**BERT – Research Series, ALBERT**

**––––––––––––––––––––**

**Basics:**

**–––––––**

Things to consider doing:

- <https://towardsdatascience.com/getting-started-with-google-colab-f2fff97f594c>

* HuggingFace: Library for transformers in NLP
  + -> Package is called ‘transformers’ (prvsly pytorch-transformers & pytorch-pretrained-bert)
    - Comes with 32+ pretrained models
* BERT uses two unsupervised training strategies:
  + Masked Language Model (MLM)
    - -> To avoid the word seeing itself; 15% of sentence randomly masked
    - Of these 15% Masked words:
      * 80% replaced w/ MASK token
      * 10% with random token
      * 10% unchanged
  + Next Sentence Prediction (NSP)
* BERT cased: true case & accent marks
* BERT Uncased: lower case & no accent marks
* When installing packages just look on anaconda website – line for most packages provided
* Projection Matrices:
  + Netflix example:
  + Vector of shape (1, 10K) -> 1's at the movies Joe has seen
  + Vector of shape (1, 10K) -> 1's at the movies Jill has seen
  + Projection Matrix:
  + Matrix of shape (10K, 100) -> Scoring the 10K movies on 100 genres
  + Taste Vectors:
  + Dot product Joe(Jill) x Projection -> (1, 100) 'Joe's taste', as we go through all 100 genres and look which one's Joe has most movies in
  + For both Joe & Jill we can then look at the similarity of their taste. If their taste is very similar, we'll recommend a movie Joe has seen that Jill hasn't seen to Jill.

**ALBERT – Should you switch from BERT to ALBERT?**

**––––––––––––––––––––**

<https://www.youtube.com/watch?v=vsGN8WqwvKg&t=89s>

Problem:

* BERT is slow for fine-tuning & training
* A Lite BERT (ALBERT)

Changes in ALBERT:

* Parameter Sharing
  + All encoders have the same weights – It’s like applying one encoder 12x times
    - On its own (BERT-base with parameter-sharing) will perform worse than BERT with unique weights // Just as BERT with only scaling the hidden-size performs worse
    - Substituting the unique encoders for scaling in hidden-size though improves performance
* Embedding Factorization
  + Embedding size is much smaller (128 instead of 768) and then increased via the Q, V, K projection matrices
* Sentence Order Prediction
  + New training task
* N-gram Masking
* LAMB Optimizer
* Sentence Piece Model

Other:

* 89% parameter reduction to BERT (for ALBERT base)
* ALBERT xxlarge: Hidden size of 4096 instead of 768
* As ALBERT xxlarge has 235M parameters in one encoder, it has only 235M unique parameters (vs BERT xlarge with 1270M unique parameters)
  + This makes file-size considerably smaller, but inference-time / finetuning-time will be similar (Backprop is still run on all weights)

**BERT Research 8 – Inner Workings V – Pre-training tasks / Masked-language Model**

**––––––––––––––––––––**

<https://www.youtube.com/watch?v=Fs8Zb4T-_CE>

<http://jalammar.github.io/illustrated-transformer/>

<http://nlp.seas.harvard.edu/2018/04/03/attention.html>

A Good Pre-training Task:

Does not need to be:

Useful (No real-world application)

Solvable (Ok for model to perform poorly)

Should be:

Unsupervised (Train on huge amounts of text)

Difficult (Requires sophisticated understanding)

MLM & NSP:

* 12% of tokens are masked (Replaced by [MASK] token)
* 1.5% of tokens replaced with random token (in unigram distribution, i.e. more frequent words get picked more often)
* 1.5% of tokens 'flagged for prediction'
* -> [CLS] passed to NSP classifier after all encoders
* -> Tokens we swapped / flagged passed to MLM classifier
  + Uses a softmax to predict the most likely token which was at that swapped/flagged place
  + -> 1.5% & 1.5% to have some differentiation from only masks
  + With only masks, the model might rely too much on only getting the meaning of words from the context (& in a real-world setting where we don't have masks, we'd also want it to infer the meaning from the word itself)

BERT is pre-trained 90% on sequence length 128 & 10% on 512.

OpenAI GPT used unidirectional attention, as otherwise the model would have already seen future words when then masking them.

BERT solves this problem by masking out words completely and then just using a new sentence in the next training step!

-> This step wasn't obvious as people thought it would take significantly longer to train, as you couldn't use each input word, but just 1 in 7. Turns out you only need ~20-30% longer training time rather than 7x as much

**BERT Research 7 – Inner Workings IV – FFN & Positional Encoding**

**––––––––––––––––––––**

<https://www.youtube.com/watch?v=YIEe7d7YqaU>

<http://jalammar.github.io/illustrated-transformer/>

<http://nlp.seas.harvard.edu/2018/04/03/attention.html>

FFN corresponds to the BertIntermediate & BertOutput

FFN in total has ~2.4m + 2.4m = 4.8m Weights

The self-attention layer ~2.4m

BertIntermediate acts like a hidden layer, scaling the 768 input vector to 4x768 (3072)

Only before the first encoder, when we look up the embeddings, we add a positional vector to each embedding.

-> They are not nec. hard-coded, generated by a fixed positional encoding function, as the paper implies they are partly learned

-> Informs the model about the absolute position of a word & the relative distance between words

-> Uses a sine & cosine function for even and uneven words

-> Detailed exp: <https://kazemnejad.com/blog/transformer_architecture_positional_encoding/>

**BERT Research 6 – Inner Workings III – Multi-headed Attention**

**––––––––––––––––––––**

<https://www.youtube.com/watch?v=0U1irnILcN0>

<http://jalammar.github.io/illustrated-transformer/>

<http://nlp.seas.harvard.edu/2018/04/03/attention.html>

'Applying the self-attention concept multiple times'

Improves performance in two ways:

1. Prevents final matrix to be too much influenced by the word itself (as its q x k is very high)
2. Gives the attention layer multiple representation subspaces; We use n sets of Q, K, V weight matrices ('n-heads')

If n is 8, we produce 8 different Z-matrices (final self-attention outputs).

We concatenate them and dot product with another WO weight matrix.

WO has two functions:

a) Scaling Z if necessary to the needed dimension

-> If we used query dimensions which are embedding/n, such as 512 / 8 = 64 (-> Q is 512 x 64) then just concatenating the 8 attention heads brings us back to a Z matrix of (words x 512) (no rescaling necessary)

-> BERT uses 12 heads with len 64 and 768 embeddings, hence no rescaling necessary -> The WO matrix in BertSelfOutput is simply (768, 768)

b) Adjust how we combine/weigh the attention heads

**BERT Research 5 – Inner Workings II – Self-Attention**

**––––––––––––––––––––**

<https://www.youtube.com/watch?v=a1Hc9soLxts>

<http://jalammar.github.io/illustrated-transformer/>

<http://nlp.seas.harvard.edu/2018/04/03/attention.html>

BERT contains 12 encoder layers.

Each encoder layer in a transformer consists of:

* Self-attention layer
* Feed-forward NN

Structure:

BertEmbedding

1. Word-embeddings dimension of 768 (512 + 256)
2. Positional Encoding
3. Sequence Encoding

BertEncoder

1. BertSelfAttention
   1. Query
   2. Key
   3. Value
   4. Dropout
2. BertSelfAttentionOutput
   1. Linear (768 -> 768) (WO-matrix)
   2. LayerNorm
   3. Dropout
3. BertIntermediate
   1. Linear (768 -> 3072 (4 \* 768))
4. BertOutput
   1. Linear (3072 -> 768)
   2. LayerNorm

-> While a word gets input from the other word embeddings during self-attention, afterwards it follows its own path through the feed forward NN. As we need to send each word through these 4 steps, it is highly parallelizable.

The first encoder's output, 'enriched embeddings', are then input into the next encoder.

Dot-product is large when both vectors point in the same direction.

Self-attention in detail:

1. Create 3 vectors from input embedding: Query, Key, Value

-> Created by multiplying the input embedding with 3 trained matrices.

-> These 3 trained matrices are the same for all words!

-> 'We're projecting the embedding vector into the Q, K & V space'

-> It like having three projection matrices as mentioned in the Netflix example

1. Create a Score by multiplying Q & K

-> Determines how much focus the respective word gets

-> Multiply the Q vector of the current word we process times all the other words K vectors (q1 \* k1; q1 \* k2..)

1. Divide Vectors by Square Root of the dimension

-> For dimension 64, divide by 8

1. Apply Softmax
2. Multiply Softmax score by Value vector to weigh the words
3. Sum up the weighted value vectors to get a final output vector

To speed up computation this can be done in matrices:

1. Stack embeddings in matrix X -> Perform Dot products to produce Q, K & V
2. Dot product btw. Q & K to produce Scores
3. Divide by Sq Root
4. Apply Softmax to rows
5. Dot Product with Value Matrix

Dot between Q & K acts as a similarity measure, hence for the Q & K of the same word, it will generally have a fairly high score.

**BERT Research 4 – Inner Workings I**

**––––––––––––––––––––**

<https://www.youtube.com/watch?v=C4jmYHLLG3A>

<http://jalammar.github.io/illustrated-transformer/>

<http://nlp.seas.harvard.edu/2018/04/03/attention.html>

Transformers use attention in a different way than used previously ('self-attention')

Neural Machine Translation: Applying an NN to translate languages

-> Introduced 2014 in paper NMT by jointly learning to align and translate

-> Deployed to G Translate in 2016

Transformers were originally for MT, but the encoder stack lends itself very well for other NLP tasks. -> BERT is just using the encoder stack

2 BERT Training Tasks:

1. MLM (Masked Language Model)
2. NSP (Next Sentence Prediction)

Training on multiple tasks is made possible by adjusting the output layer.

These are 'fake tasks', as they have no real world use and there's not 100% correct answer for each. It's about getting the weights.

**BERT Research 3 – Finetuning**

**––––––––––––––––––––**

<http://mccormickml.com/2019/07/22/BERT-fine-tuning/>

<https://www.youtube.com/watch?v=x66kkDnbzi4>

<https://www.youtube.com/watch?v=Hnvb9b7a_Ps>

*Chris McCormick*

* Advantages of BERT:
  + Quick Development – Via Transfer Learning; Huge amount of data BERT trained on
  + Less training data – With finetuning we can train BERT w/ less data
  + Better results – All previous task-tailored solutions are outperformed by the general BERT model, hence you might waste your time task-tailoring
* Problems of BERT:
  + Very Large -> Slow finetuning & inferencing:
    - Embed Layer: 30K Tokens mapped to768 vals each ~23M
    - 12 Transformers w/ about 7M weights each ~85M
    - -> Params stored as 32bit float -> 4 Byte \* 109M = 400 Megabytes for state\_dict
  + Jargon (Domain Speicifc Knowledge)
    - Just like I’d have no clue, BERT might have no clue listening to liver doctors
  + Not all NLP applications
    - Yes:
      * Classification
      * Named Entity Recognition
      * Part of Speech Tagging (POS)
      * Question answering (if answer provided in some text)
    - No:
      * Language Modelling (Whats the next word in a sentence likely to be;
      * Text Generation
      * Translatio
  + DataLoader – We could also replace these with looping through our dataset, but with a loop the entire dataset needs to be loaded into memory, while the dataloader acts like generator
  + BertForSequenceClassification etc, are just variatons of BERT by huggingface with e.g. a classification layer added
  + Token\_type\_ids are for multi-sentence tasks -> When using these we need to pass it a tensor with 1’s and 0’s for each sentence
  + Depending on whether or not we supply the labels to BERT it will either return the loss as output[0] or the output as output[0]

**BERT Research 2 – The embedding**

**––––––––––––––––––––**

<https://www.youtube.com/watch?v=zJW57aCBCTk>

[*https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/*](https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/)

*Chris McCormick*

* ‘WordPiece’: BERT can encode ANY word as it uses substrings -> embedding becomes em, ##bed, ##ding; It wll obviously learn to pay a lot of attention to the substrings bed & ding when processing the ‘em’
  + ## indicates subword / char
* It knows all chars + ~30K most common words; worst case a word is broken down to chars
  + It might still know a word like bedroom and not break it down to bed & room, as it tries to keep the most of the word together as possible
* At the time of tokenizing river bank & (money) bank are still the same, BUT ONLY AFTER RUNNING THEM through the MODEL ONCE and then taking the vectors of banks from one of the LAST LAYERS (for the first it will still be the same, remember BERT has 12 LAYERS!), we see that it understood meanings
  + - In fact – first layer still slightly diff., due to positional encoding or perhaps randomness
    - BERT uses 768 Features & has 30K words
* Word Embeddings are really just about grouping words in a coordinate system -> Having only one word in a coord has absolutley no meaning, but once you have mutliple it gets meaning
  + -> This is done by simply looking if they appear in similar contexts (& in which contexts they do not appear at all together)
  + ## subwords have their OWN place in the dictionary i.e. in the 30K words there is one entry for bed and one entry for ##bed
  + There are also numbers etc in the vocabulary
  + -> JP/CN characters are similarly encoded into sub ‘signs’ (Radicals)
* According to Han Xiao (Bert-as-service), the embeddings start out in the first layer as having no contextual information (i.e., the meaning of the initial ‘bank’ embedding isn’t specific to river bank or financial bank) -> Hence this means they retrieve it from some pre-defined embedding dict (as before)
  + https://github.com/hanxiao/bert-as-service

**BERT Research 1 – Basic Idea**

**––––––––––––––––––––**

[*http://mccormickml.com/2019/11/11/bert-research-ep-1-key-concepts-and-sources/*](http://mccormickml.com/2019/11/11/bert-research-ep-1-key-concepts-and-sources/)

[*https://www.youtube.com/watch?v=FKlPCK1uFrc*](https://www.youtube.com/watch?v=FKlPCK1uFrc)

*Chris McCormick*

Links:

Bert Paper: https://arxiv.org/pdf/1810.04805.pdf

Bert Repo: [github.com/google-research/bert](https://github.com/google-research/bert)

Google Post: https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html

Attention is all you need: https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

From scratch Tranformer Impementation: https://nlp.seas.harvard.edu/2018/04/03/attention.html

Jai Posts:

1. **BERT** —— [The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)](http://jalammar.github.io/illustrated-bert/)
   * Published: Dec 3, 2018
2. **Transformer** —— [The Illustrated Transformer](https://jalammar.github.io/illustrated-transformer/)
   * Published: Jun 27, 2018
3. **Attention** —— [Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)](https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)
   * **A screenshot of a cell phone

     Description automatically generated**Published: May 9, 2018

NLP Course, Coursera (Already completed)

* RNNs
* Encoder-Decoder
* LSTMs
* Bidirectional RNN
* Attention

Fake Tasks:

- Do not matter in real life, but we have vasts amount of data on -> BERT was trained on two fake tasks

**LSTMs vs Transformers**

**––––––––––––––––––––**

[*https://www.youtube.com/watch?v=S27pHKBEp30*](https://www.youtube.com/watch?v=S27pHKBEp30)

*Leo Dirac*

* Vanishing & Exploding Gradient
  + W \* W \* W \* W \* W – No matter what values these are; it’ll most likely to yield a very high or very low value; esp. if you initialize e.g. from -1 to 1 -> tiny negative / positive value
    - Eigenvalue of a vector >1 / <1 says whether it’s going to explode / vanish
      * Eigenvalue says where the vector moves when a transformation (scaling the vector by some factor) is applied; If it is just 1 then the vector stays the same
* LSTM Problems:
  + A screenshot of a cell phone

    Description automatically generatedLoads of gradients
  + Transfer learning doesnt really work
* Transformer – 3 Parts:
  + FC
  + Multi Head Attention
    - All-to-all comparison (Each layer is O(N^2) for seq len of N) -> Thanks to GPUs these are still efficient – while in an LSTM you need to process them step by step (as the result matters for the next)
    - -> Output is Weighted Sum of all inputs (Great to visualize) (See very clearly that for European Economic Zone it suddenly pays attention to more words than just the equivalent as for The)
    - This works by generating a QUERY for every output your considering and a KEY for every input -> The relevance score is the dot product
    - A screenshot of a cell phone screen with text

      Description automatically generatedValues are the 3rd version of each token, which you dot product with Softmax (relevance) -> Final Out
    - Attention mechanism in pseudo-code; Input is one tensor per token in the input (list of tensors); Beginning is the calculation of the Q & K & V of each input -> self.Q etc are learned matrices
    - ‘Multi-headed part: You do that attention mechanism x times (e.g. 8) w/ different Q,K,V matrices; This lets the network learn 8 different semantic meanings (e.g. vocab, gender, tense) -> it looks at different parts of the sentence and do this at each layer
  + Input Embedding / Pos Encoding
    - Going away from bag of words (where work to live & live to work is the same)
    - They still use word2vec to calculate some vector for each input token
    - Then add sines & cosines ONTO the word embeddings (inspired by Fourier Theory) of different frequencies starting at just pi and then stretching out longer & longer
      * This lets the model reason about the relative position of the words
* Why ReLU works so much better than tanh & sigmoid:
  + tanh & sigmoid cannot tell the difference of 10 & 20 (due to the derivative being ~the same)
  + Hinton: ReLUs allow each Neuron to express a strong opinion
  + ReLUs Gradients do not saturate (i.e. stop changing)
  + ReLUs are less sensitive to initalization (For sigmoid you need to make values will be in that middle part)
  + Runs also in worse hardware 8BITFLOAT possible
  + Derivatives are ridiculously easy
  + Downsides:
    - Dead Neurons left of 0 (-> leakyRELU)
    - Grad Discontinous at 0 (-> GELU) (Used by BERT)
* DL Wisdom:
  + Do not bother w/ trying diff. activations, just go w/ ReLU
  + Different optimizers DO make a difference
    - Adam – fast X tends to overfit
    - SGD – slow (takes long to converge) X tends to generalize better
    - Sts RMSProp works best
    - SWA can easily improve quality
    - AdaTune – You do not need to pick a LR, dynamically calculated
  + There are no local minima only saddle points
    - You want to get to the middle of the global minima, but SGD has problems doing so – it converges in circles once it is close to the minima (due to the LR, i.e. it’s jumping from before the minima to after etc)
      * -> Stochastic Weight Averaging – Averages the results around the minimum to get into the middle
* MegatronLM by NVidia: Going to the limit of compute
  + 8.3B parameters; 512 GPUs for 9 days (450K$ on EC2)
  + Similar w/ XLM RoBERTa
* Key Advantages of TF’s:
  + Easier To Train, More Efficient
  + TRANSFER LEARNING WORKS!
  + Can be trained UNSUPERVISED -> All of the worlds text data becomes available
* LSTMs still good when:
  + Sequence length is long / infinite, such as real time control for robots
* word-CNNs
  + Similar to LSTMs only take a part of the text
  + Have the advantage of parallelization like Transformers
  + Do not need the positional encoding like Transformers
  + Cannot like transformers answer the question Does this concept exist anywhere in the document