \ldots, k_T\}$
- Assume each key has a value vector representation $\{v_1, \ldots, v_T\}$
- Given this, how can we express q in terms of v using the similarity between q and k?
- Idea: Attention!

Self Attention

- For each query $\{k_t\}_{t=1}^T$ you have a score $score_t = q^\intercal k_t$
- For all the scores, you can apply the softmax to get alignment probabilities $a = softmax(score)$ so a is a vector of probabilities
- Then, we can represent q as $\sum_{t=1}^T a_t v_t$, a weighted sum of vectors
- Assume that $q, k, v \in \mathbb{R}^d$ and their components are i.i.d. $N(0, 1)$ random variables ... What is the mean and variance of $q^\intercal k$?
- We have $E(q_j) = 0$ and $Var(q_j) = 1$ and similarly for k_j
  - $E(q^\intercal k) = \sum_{j=1}^d E(q_j)E(k_j) = 0$
  - $Var(q^\intercal k) = Var(\sum_{j=1}^d q_j k_j) = \sum_{j=1}^d Var(q_j k_j) = \sum_{j=1}^d Var(q_j)Var(k_j) = d$
- Because of this, we might want to set $score_t = q^\intercal k_t / d$ since then $score_t$ will have mean 0 and variance 1

Self Attention in NLP

- At a high level, the mechanism above is called Self-Attention: you have a query and you use its similarity with some keys (k) to get a expression for q in terms of v
- Example: for NLP, we have a sequence of token embeddings $\{x_1, \ldots, x_T\}$ ... How can we use the above?
- Assume each token embedding is dimension d_{model}
- Well, what if we fix $W^Q \in \mathbb{R}^{d_{model} \times d_q}$, $W^K \in \mathbb{R}^{d_{model} \times d_k}$ and $W^V \in \mathbb{R}^{d_{model} \times d_v}$ as matrices
  - Create $q_t = W^Q x_t$, the query vectors
  - Create $k_t = W^K x_t$, the key vectors
  - Create $v_t = W^V x_t$, the value vectors
  - Since we will take inner products of q_t and $\{k_s\}_{s=1}^T$, $d_q = d_k$
  - For each q_t, we can get a set of weights $a_{ts} = softmax(\frac{q_t^T k_s}{d_k})$ and get a new representation

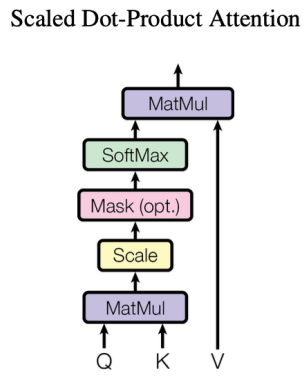
$$\sum_{s=1}^{T} a_{ts} v_s$$

for each q_t

Attention with Matrices

- If you have T queries-key-values we can succinctly write the new representations is

$$Attention(Q, K, V) = softmax(\frac{QK^\intercal}{\sqrt{d_k}})V$$



  - $Q \in \mathbb{R}^{T \times d_k}$
  - $K \in \mathbb{R}^{T \times d_k}$
  - $V \in \mathbb{R}^{T \times d_v}$; usually $d_v = d_k$
- In the NLP example, we went from $x \rightarrow q, k, v$ via $W^{Q,K,V}$ and then applied Attention; $Q = XW^Q$, $K = XW^K$, $V = XW^V$

Multi-Head Attention

- The discussion above can be generalized to Multi-Head Attention: you have h different triplet $\{W_i^{Q,K,V}\}_{i=1}^h$ and do the same thing

$$h_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

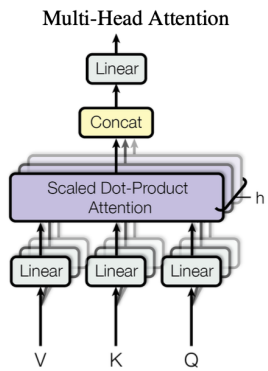
  - You can start with X and go to Q, K, V via $W^{Q,K,V}$
  - Or, just start with Q, K, V and apply $W^{Q,K,V}$ to each
  - For the example before, $X = Q, K, V \in \mathbb{R}^{d_{model} \times d_k}$
- After applying the logic above with multiple heads, you concatenate the results to get something of dimension hd_v
- Finally, we can have a matrix $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ which projects back to d_{model} space directly
- We then have

$$Multihead(Q, K, V) = Concat(head_1, head_2, \ldots, head_h)W^O$$

- Idea: each triple head $\{W^{Q,K,V}\}_{i=1}^h$ can learn different things so each $head_i$ is different

Multi-head Attention: Spelled-out Example!

- Suppose you have $h = 8$ heads
- Suppose you have a sentence $\{x_1, \ldots, x_T\}$ with word embeddings per token and $d_{model} = 512$ and $X \in \mathbb{R}^{T \times 512}$
- We have $d_k = d_q = d_{model}/8 = 64$
- Suppose we have 8 sets of triplets $\{W_i^{Q,K,V}\}_{i=1}^{8}$
- We map $X \to Q_i \in \mathbb{R}^{T \times 64}$ via $Q_i = XW_i^Q$ and similarly for K_i, V_i; $\{W_i^{Q,K,V}\}_{i=1}^{h} \in \mathbb{R}^{512 \times 64}$
- Apply self attention to get the new $head_i$ representations
- Each $head_i \in \mathbb{R}^{T \times 64}$ and we concatenate to get $(head_1, \ldots, head_8) = \mathbb{R}^{T \times 512}$
- $W^O \in \mathbb{R}^{512 \times 512}$
- The final representation is $(head_1, \ldots, head_8)W^O$

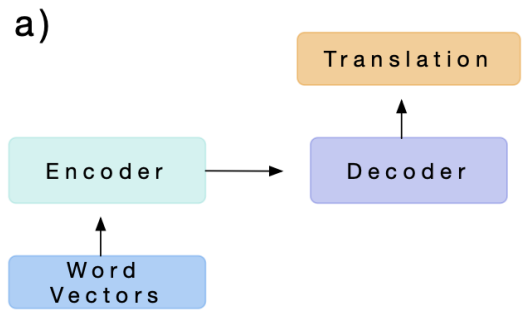


Review of Attention

- Starting with initial embeddings x_t, this mechanism allows us
  - Get different representations of q_t
  - Using the association between q_t and k_s, get a new representation of q_t that's aware of all the other k_s
  - Since we start with x_t, this is like a contextual representation of x_t in terms of all other x_s, but in another space
  - Multi-Head Attention: Why do we have 1 set of matrices $W^{Q,K,V}$; why not have h of them and then concatenate the results?
  - So, for each x_t we get a new representation of total length $d_{model} \rightarrow hd_v$
  - We map back to dimension d_{model} via W_O
  - Typically, $d_v = d_k$ and we can pick h, d_k so that $d_{model} = hd_k$

Back to the Transformer

- At a high level, the Transformer is an Encoder-Decoder model
- The main use case was Machine Translation, where we have a source sentence and want to translate to a target sentence
- Recall the picture from the COVE paper:



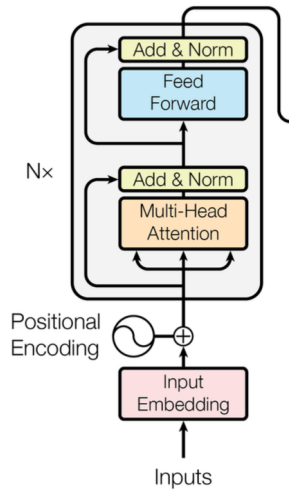
- But some differences
  - Encoder is no longer LSTM, it's a model based on Self-Attention
  - Word Vectors are not used, instead WordPiece tokenization is used (see later)

Transformer Encoder

- At a high level, we want to *mix* the representations x via Self-Attention so that a position x_t is aware of the ones around it
- This way, we can build a representation that's context-aware
- If you look at Attention above, you'll notice that it is position invariant! We need to add in positional information
  - "a b c d" and "d c a b" give the same *head* representation
  - To disallow this, we add positional embeddings so that we use $x_t + p_t$ where p_t tells us we are at the start, middle, etc of the sentence
- To help with gradients, we add residual connections and layer norm (LN)
- After each Self-Attention application, we pass each embedding through a Feed-Forward Neural Network with a ReLU activation that projects each token's embedding up to $4 * d_{model}$ dimension and then back down to d_{model} space
- We repeat this process several times until we get a good enough representation

Transformer Encoder - Algorithm of Encoder

- Step 1: X is the initial embeddings matrix and P_1 is the positional encoding matrix
- Step 2: Set $X = X + P_1$
- Step 3: Pick a number of heads h and do Multi-Head Self Attention for each triplet $\{W_{1,i}^{Q,K,V}\}_{i=1}^{h}$ and then concatenate to get H_1
- Step 4: Project H_1 so $H_1 = H_1 W_1^O$
- Step 5: Set $X = LayerNorm(X + H_1)$
- Step 6: Set $X = LayerNorm(X + FF(X))$
- Step 7: Repeat Step 3 - Step 6 several (N) times until you get the final token representations, a new X

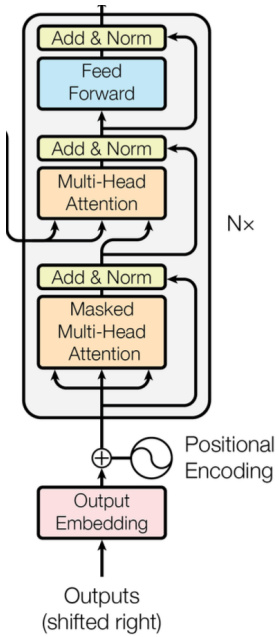


Transformer Decoder

- Once we have the representation of the Encoder's input, we use the Decoder to generate new data
- This is similar, but we can't look ahead
- To get a new representation of y_t via Self-Attention, we have to make sure that we always mix y_t with only other y_s that have $s \leq t$
- This is called Masked Multi-Head Attention
  - Practically, when we have $\sum_{s=1}^{T} a_{ts} v_s$ we want to make sure $a_{ts} = 0$ if $s > t$
  - This can easily be done by using a triangular mask over the score matrix so that $score_{ts} = -\infty$ if $s > t$
  - When we apply softmax, $a_{ts} = 0$

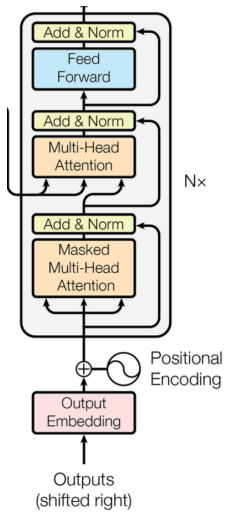
Transformer Decoder - Algorithm of Decoder

- Step 1: Y is the initial embeddings matrix and P_2 is the positional encoding matrix; note that the goal is to take $(y_0, \ldots y_{T-1})$ and predict $(y_1, \ldots, y_T)$
- Step 2: Set $Y = Y + P_2$
- Step 3: Pick a number of heads h and do Masked Multi-Head Self Attention for each triplet $\{W_{2,i}^{Q,K,V}\}_{i=1}^{h}$ and then concatenate to get H_2
- Step 4: Project H_2 so $H_2 = H_2 W_2^O$
- Step 5: Set $Y = LayerNorm(Y + H_2)$



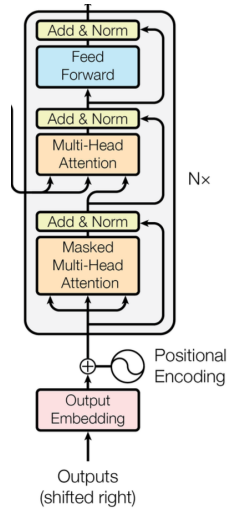
Transformer Decoder - Algorithm of Decoder

- Step 6: Bring in the final representations from the Encoder X so that now we have them and Y
- Step 7: Pick a number of heads h and do Masked Multi-Head Self Attention for each triplet $\{W_{3,i}^{Q,K,V}\}_{i=1}^{h}$
  - For queries, use Y so that $Q_i = YW_{3,i}^{Q}$
  - For keys, use X so that $K_i = XW_{3,i}^{K}$
  - For values, use X so that $V_i = XW_{3,i}^{V}$
- Step 8: Do Multi-head Self Attention as before to get H_3 and project $H_3 = H_3 W_3^O$
- Step 9: Set $Y = LayerNorm(Y + H_3)$

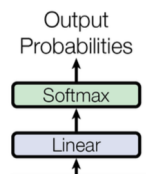


Transformer Decoder - Algorithm of Decoder

- Step 10: Set $Y = LayerNorm(Y + FF(Y))$
- Step 11: Repeat Step 3 - Step 10 several (N) times until you get the final token representations, a new Y



- For the loss, use a Cross-Entropy loss to try and predict the y_t from the representation of y_{t-1} after we do the above mixing several times over



Transformer Benefits and Drawbacks

- Benefits
  - There are no sequential operations - easy to parallelize!

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

  - Can train large models very fast due to the above
- Drawbacks
  - Self-Attention is Quadratic in length of the context T, this can make inference slow; large computational complexity
  - Context T is finite and must be specified upfront
  - You need lots of compute to train a large model
  - You still might want to use an LSTM if you have small data, don't care about very long range dependencies (speech), or preprocess via an LSTM before feeding to a Transformer

Results on Machine Translation

- State of the art BLEU scores

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

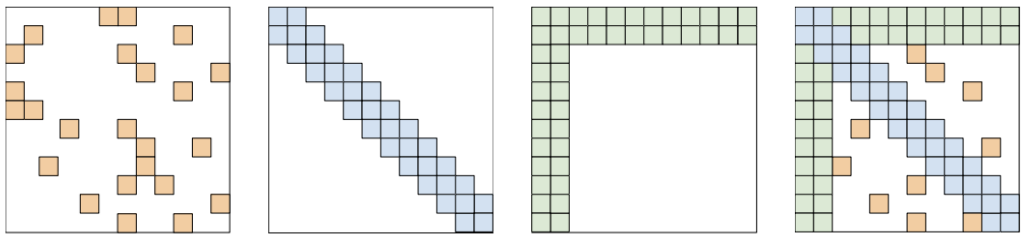
- You can train a model with 213 M parameters fairly quickly

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

Earlier Papers on Transformers

- Big Bird: How do we lower $O(T^2)$ Attention complexity?



(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

Figure 1: Building blocks of the attention mechanism used in BIGBIRD. White color indicates absence of attention. (a) random attention with $r = 2$, (b) sliding window attention with $w = 3$ (c) global attention with $g = 2$. (d) the combined BIGBIRD model.

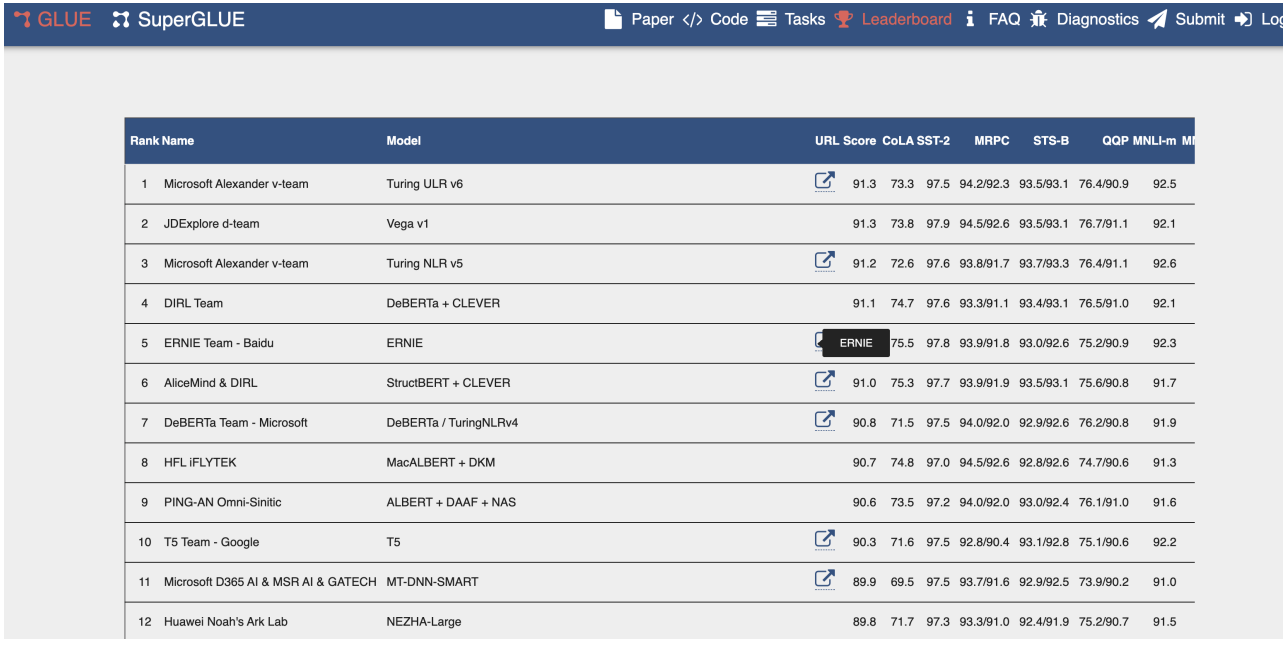
- Summarization: Great results; we'll look at this more later

Table 4: Performance of best models of each model architecture using the combined corpus and tf-idf extractor.

Model	Test perplexity	ROUGE-L
seq2seq-attention, $L = 500$	5.04952	12.7
Transformer-ED, $L = 500$	2.46645	34.2
Transformer-D, $L = 4000$	2.22216	33.6
Transformer-DMCA, no MoE-layer, $L = 11000$	2.05159	36.2
Transformer-DMCA, MoE-128, $L = 11000$	1.92871	37.9
Transformer-DMCA, MoE-256, $L = 7500$	1.90325	38.8

Transformers

- GLUE is a NLP benchmark with several tasks
- Transformer-based models have dominated



Rank	Name	Model	URL	Score	CoLA	SST-2	MRPC	STS-B	QQP	MNLI-m	MI
1	Microsoft Alexander v-team	Turing ULR v6		91.3	73.3	97.5	94.2/92.3	93.5/93.1	76.4/90.9	92.5	
2	JDExplore d-team	Vega v1		91.3	73.8	97.9	94.5/92.6	93.5/93.1	76.7/91.1	92.1	
3	Microsoft Alexander v-team	Turing NLR v5		91.2	72.6	97.6	93.8/91.7	93.7/93.3	76.4/91.1	92.6	
4	DIRL Team	DeBERTa + CLEVER		91.1	74.7	97.6	93.3/91.1	93.4/93.1	76.5/91.0	92.1	
5	ERNIE Team - Baidu	ERNIE		91.1	75.5	97.8	93.9/91.8	93.0/92.6	75.2/90.9	92.3	
6	AliceMind & DIRL	StructBERT + CLEVER		91.0	75.3	97.7	93.9/91.9	93.5/93.1	75.6/90.8	91.7	
7	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4		90.8	71.5	97.5	94.0/92.0	92.9/92.6	76.2/90.8	91.9	
8	HFL iFLYTEK	MacALBERT + DKM		90.7	74.8	97.0	94.5/92.6	92.8/92.6	74.7/90.6	91.3	
9	PING-AN Omni-Sinitic	ALBERT + DAAF + NAS		90.6	73.5	97.2	94.0/92.0	93.0/92.4	76.1/91.0	91.6	
10	T5 Team - Google	T5		90.3	71.6	97.5	92.8/90.4	93.1/92.8	75.1/90.6	92.2	
11	Microsoft D365 AI & MSR AI & GATECH	MT-DNN-SMART		89.9	69.5	97.5	93.7/91.6	92.9/92.5	73.9/90.2	91.0	
12	Huawei Noah's Ark Lab	NEZHA-Large		89.8	71.7	97.3	93.3/91.0	92.4/91.9	75.2/90.7	91.5	

What is WordPiece?

- The authors used WordPiece which is a tokenization that expressed words like "immaculate" as 2 tokens "imma" + "##culate"
  - "immaculate" $\rightarrow$ "imma" + "##culate"
- The tokes you choose have an impact on your model ...
- What is wrong with just using words and characters?
  - Characters will almost surely cover all words (no "[UNK]")
  - Characters lose semantic meaning
  - 4 Words could map into 20 characters so then you have to keep a long context and have models which are good at memory
  - Words require huge vocabularies and also what do you do when you have a new word?
- Instead of the most granular (characters) or the widest (words), find a better middle ground

What is WordPiece?

- High level idea of WordPiece:
  - Step 1: Begin with a set of character tokens
  - Step 2: Break up all words into tokens and get counts for tokens being next to each other
  - Step 3: From the most popular tokens next to each other, make a new token which is the concatenation of the above
  - Step 4: Given the new token, get a count as in Step 2 by greedily breaking up words into tokens
  - Step 5: Repeat Step 2 - Step 4 until you get a token universe of the desired size or you can't go on
- The transformer used 32000 tokens as its vocabulary V

WordPiece: Learning tokens

- Assume the Corpus is a bunch of words with frequencies

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

- Tokenize each at the most granular level
  - For example, "abc" gets tokenized to "a", "##b", "##c" since "b", "c" are inside of a word
- The above leads to each token having counts equal to:
  - ("h" "##u" "##g", 10), ("p" "##u" "##g", 5)
  - ("p" "##u" "##n", 12)
  - ("b" "##u" "##n", 4), ("h" "##u" "##g" "##s", 5)
- Given the above, the vocabulary is
  - $\rightarrow$ ["b", "h", "p", "##g", "##n", "##s", "##u"]
- For any pair of tokens, define the "information" score as

$$score(t_1, t_2) = \frac{freq(t_1, t_2)}{freq(t_1)freq(t_2)}$$

- Find the pair with the largest score, and create a new token by merging the tokens t_1 and t_2
- Repeat until you can't merge anymore or the vocabulary size is large enough

Word-Piece: Decoding

- With WordPiece, a word gets tokenized from left to right greedily; we try and use the largest token in the vocabulary as we go from left to right
- If we get to a place where a subword is not in the vocabulary, we classify the whole word as "[UNK]"
- For example
  - "bugs"
    - → ["b", WP("##ugs")]
    - → ["b", "##u", WP("##gs")]
    - → ["b", "##u", "##gs"]
  - "hug"
    - → ["hu", WP("##g")]
    - → ["hu", "##g"]
  - "bum"
    - → ["b", WP("##um")]]
    - → ["UNK"], since "##um" is not in the vocabulary

Outline

- The Transformer
- GPT1

GPT1

- The Transformer was a Encoder-Decoder Model, and after it came out it got a lot of buzz
  - How do you make it faster?
  - How do you apply it to other tasks?
  - What if you try and make it bigger?
  - What if you use just the Decoder? (GPT)
  - What if you use just the Encoder? And, how? (BERT)
- Let's look at GPT1 which addresses some of the above

GPT1: A big model for its time

- Transformer with 12 layers
- 768-dimensional representation of the token vectors; feed forward layers were 3072 dimensional
- Byte-Pair encoding tokenization with 40,000 merges
- Trained on book corpus: over 700 unique books
  - Contains long spans of contiguous text, good for learning long range dependencies
- The acronym "GPT" was never in the paper, but seems like it means "Generative Pretrained Transformer"
- GPT model sizes over time
  - GPT 1: 120 Million
  - GPT 2: 1.5 Billion
  - GPT 3: 175 Billion
  - ChatGpt: 175 Billion
  - GPT 4: XYZ Trilion ?

GPT1 ideas

- The idea was to train GPT1 on some large corpus as a language model and then "fine tune it for some task"
- This idea we saw in ULM-Fit, COVE, but it dates back to 2015 and the paper below which pretrains an LSTM and then uses it to Encode a piece of text which is then fed to a classifier

---

Semi-supervised Sequence Learning

Andrew M. Dai
Google Inc.
adai@google.com

Quoc V. Le
Google Inc.
qvl@google.com

Abstract

We present two approaches that use unlabeled data to improve sequence learning with recurrent networks. The first approach is to predict what comes next in a sequence, which is a conventional language model in natural language processing. The second approach is to use a sequence autoencoder, which reads the input sequence into a vector and predicts the input sequence again. These two algorithms can be used as a "pretraining" step for a later supervised sequence learning algorithm. In other words, the parameters obtained from the unsupervised step can be used as a starting point for other supervised training models. In our experiments, we find that long short term memory recurrent networks after being pretrained with the two approaches are more stable and generalize better. With pretraining, we are able to train long short term recurrent networks up to a few hundred timesteps, thereby achieving strong performance in many text classification tasks, such as IMDB, DBpedia and 20 Newsgroups.

GPT1 ideas

- Open AI researches trained the GPT1 model on a large corpus and fine-tuned it to 4 main tasks
- Entailment: A: "I walk home" B: "My dog is happy to see me arrive"; predict if $P(B \text{ follows } A)$ or not

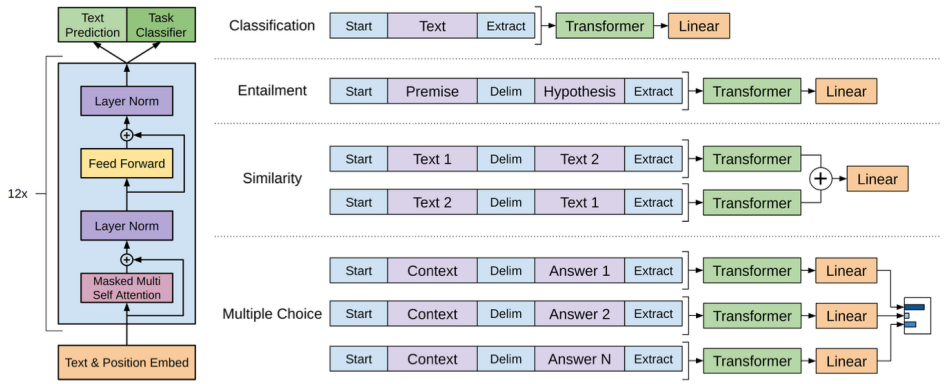


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

Table 1: A list of the different tasks and datasets used in our experiments.

Task	Datasets
Natural language inference	SNLI [5], MultiNLI [66], Question NLI [64], RTE [4], SciTail [25]
Question Answering	RACE [30], Story Cloze [40]
Sentence similarity	MSR Paraphrase Corpus [14], Quora Question Pairs [9], STS Benchmark [6]
Classification	Stanford Sentiment Treebank-2 [54], CoLA [65]

GPT1 ideas

- Without much work, just a little bit of model and input formatting, they got SOTA on several tasks
- Initially, they trained GPT1 as a Language Model so that we start with "The man walks fast" and predict "man walks fast home"
- Then they fed it data with specialized inputs like "[START] This is a great movie! [EXTRACT]"
- Usually, the [EXTRACT] token was what was fed as input into the classifier layers after

Table 2: Experimental results on natural language inference tasks, comparing our model with current state-of-the-art methods. 5x indicates an ensemble of 5 models. All datasets use accuracy as the evaluation metric.

Method	MNLI-m	MNLI-mm	SNLI	SciTail	QNLI	RTE
ESIM + ELMo [44] (5x)	-	-	89.3	-	-	-
CAFE [58] (5x)	80.2	79.0	89.3	-	-	-
Stochastic Answer Network [35] (3x)	80.6	80.1	-	-	-	-
CAFE [58]	78.7	77.9	88.5	83.3		
GenSen [64]	71.4	71.3	-	-	82.3	59.2
Multi-task BiLSTM + Attn [64]	72.2	72.1	-	-	82.1	**61.7**
Finetuned Transformer LM (ours)	**82.1**	**81.4**	**89.9**	**88.3**	**88.1**	56.0

Table 3: Results on question answering and commonsense reasoning, comparing our model with current state-of-the-art methods.. 9x means an ensemble of 9 models.

Method	Story Cloze	RACE-m	RACE-h	RACE
val-LS-skip [55]	76.5	-	-	-
Hidden Coherence Model [7]	77.6	-	-	-
Dynamic Fusion Net [67] (9x)	-	55.6	49.4	51.2
BiAttention MRU [59] (9x)	-	60.2	50.3	53.3
Finetuned Transformer LM (ours)	**86.5**	**62.9**	**57.4**	**59.0**

GPT1: Bype Pair Encoding

- GPT 1 used Byte-Pair encoding to get the universe of tokens
- This is very similar to WordPiece but a bit different
- You get all words and then start with all unique characters
- As in WordPiece, you define a score $score(t_1, t_2) = freq(t_1, t_2)$
- Identify the most frequent pair and add the rule $(t_1, t_2) \rightarrow t_1 t_2$
- Keep track of the order of the rules and use these rules when tokenizing new words

GPT1: Bype-Pair Encoding: Learning Rules

- Example:

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

- Tokenize into characters:

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

- The most frequent pair of tokens is ("u", "g") $\rightarrow$ "ug" with a count of 20, apply this rule and add it to vocabulary

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
```

- The most frequent pair of tokens is ("u", "n") $\rightarrow$ "un" with a count of 16, apply this rule and add it to vocabulary

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("h" "ug" "s", 5)
```

- The most frequent pair of tokens is ("h", "ug") $\rightarrow$ "hug" with a count of 15, apply this rule and add it to vocabulary

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]
Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

GPT1: Bype-Pair Encoding: Decoding new Words

- We learning the merge rules below, in order:

```
("u", "g") -> "ug"
("u", "n") -> "un"
("h", "ug") -> "hug"
```

- Consider "bug" which becomes ["[UNK]", "u", "g"]
- Apply the merge rules in order to get:
  - → ["[UNK]", "ug"]
- Consider "thug" which becomes ["[UNK]", "h", "u", "g"]
- Apply the merge rules in order to get:
  - → ["[UNK]", "h", "ug"]
  - → ["[UNK]", "hug"]
- Consider "unhug" which becomes ["u", "n", "h", "u", "g"]
- Applying the merge rules this becomes:
  - → ["u", "n", "h", "ug"]
  - → ["un", "h", "ug"]
  - → ["u", "n", "hug"]

References

- NMT Book
- Attention is All You Need
- Jay Ammar's Visuals on ELMo
- GPT 1 paper
- Semi Supervised Sequence Learning
- Byte Pair Encoding
- Word Piece
- The Annotated Transformer