Rishabh Indoria
21F3001823

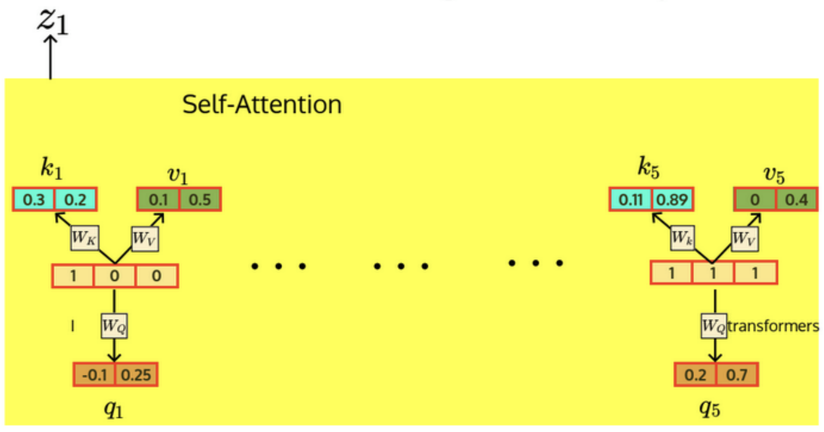
Large Language Models

May 25, 2025

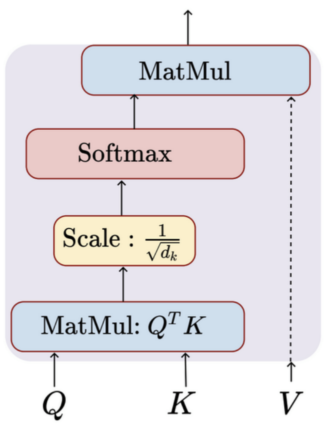
1 Transformer Architecture

1.1 Core Concepts

- The original "Attention is all you need" paper
- In RNN, the sequence-to-sequence translation doesn't happen in one go. It requires passing each input sequentially, one by one.
- The final hidden state in RNN is called a concept vector, thought vector, context vector, or annotation.
- However, the RNN decoder doesn't care about the alignment between the source sentence and the target sentence. Attention mechanism resolves this.
- Attention essentially gives a weighted combination of all hidden states from the encoder to the decoder, instead of just passing the last hidden state.
- Attention can always be parallelized.
- Now, we want to come up with a new architecture that incorporates the attention mechanism and also allows parallelization, and of course, get rid of vanishing/exploding gradient problems.
- **Self-Attention**: Self-attention allows a transformer model to attend to different parts of the same input sequence. Suppose we have word embeddings of some sentence, $\{h_i\}$. We have all the sequence inputs available at the same time. We need three vectors to compute self-attention
  q_j = the query vector for the word embedding $q_j = W_Q h_j$
  k_j = the key vector for the word embedding $k_j = W_K h_j$
  v_j = the value vector for the word embedding $v_j = W_V h_j$
  W_Q, W_K, and W_V are called respective linear transformation matrices.
  Q, K, and V can be computed directly via matrix multiplication.
  $Q = [q_1, q_2, \dots, q_N] = W_Q[h_1, h_2, \dots, h_N]$
  Attention is computed as
  $$Z = [z_1, z_2, \dots, z_N] = softmax\left(\frac{Q^T K}{\sqrt{d_k}}\right) V^T$$
  where d_k is the dimension of the key vector.



(a) Self-Attention Block



(b) Scaled Dot Product Head

- **Multi-Headed Attention**: Simply compute multiple self-attentions. The idea is that it captures diverse relationships and dependencies within input data. We concatenate the outputs of each self-attention block. If one block outputs N vectors of dimension d, then 2 self-attention blocks will output N vectors of dimension $2d$.
- We can also have an FFN right after every self-attention block, such that outputs from each self-attention block goes through a feedforward network. There will be the same number of FFN as self-attention blocks.

- The encoder is composed of N identical layers, and each layer is composed of 2 sub-layers.
- The computation is parallelized in the horizontal direction (i.e., within a training sample) of the encoder stack, not along the vertical direction.
- The decoder is a stack of N layers. However, each layer is composed of three sub-layers. It has masked multi-head self-attention, masked multi-head cross-attention, and FFN.
- Usually, we use only the decoder's previous prediction as input to make the next prediction in the sequence. However, the drawback of this approach is that if the first prediction goes wrong, then there is a high chance the rest of the predictions will go wrong, because of conditional probability. This will lead to an accumulation of errors.
- **Teacher Forcing**: It involves feeding observed sequence values (i.e., ground-truth samples) back into the RNN after each step, thus forcing the RNN to stay close to the ground-truth sequence.
- For this, we need to **mask** some inputs, so that it does not compute self-attention based on them.
- Of course, we can't use teacher forcing during inference. Instead, the decoder acts as an auto-regressor.
- **Masked Self-Attention**: This is similar to regular self-attention, with the key difference that future positions are masked out to prevent the model from attending to them. Specifically, we apply a mask to the attention scores by adding a matrix M, where the upper triangular part (excluding the diagonal) is set to $-\infty$, and the rest is set to 0. This ensures that, after applying the softmax, the attention weights for future tokens become effectively zero. Formally, the mask matrix is defined as:

$$M_{ij} = \begin{cases} 0, \text{ if } j \leq i \\ -\infty, \text{ if } j > i \end{cases}$$

  This matrix is added to the attention logits before the softmax operation.
- **Multi-Head Cross Attention**: Now we have the vectors $\{s_1, s_2, \ldots, s_N\}$ coming from the self-attention layer of decoder. We also have a set of vectors $\{e_1, e_2, \ldots, e_N\}$ coming from the top layer of the encoder stack that is shared with all layers of the decoder stack. Therefore, it is called Encoder-Decoder attention or cross-attention. We construct query vectors using vectors from the decoder layer and the key and value vectors using vectors from the encoder.
  $Q = W_Q S$, $K = W_K E$, and $V = W_V E$

1.2 Positional Encoding

- The position of words in a sentence was encoded in the hidden states of RNN based sequence to sequence models.
- However, in transformers, no such information is available to either encoder or decoder. Moreover, the output from self-attention is permutation invariant.
- So, it is necessary to encode the positional information.
- **Hypothesis**: Embed a unique pattern of features for each position j and the model will learn to attend by the relative position
- **Sinusoidal encoding function**

$$PE_{(j,i)} = \begin{cases} \sin\left(\frac{j}{10000^{\frac{2i}{d_{model}}}}\right), \text{ if } i = 0, 2, 4, \ldots, 254 \\ \cos\left(\frac{j}{10000^{\frac{2i}{d_{model}}}}\right), \text{ if } i = 1, 3, 5, \ldots, 255 \end{cases}$$

  where, $d_{model} = 512$. For the fixed position j, the value for i is sampled from sin() if i is even or cos() if i is odd. j represents the word position. The above is a 255 sized vector, can be changed.
- We can add this positional encoding to the word embedding.
- Add $j = 0$, to the first word embedding of all sentences.

1.3 Transformer Concepts

- Blog: The illustrated transformer
- LayerNormalization paper
- A book chapter on transformers by Dan Jurafsky (Draft Jan 2023 version)
- For a rough comparison, we may think of the transformer architecture is composed of attention layers and hidden layers.
- Then there is one attention layer and two hidden layers in the encoder layer, and 2 attention layers and 2 hidden layers in the decoder layer.
- How do we ensure the gradient flows across the network? **Residual Connections**
- How do we speed up the training process? **Normalization**
- We have three variables l, i, j involved in the statistics computation.
- At lth layer, let x_i^j denotes the activation of ith neuron for jth training sample
- **Batch Normalization**: Let us associate an accumulator with lth layer that stores the activations of batch inputs.

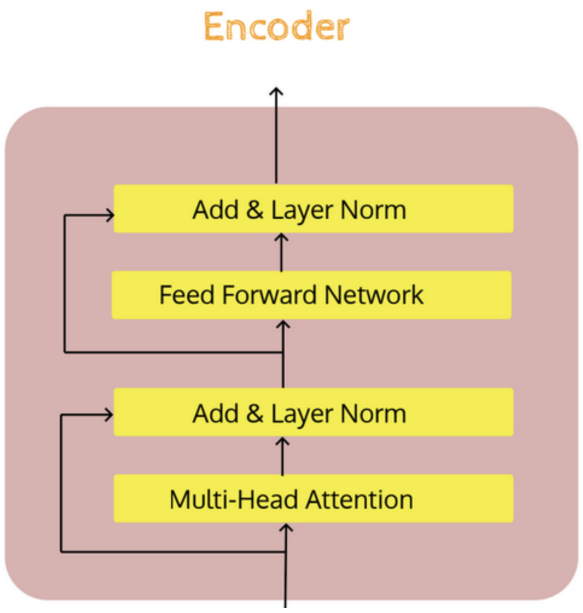
  $$\mu_{i=2} = \frac{1}{m}\sum_{j=1}^{m} x_2^j \qquad \sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_2^j - \mu_i)^2$$
  $$\hat{x} = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$
  $$\hat{y}_i = \gamma \hat{x}_i + \beta$$

  γ and β are parameters learned during training to allow the model to undo normalization if needed. $\hat{y}_i$ is the output.
- The accuracy of estimation of mean and variance depends on the size of m. So using a smaller size of m results in a high error.
- Because of this, we can't use a batch size of 1 at all.
- Other than this limitation, it was also empirically found that the naive use of BN leads to performance degradation in NLP tasks.
- There was also a systematic study that validated the statement and proposed a new normalization technique by modifying BN called powerNorm
- Fortunately, we have another simple normalization technique called Layer Normalization that works well.
- **Layer Normalization**: The computation is simple. Take the average across the outputs of hidden units in the layer. Therefore, the normalization is independent of the number of samples in a batch. This allows us to work with a batch size of 1.

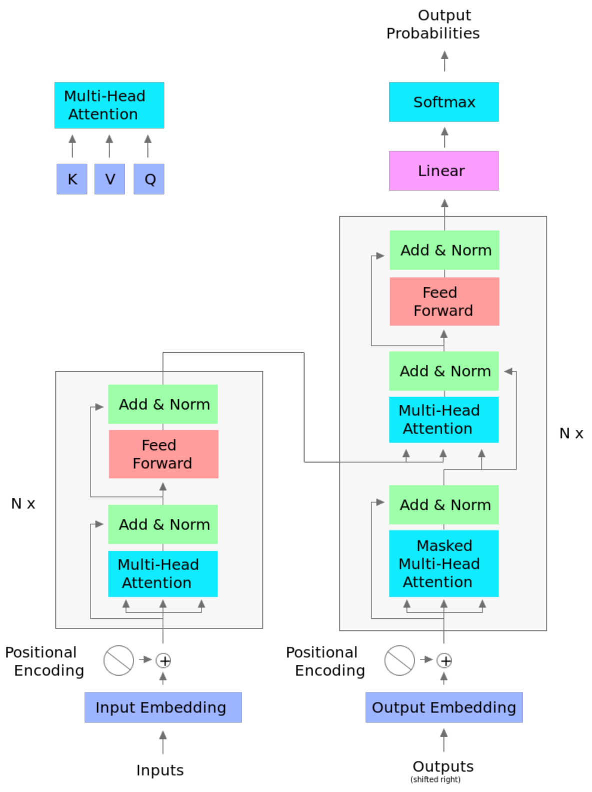
  $$\mu_l = \frac{1}{H}\sum_{i=1}^{H} x_i \qquad \sigma_l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(x_i - \mu_l)^2}$$
  $$\hat{x}_i = \frac{x_i - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}}$$
  $$\hat{y}_i = \gamma \hat{x}_i + \beta$$

- Add residual connection and layer norm block after every multi-head attention, feed-forward network, and cross-attention block.
- The input embedding for words in a sequence is learned while training the model. No pretrained embedding models like Word2Vec are used.
- This amounts to an additional set of weights in the model.
- The positional information is encoded and added with input embedding.

- The output from the top encoder layer is fed as input to all the decoder layers to compute multi-head cross-attention.



(a) Encoder Architecture



(b) Transformer Architecture

- Assume that decaying learning rate takes too long to converge and growing learning rate works well at the start but gets worse later. So, what to do next?
- **Warm Up**: Combine growing and decaying learning rate

$$\eta = \begin{cases} steps \cdot 4000^{-1.5} & \text{if } steps \leq 4000 \\ steps^{-0.5} & \text{otherwise} \end{cases}$$

  Here we do a 'warm-up' during initial steps and decrease the learning rate after "warm-up steps".

- This can also be written as

$$\eta = d_{model}^{-0.5} \cdot \min(step_{num}^{-0.5}, step_{num} \cdot warmupsteps^{-1.5})$$

2 Language Modeling Techniques

2.1 Approaches to Language Modeling

- Reference: `https://openai.com/index/language-unsupervised/`
- If we want to use the transformer architecture for other NLP tasks, then we need to train a separate model for each task using a dataset specific to the task.
- If we train the architecture from scratch for each task, it takes a long time for convergence.
- Often, we may not have enough labeled samples for many NLP tasks.
- On the other hand, we have a large amount of unlabeled text easily available on the internet.
- We develop a strong understanding of language through various language-based interactions (listening/reading) over our lifetime without any explicit supervision.
- Can a model develop a basic understanding of language by getting exposure to a large amount of raw text? **pre-training**

- More importantly, after getting exposed to such raw data, can it learn to perform well on downstream tasks with minimum supervision? **Supervised Fine-tuning**

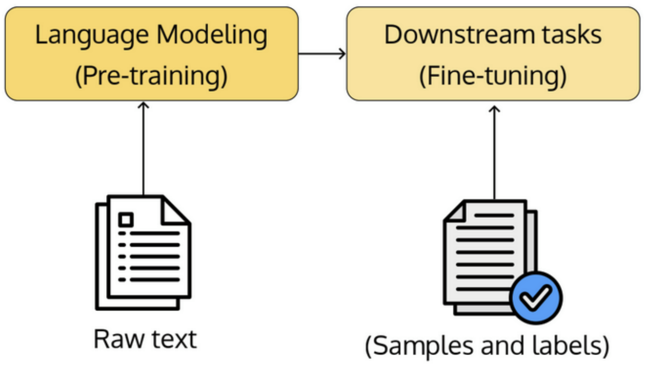

Language Modeling (Pre-training)
Downstream tasks (Fine-tuning)
Raw text
(Samples and labels)

Figure 3: Language Modeling

- **Language Modeling**: Let ν be a vocabulary of language. We can think of a sentence as a sequence $X_1, X_2 \ldots, X_n$, where $X_i \in \nu$.
- Intuitively, we expect some sentences to appear more frequently than others, hence, more probable.
- We are now looking for a function which takes a sequence as input and assigns a probability to each sequence

  $$f : (X_1, X_2, \ldots, X_n) \rightarrow [0, 1]$$

  Such a function is called a language model
- The joint probability of the sentence can be written as

  $$P(x_1, x_2, \ldots, x_T) = P(x_1)P(x_2|x_1)P(x_3|x_2, x_1) \cdots P(x_T|x_{T-1}, \ldots, x_1) = \prod_{i=1}^{T} P(x_i|x_1, \ldots, x_{i-1})$$

  If we naively assume that the words in a sequence are independent of each other then

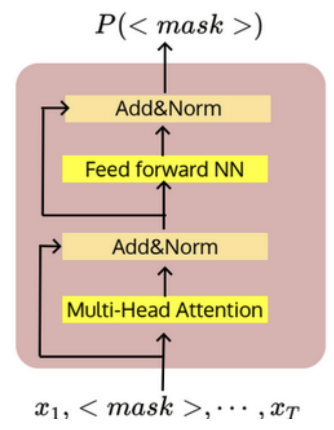
  $$P(x_1, x_2, \ldots, x_T) = \prod_{i=1}^{T} P(x_i)$$

- How do we enable a model to understand language?
  Simple Idea: Teach it the task of predicting the next token in a sequence.
- Roughly speaking, this task of predicting the next token in a sequence is called language modeling.
- However, we know that the words in a sentence are not independent but depend on the previous words. Hence, the naive assumption does not make sense

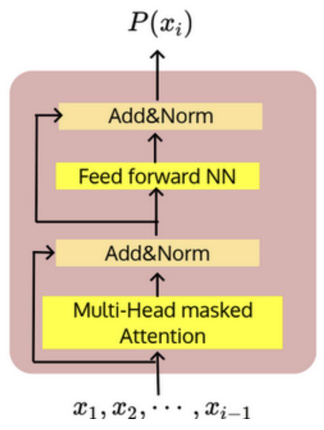
  $$\prod_{i=1}^{T} P(x_i|x_1, \ldots, x_{i-1}) : \text{Current word } x_i \text{ depends on previous words } x_1, \ldots, x_{i-1}$$

- How do we estimate these conditional probabilities?
  One solution: Use **autoregressive models** where the conditional probabilities are given by parameterized functions with a fixed number of parameters (like transformers)
- **Causal Language Modeling**: We are looking for f_θ such that

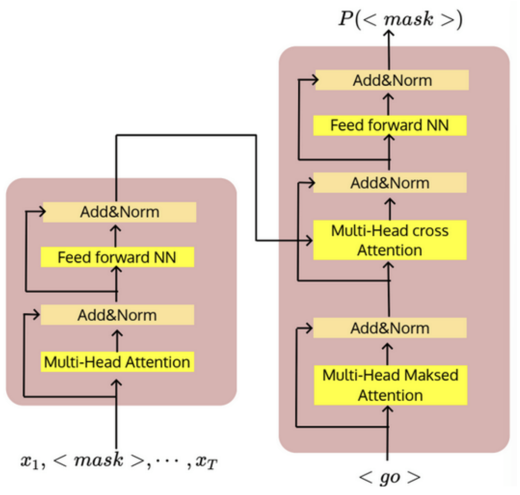
  $$P(x_i|x_1, \ldots, x_{i-1}) = f_\theta(x_i|x_1, \ldots, x_{i-1})$$



(a) Using only the encoder of the transformer (encoder only models)



(b) Using only the decoder of the transformer (decoder only models)



(c) Using both the encoder and decoder of the transformer (encoder decoder models)

Figure 4: Some Possibilities

- The input is a sequence of words. We want the model to see only the present and past inputs. We can achieve this by applying the mask. The masked multi-head attention layer is required. However, we do not need multi-head cross-attention layer (as there is no encoder)
- The outputs represent each term in the chain rule

$$= P(x_1)P(x_2|x_1)P(x_3|x_2, x_1)P(x_4|x_3, x_2, x_1)$$

  However, this time the probabilities are determined by the parameters of the model

$$= P(x_1; \theta)P(x_2|x_1; \theta)P(x_3|x_2, x_1; \theta)P(x_4|x_3, x_2, x_1; \theta)$$

  Therefore, the objective is to maximize the likelihood $L(\theta)$
- **Generative Pretrained Transformer (GPT)**: Now we can create a stack (n) of modified decoder layers (called transformer block in the paper). Decoder only model
- Let X denote the input sequence

$$h_0 = X \in \mathbb{R}^{T \times d_{model}}$$

$$h_l = transformer\_block(h_{l-1}), \quad \forall l \in [1, n]$$

  where $h_n[i]$ is the ith output vector in h_n block.

$$P(x_i) = softmax(h_n[i]W_v)$$

$$L = \sum_{i=1}^{T} \log(P(x_i|x_1, \ldots, x_{i-1}))$$

- The positional embeddings are also learned, unlike the original transformer, which uses fixed sinusoidal embeddings to encode the positions.
- Consider 12 decoder layers (transformer blocks), with context size 512, 12 attention heads, and FFN hidden layer size $768 \times 4 = 3072$, usually FFN hidden layer is 4 times the embedding dimension.
  Token embeddings weight: $|\nu| \times$ embedding dimension$= 40478 \times 768 = 31M$
  Position embeddings weights: context length $\times$ embedding dimension$= 512 \times 768 = 0.3M$
  Attention parameters per block, $W_Q = W_K = W_V = (768 \times 64)$, 64 because output of all 12 attention heads should combine to become 768(embedding dimension)
  Per attention head, $3 \times (768 \times 64) \approx 147 \times 10^3$
  For 12 heads, $12 \times 147 \times 10^3 \approx 1.7M$
  For a Linear layer, $768 \times 768 \approx 0.6M$
  For all 12 blocks, $12 \times 2.3 = 27.6M$
  FFN parameters per block, $2 \times (768 \times 3072) + 3072 + 768 = 4.7M$
  For all 12 blocks, $12 \times 4.7 = 56.4M$

2.2 Fine-Tuning and Adapting

- **Pre-Training**: Input size $(B, T, C) = (64, 512, 768)$, where B is batch size, T is context size, and C is the embedding dimension. We want to minimize

  $$L = -\sum_{x \in V : X \in \mathcal{X}} \sum_{i=1}^{T} y_i \log(\hat{y}_i)$$

  Optimizer: Adam with cosine learning rate scheduler
  Strategy: Teacher forcing (instead of auto-regressive training) for quicker and stable convergence

- **Fine-Tuning**: Involves adapting the model for various downstream tasks (with a minimal change in the architecture).

- Each sample in a labeled data set C consists of a sequence of tokens $x_1, x_2, \dots, x_m$ with the label y

- Initialize the parameters with the parameters learned by solving the pre-training objective. At the input side, add additional tokens based on the type of downstream task. For example, start and end tokens for classification tasks. At the output side, replace the pre-training LM head with the classification head (a linear layer W_y)

- Now our objective is to predict the label of the input sequence

  $$\hat{y} = P(y|x_1, \dots, x_m) = softmax(W_y h_l^m)$$

  Note that we take the output representation at the last time step of the last layer h_l^m. It makes sense as the entire sentence is encoded only at the last time step due to causal masking. Then we can minimize the following objective

  $$L = -\sum_{(x,y)} \log(\hat{y}_i)$$

  Note that W_y is randomly initialized. Padding or truncation is applied if the length of the input sequence is less than or greater than the context length.

- Now, we need to discourage degenerative (repeated or incoherent) texts, and encourage the model to be creative in generating a sequence for the same prompt.

- We also need to accelerate the generation of tokens

2.3 Decoding Strategies

- Reference: Beam search strategies for Neural Machine Translation

- Reference: The curious case of neural text degeneration

- Reference: Decoding stratagies by Maxime Labonne

- **Deterministic Strategies**: Exhaustive Search, Greedy Search, Beam Search, Contrastive Decoding

- **Stochastic**: Top-K sampling, Top-P (Nucleus Sampling)

- **Exhaustive Search**: Exhaustively search for all possible sequences with the associated probabilities and output the sequence with the highest probability.
  Suppose we want to exhaustive search for a sequence of length 2 with the vocabulary of size 3. At time step 1, the decoder outputs a probability for all 3 tokens. At time step 2, we need to run the decoder three times independently conditioned on all three predictions from the previous time step. At time step 3, we will need to run the decoder 9 times.
  If $|\nu| = 40000$, then we need to run the decoder 40000 times in parallel.

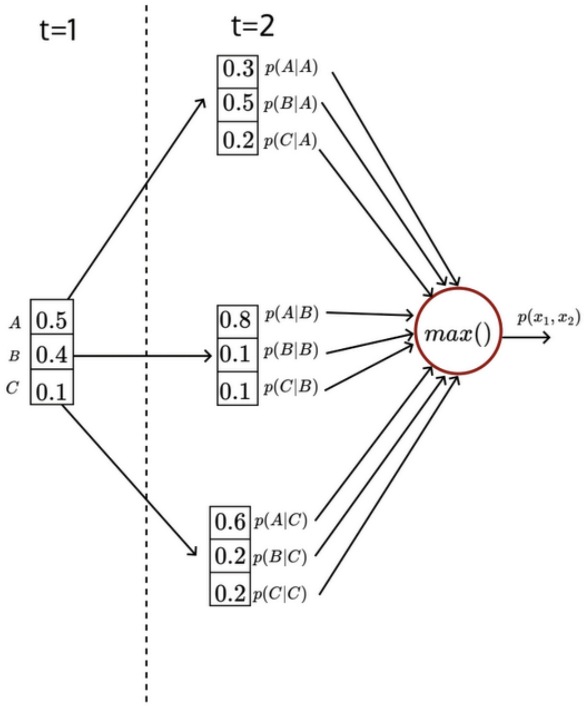
- Clearly, this will take way too long.

- **Greedy Search**: On the other extreme, we can do a greedy search. At each time step, we always output the token with the highest probability

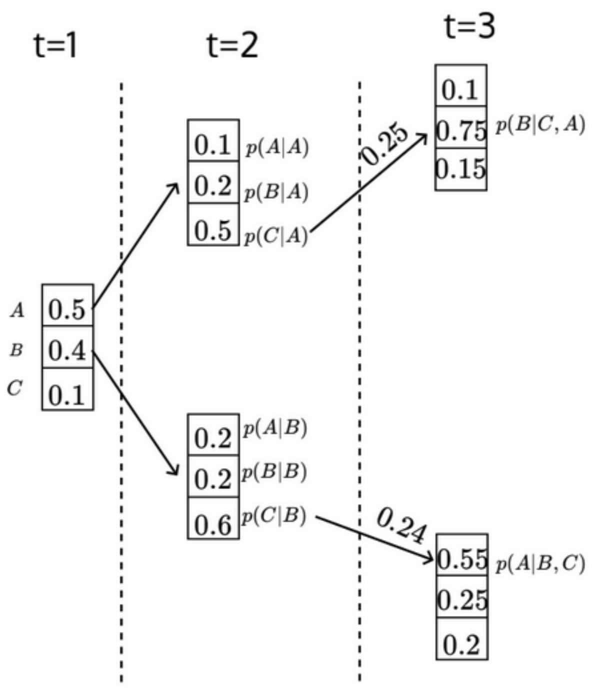
- If the starting token is the word "I", then it will always end up producing the same sequence.

- If we picked the second most probable token in the first time step, then the conditional distribution in the subsequent time steps will change and we would have ended up with a different sequence.

- Greedily selecting a token with max probability at every time step does not always give the sequence with maximum probability
- **Beam Search**: Instead of considering the probability for all the tokens at every time step (as in exhaustive search), consider only the top-k tokens. It requires $k \times |\nu|$ computations at each time step.
  Now we will have k sequences at the end of time step T and output the sequence with the highest probability.
  The parameter k is called the **beam size**. It is an approximation to exhaustive search.
  If $k = 1$, then it is equal to greedy search.
- Divide the probability of the sequence by its length, otherwise longer sequences will have lower probability



(a) Exhaustive Search Illustration



(b) Beam Search Illustration

- Both the greedy search and the beam search are prone to be degenerative.
- Neither greedy nor beam search can result in creative outputs.
- Note, however, that the beam search strategy is highly suitable for tasks like **translation** and **summarization**.
- **Top-K Sampling**: At every time step, consider top-k tokens from the probability distribution.
- Surprise is an outcome of being random
- Human predictions have a high variance, whereas beam search predictions have a low variance. Giving a chance to other highly probable tokens leads to a variety in the generated sequences.
- If we fix the value of k, then we are missing out on other equally probable tokens from the flat distribution. It will miss generating a variety of sentences (less creative).
- For a peaked distribution, using the same value of k, we might end up creating some meaningless sentences as we are taking tokens that are less likely to come next.
- **Low temperature sampling**: Given the logits, $u_{1:|\nu|}$, and temperature parameter T, compute the probabilities as
  $$P(x = u_l | x_{1:i-1}) = \frac{\exp(\frac{u_l}{T})}{\sum_{l'} \frac{u_{l'}}{T}}$$
  Low temperature = skewed distribution = less creativity
  High temperature = flatter distribution = more creativity
- **Top-P (Nucleus) sampling**: Sort the probabilities in descending order. Set a value for the parameter p, $0 < p < 1$. Sum the probabilities of tokens starting from the top token. If the sum exceeds p, then sample a token from the selected tokens.
  It is similar to top-k with k being dynamic.

3 Bidirectional Encoder Representations from Transformers

3.1 Masked Language Modeling

- In GPT, the representation for a language is learned in an unidirectional (left-to-right) way.
- This is a natural fit for tasks like text generation.
- How about tasks like Named Entity Recognition(NER), and fill in the blanks.
- We need to look at both directions (surrounding words) to predict the masked words
- Predict the masked words using the context words in both directions, like CBOW.
- This is called **Masked Language Modeling**(MLM)
- We cannot use the decoder component of the transformer as it is inherently unidirectional
- We can use the encoder part of the transformer
- Now, the problem is cast as
$$P(y_i|x_1, x_i = [mask], \ldots, x_T) = ?$$
It is assumed that the predicted tokens are independent of each other. We need to find out ways to mask the words in the input text.
- We know that each word attends to every other word in the sequence of words in the self-attention layer
- Our objective is to mask a few words randomly and predict the masked words
- Can we MASK the attention weights of the words to be masked as in CLM?
No, because we want the model to learn (attend to) what the blanks are.
- We can think of the [mask] token as noise that corrupts the original input text.
- Then the model is tasked to recover the original token. This is similar to the denoising objective of autoencoders.
- For this reason, MLM is also called as pre-training denoising objective
- A simple approach is to use [mask] token for the words to be masked. Add [mask] as a special token in the vocabulary and get an embedding for it.
- Note carefully that only a set M of masked words are predicted and hence the cross entropy loss for those predictions is computed. Typically, 15% of words in the input sequence are masked.
- Very high masking would result in severe loss of context information and the model will not learn good representations.
- On the other hand, very little masking takes a long time for convergence, because the gradient values are comparatively small, and makes training inefficient
- However, tradeoff could be made based on the model size, masking scheme and optimization algorithm
- Reference: Should you mask 15% in MLM?
- **BERT**: A multi-layer bidirectional transformer encoder architecture. BERT base model contains 12 layers with 12 attention heads per layer.
The masked words (15%) in an input sequence are sampled uniformly. Of these, 80% are replaced with [mask] token and 10% are replaced with random words, and the remaining 10% are retained as it is.
The special mask token won't be a part of the dataset while adapting for downstream tasks.
- Is pre-training objective of MLM sufficient for downstream tasks like Question-Answering, where interaction between sentences is important?
- **Next Sentence Prediction (NSP)**: Now, let's extend the input with a pair of sentences (A,B) and the label that indicates whether sentence B naturally follows sentence A. The two sentences are separated with a special token [SEP]. The hidden representation corresponding to the [CLS] token is used for final classification (prediction). CLS is a token that represents the entire input sequence or sentence and is placed at the beginning of the input.
In 50% of the instances, sentence B is the natural next sentence that follows sentence A. In 50% of the instances, sentence B is a random sentence from the corpus labeled as `NotNext`.

- Pretraining with NSP objective improves the performance of QA and similar tasks significantly.
- To distinguish the belonging-ness of the token to sentence A or B, a separate learnable segment embedding is used in addition to token and positional embeddings.

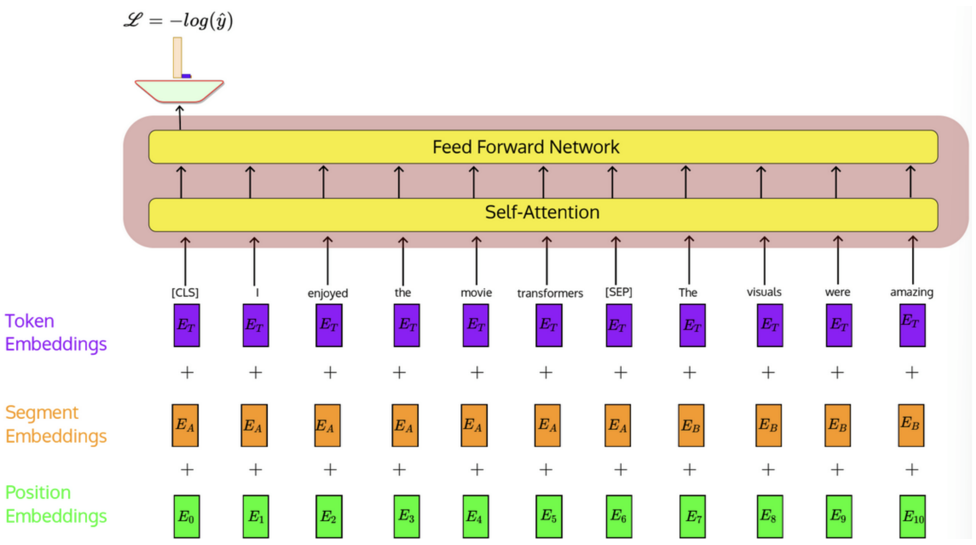
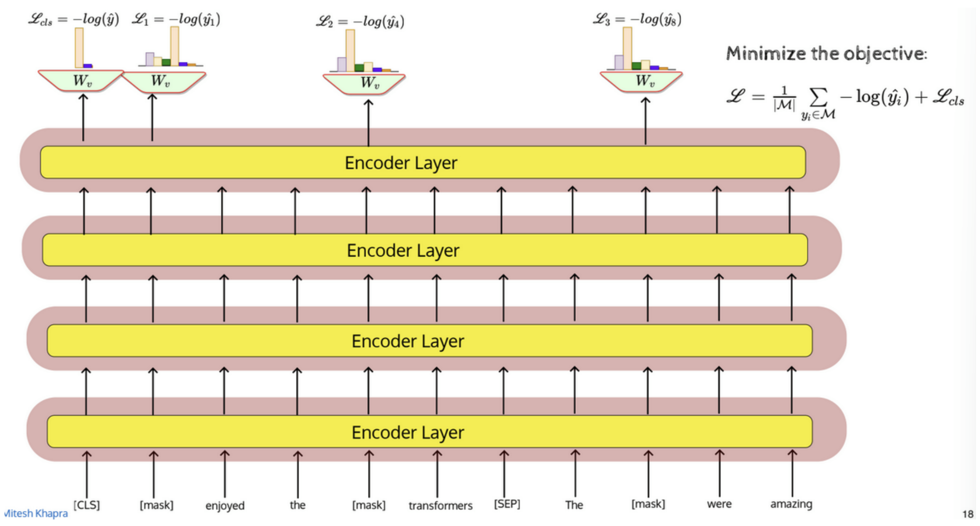


Figure 6: Next Sentence Prediction



$\mathscr{L}_{cls} = -log(\hat{y})$
$\mathscr{L}_1 = -log(\hat{y}_1)$
$\mathscr{L}_2 = -log(\hat{y}_4)$
$\mathscr{L}_3 = -log(\hat{y}_8)$
W_v
Minimize the objective:
$\mathscr{L} = \frac{1}{|\mathcal{M}|} \sum_{y_i \in \mathcal{M}} -\log(\hat{y}_i) + \mathscr{L}_{cls}$
Encoder Layer
[CLS]
[mask]
enjoyed
the
[mask]
transformers
[SEP]
The
[mask]
were
amazing
Mitesh Khapra
18

Figure 7: Combined MLM and NSP objectives

3.2 Adapting to Downstream Tasks

- BERT can be used as a feature extractor or we can directly fine-tune BERT
- **Feature Based Classification**: Take the final hidden representation (output of the last encoder) as a feature vector for the entire sentence. This representation is superior to merely concatenating representations (say from word2vec) of individual words.
- Finally, we can use any ML model, called the head, like logistic regression, naive bayes, or neural networks, for classification.
- All the parameters of BERT are frozen, and only the classification head is trained from scratch.
- **Fine-Tuning Classification**: Add a classification head. Initialize the parameters of the classification head randomly. Now, train the entire model, including the parameters of the pre-trained BERT for the new dataset.
- Note, however, that we do not mask words in the input sequence (the reason why we replaced 10% of masked words by random words during pre-training)

- It is observed that the model used in the classification head converges quickly with a less number of labeled training samples than the feature-based approach.
- Note that the final vector representation is the first vector of the output, corresponding to [CLS] token.
- **Extractive Question-Answering**: We give the question and a paragraph as input to the encoder. We need to make use of the final representations to find the start and end tokens. Take the representation corresponding to the paragraph only.
  One approach is to pose this as a classification problem. The classification head takes in all these final representations and returns the probability distribution for a token to be a start or end token.
  Let S denote a start vector of size h_i, then the probability that the ith word in the paragraph is the start token is
  $$s_i = \frac{\exp(S \cdot h_i)}{\sum_{j=1}^{25} \exp(S \cdot h_j)}$$
  Similarly, let E denote an end vector of size h_i
  $$e_i = \frac{\exp(E \cdot h_i)}{\sum_{j=1}^{25} \exp(E \cdot h_j)}$$
  Both S and E are learnable parameters
- It is possible that the end token index might be less than the start token index. In that case, return an empty string.

3.3 BERT Variations

- **SpanBERT**: Instead of masking tokens randomly, mask a contiguous span of random tokens, which is more natural for downstream tasks like Question-Answering.
- The span length is a random variable that follows the geometric distribution, which is skewed towards shorter spans.
- Dropped NSP objective, instead adds an auxiliary objective called Span Boundary Objective (SBO) that predicts the entire span of texts using the representation of (unmasked) span boundary tokens.
- **RoBERTa**: Proposed a set of Robust design choices for BERT, because training is an expensive process
  1. Training the model longer, with bigger batches (10 to 30 fold), over more data (almost 10-fold).
  2. Remove the next sequence prediction objective
  3. Training on longer sequences
  4. Dynamically changing the masking pattern applied to the training data.

  It stands for **R**obustly **O**ptimized **BERT** pre-training **A**pproach.
- **ALBERT-TinyBERT-DistillBERT**: The other aspect is to reduce the number of parameters in BERT using factorization of embedding matrix, weight sharing and knowledge distillation (KD) techniques. This is helpful for storing and running BERT on resource-constrained devices. The model size is reduced from 108MB to 12MB, and about 5 to 9 times speedup in inference time.
- **Electra**: Instead of masking tokens randomly. Replace the tokens randomly with the plausible alternatives sampled from a small generator network. Then the discriminator network is tasked to distinguish the replaced token from the real token. This improves performance even without increasing the train data size with comparatively less compute. The performance is comparable to that of RoBERTa and ALBERT (A Little BERT)

4 Tokenizers

4.1 Introduction

- Reference: On subword units by Kyle Chung
- Summary of Tokenizers by HF
- Language modeling involves computing probabilities over a sequence of tokens.
- This requires us to split the input text into tokens in a consistent way. A simple approach is to use whitespace to split the text.

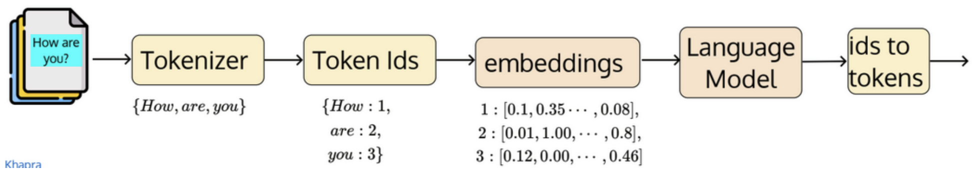


Figure 8: Training and Inference Pipeline

- The first step is to build (learn) the vocabulary ν that contains unique tokens (x_i).
- We can split the text into words using whitespace (called pre-tokenization) and add all unique words in the corpus to vocabulary.
- We also include special tokens such as $\langle$go$\rangle$, $\langle$stop$\rangle$, $\langle$mask$\rangle$, $\langle$sep$\rangle$, $\langle$cls$\rangle$, and others to the vocabulary based on the type of downstream tasks and the architecture (GPT/BERT) choice.
- Challenges in building a vocabulary
  1. What should be the size of vocabulary?
     Larger the size, larger the size of embedding matrix and greater the computation of softmax probabilities. What is the optimal size?
  2. Out-of-vocabulary
     If we limit the size of the vocabulary (say, 250K to 50K), then we need a mechanism to handle out-of-vocabulary (OOV) words. How do we handle them?
  3. Handling misspelled words in corpus
     Often, the corpus is built by scraping the web. There are chances of typo/spelling errors. Such erroneous words should not be considered as unique words.
  4. Open Vocabulary Problem
     A new word can be constructed (say, in agglutinative languages) by combining the existing words. The vocabulary, in principle, is infinite (that is, name, numbers,...) which makes a task like machine translation challenging.
- We want a Moderate-sized Vocabulary, to Efficiently handle unknown words during inference, and Be language agnostic
- Tokenization can be character level, sub-word level, and word level.
- **Sub-Word Level**: A middle ground between character-level and word-level tokenizers. The size of the vocabulary is carefully built based on the frequency of occurrence of sub-words.
  The most frequently occurring words are preserved as is, and rare words are split into sub-word units. The size of the vocabulary is moderate.
- Sub-word level tokenizers are preferred for LLMs

4.2 Byte Pair Encoding

- Reference: Neural Machine Translation of Rare Words with Subword Units
- **General Pre-processing**: First the text is normalized which involves operations such as treating cases, removing accents, eliminating multiple whitespace, handling HTML tags, etc.
- Splitting the text by a whitespace was traditionally called tokenization. However, when it is used with a sub-word tokenization algorithm, it is called **pre-tokenization**.

- Learn the vocabulary(training) using these words.
- Start with a dictionary that contains words and their count.
- Append a special symbol ⟨\w⟩ at the end of each word in the dictionary
- Set required number of merges (a hyperparameter)
- Initialize the character-frequency table (a base vocabulary)
- Get the frequency count for a pair of characters
- Merge pairs with maximum occurrence

Algorithm 1 Byte Pair Encoding

```
import re, collections

def get_statc(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols) - 1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>': 5, 'l o w e r </w>': 2,
'n e w e s t </w>': 6, 'w i d e s t </w>': 3}
num_merges = 10

for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
print(best)
```

- The algorithm offers a way to adjust the size of the vocabulary as a function of the number of merges.
- For a larger corpus, we often end up with vocabulary of size smaller than considering individual words as tokens.
- Languages such as Japanese and Korean are non-segmented. However, BPE requires space-separated words.
- In practice, we can use **language-specific morphology** based word segmenters such as Juman++ for Japanese.
- However, in the case of multilingual translation, having a language agnostic tokenizer is desirable.

4.3 Word Piece Encoding

- In BPE we merged a pair of tokens which has the highest frequency of occurrence.
- What if there are more than one pair that is occurring with the same frequency?
- Take the frequency of occurrence of individual tokens in the pair into account

$$score = \frac{count(\alpha, \beta)}{count(\alpha)count(\beta)}$$

- Now we can select a pair of tokens where the individual tokens are less frequent in the vocabulary.
- The Word Piece algorithm uses this score to merge the pairs.

4.4 Sentence Piece Encoding

- Reference: Rust implementation of Sentence piece by Guillaume
- For BPE and word piece, we start with a character-level vocab and keep merging until a desired vocabulary size is reached.
- We can do reverse as well, start with word level vocab and keep eliminating words until a desired vocabulary size is reached.
- BPE is greedy and deterministic, we can use BPE-Dropout to make it stochastic.
- The probabilistic approach is to find the subword sequence $x^* \in \{x_1, x_2, \ldots, x_k\}$ that maximizes the likelihood of the word X
- The word X in sentence piece means a sequence of characters or words (without spaces)
- Therefore, it can be applied to languages (like Chinese and Japanese) that do not use any word delimiters in a sentence.
- Let x denote a subword sequence of length n, $x = (x_1, x_2, \ldots, x_n)$, then the probability of the subword sequence (with unigram LM) is simply

$$P(x) = \prod_{i=1}^{n} P(x_i) \qquad \sum_{x \in \nu} p(x) = 1$$

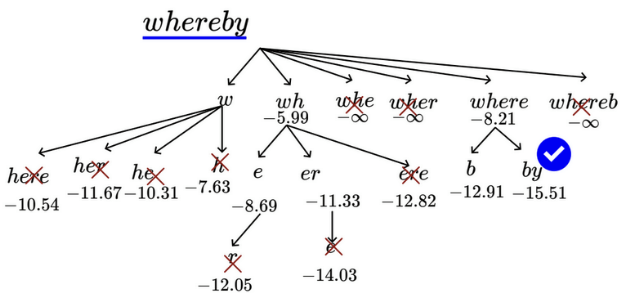
- The objective is to find the subword sequence for the input sequence X (from all possible segmentation candidates od $S(X)$) that maximizes the log likelihood of the sequence

$$x^* = \arg\max_{x \in S(X)} \log P(x)$$

- We can use Viterbi decoding to find x^*.
- Then, for all sequences in the dataset D, we define the likelihood function as

$$L = \sum_{s=1}^{|D|} \log(P(X^s)) = \sum_{s=1}^{|D|} \log \left(\sum_{x \in S(X^s)} P(x) \right)$$

- The subwords $p(x_i)$ are hidden (latent) variables
- Therefore, given the vocabulary ν, the Expectation-Maximization (EM) algorithm could be used to maximize the likelihood
- Unigram model favors the segmentation with the least number of subwords.
- In practice, we use Viterbi decoding to find x^* instead of enumerating all possible segments
- Set the desired vocabulary size
  1. Construct a reasonably large seed vocabulary using BPE or Extended Suffix Array algorithm
  2. **E-Step**: Estimate the probability for every token in the given vocabulary using frequency counts in the training corpus
  3. **M-Step**: Use Viterbi algorithm to segment the corpus and return optimal segments that maximizes the log likelihood
  4. Compute the likelihood for each new sub-word from optimal segments
  5. Shrink the vocabulary size by removing top $x\%$ of sub-words that have the smallest likelihood
  6. Repeat step 2 to 5 until the desired vocabulary size is reached



Token	log(p(x))
b	-4.7
e	-2.7
h	-3.34
r	-3.36
w	-4.29
wh	-5.99
er	-5.34
where	-8.21
by	-7.34
he	-6.02
ere	-6.83
here	-7.84
her	-7.38
re	-6.13

Figure 9: Viterbi Decoding Example

- Another algorithm that is related to sentence piece is Morfessor which uses Maximum a posterior (MAP) Estimation as an objective function that allows us to incorporate a prior.
- In general, Morfessor allows us to incorporate the linguistically motivated a prior knowledge into the tokenization procedure. Also, allows us to add a small annotated data while training the model.

5 Architectural Choices in Transformers

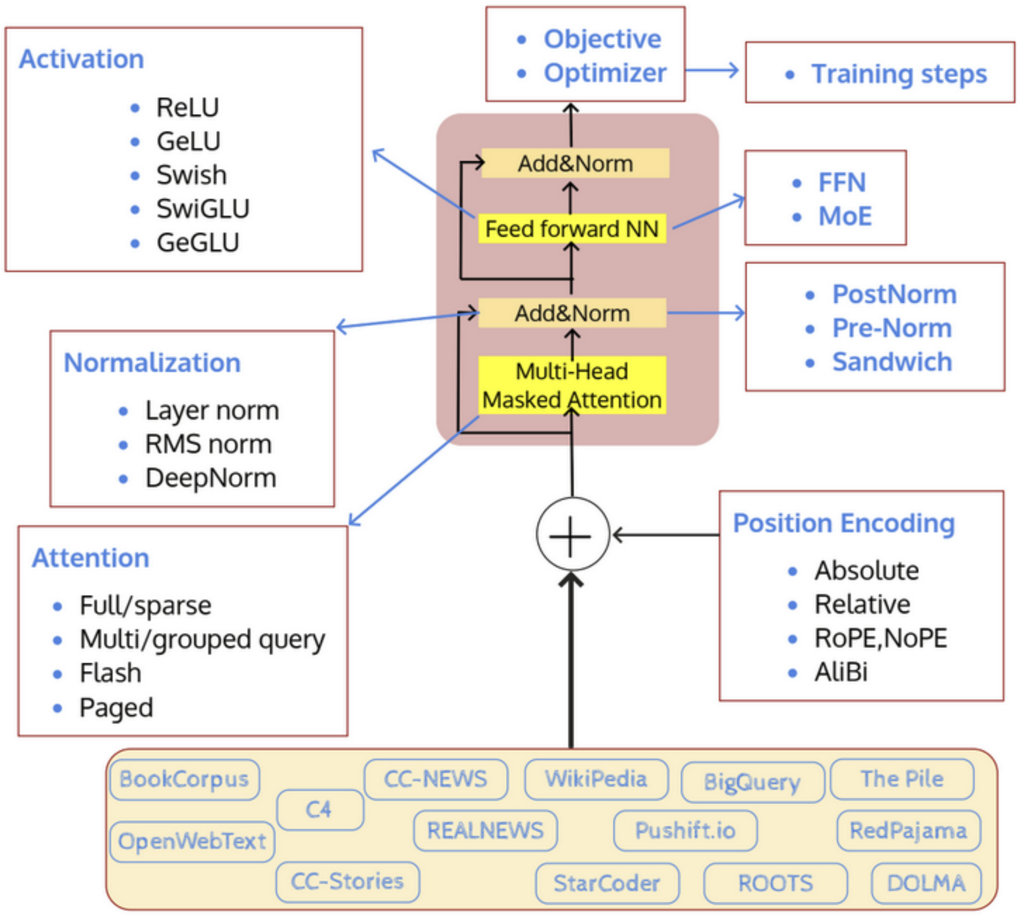


Figure 10: Unifying Diagram

5.1 From BERT to BART

- Reference: BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension
- So far, we have learned two approaches with two different objectives for language modeling.

- BERT is good at comprehension tasks like question answering because of its bidirectional nature
- GPT is good at text generation because of its unidirectional nature
- Why not take the best of both worlds?
- **BART**: Bidirectional AutoRegressive Transformer, a encoder-decoder model
- BART is a vanilla transformer architecture with GELU activation function
- The input sequence is corrupted in multiple ways: random masking, token deletion, document rotation, and so on
- The encoder takes in the corrupted sequence
- The decoder predicts the entire sequence and computes the loss (as in the case of GPT) over the entire sequence
- One can think of this as a denoising autoencoder.

5.2 GPT2: Promprint Large Language Models

- Reference: Language Models are Few-Shot Learners
- Using any of these pre-trained models for downstream tasks requires us to independently fine-tune the parameters of the model for each task (objective might differ for each downstream task)
- That is, we make a copy of the pre-trained model for each task and fine-tune it on the dataset specific to that task.
- However, fine-tuning large models is costly, as it still requires thousands of samples to obtain good performance in a downstream task.
- Some tasks may not have enough labeled samples.
- Moreover, this is not how humans adapt to different tasks one they understand the language.
- We can give them a book to read and ”prompt” them to summarize it or find an answer for a specific question. That is, we do not need ”supervised fine-tuning” at all (for most of the tasks).
- In a nutshell, we want a single model that learns to do multiple tasks with zero to a few examples.
- Note that the prompts (or instructions) are words. Therefore, we just need to change the single task (LM) formulation to multi-task

  $$p(output|input) \rightarrow p(output|input, task)$$

  where the task is just an instruction in plain words that is prepended (appended) to the input sequence during inference
- Surprisingly, this induces a model to output a task-specific response for the same input.
- To get a good performance, we need to scale up both the model size and the data size.
- A new dataset called WebText was created by crawling outbound links (with at least 3 karma) from Reddit, excluding links to Wikipedia. This is to ensure the diversity and quality of data.
- This outperformed other language models trained on domains like Wiki and books, without using domain-specific training datasets.
- However, the performance of the model on tasks like question answering, reading comprehension, summarization, and translation is far from SOTA models specifically fine-tuned for these tasks.
- Moreover, the 1.5B parameters model still underfits the training set.
- This work established that ”Large Language models are unsupervised multitask learners”
- Reference: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,2019
- Reference: A Case Study on the Colossal Clean Crawled Corpus
- Reference: Parameter-efficient transfer learning for NLP
- Reference: Presentatation by Collin Raffel

- After extensive experimentation, authors made the following design choices for T5 model
  1. Architecture: Encoder-Decoder
  2. Objective: Denoising with span corruption
  3. Pre-training Dataset: C4
  4. Training strategy: Multi-task Pre-training
  5. Model size: Bigger the better
  6. Training steps: Longer the better

6 Pre-Training and Data Processing

6.1 Motivation for Scaling the data size

- **Scaling Law**: The more the data, the better the model
- The study on T5 demonstrated that the model overfits if the data is repeated for many epochs during training. This suggests that we should scale the data size.
- The other factor that motivated scaling the data size was the study on Scaling Laws (Scaling Laws for Neural Language Models)
- Given the number of non-embedding parameters N and the size of the dataset D (in tokens), the test loss

$$L(N, D) = \left[\left(\frac{N_c}{N} \right)^{\frac{\alpha_N}{\alpha_D}} + \frac{D_c}{D} \right]^{\alpha_D}$$

$$N_c = 8.8 \times 10^{13} : 88 \text{ Trillion parameters} \qquad \alpha_N = 0.076$$

$$D_c = 5,4 \times 10^{13} : 54 \text{ Trillion tokens} \qquad \alpha_D = 0.095$$

- Increasing the Dataset size lowers the test loss
- Reference: Language Models are Few-Shot Learners
- Diversity of data and data quality is super important.
- Open source data is good enough
- Detailed information about the data-cleaning pipeline is available publicly for C4 from T5, The Pile, Falcon, ROOTS, Redpajama-v1/v2, DOLMA, Sangraha

6.2 Data Handling

- Often, datasets used in pre-training large models lack proper documentation. Technical reports only provide very basic information
- Almost all the pre-training datasets were constructed by scraping/crawling the web.
- One can write a custom HTML scraper or use Common Crawl (CC) for fetching the data.
- Each month they release a snapshot of the web that contains about 3 to 5 billion web pages
- This raw data contains duplicate, toxic, low-quality content and also personally identifiable information.
- The raw CC data is available in three types
  1. WARC (Web ARChive, 86 TiB)
  2. WAT (Web Archive Transformation, 21 TiB)
  3. WET (WARC Encapsulated Text, plain texts, 8.69 TB). Informally, Web Extracted Text
- The pipeline takes in the raw data in any one of these formats and produces clean text.
- To build a large dataset, researchers often take CC snapshots of multiple months or years
- Therefore, it is a challenging task to build an automatic pipeline that takes in the raw text and outputs clean text
- It is also challenging to directly measure how clean the output text is.

- To get high-quality text from the raw data, the pipeline requires mechanisms
    1. to detect the language of the web pages (FastText, langdetect, pycld2, IndicLID, or combination of these)
    2. to de-duplicate the content (line, paragraph and document level) (Similarity measure using hashing/ml techniques)
    3. to detect toxicity level (bad words, hate speech, stereotypes, ...) of the content in the pages (Simple heuristics or LM model, Perspective AI)
    4. to detect the quality of the content in the pages (LM model trained with reference text like Wikipedia or books, simple heuristics)
    5. to detect and remove Personally Identifiable Information (PII) in the pages (Regex Patterns)
- **Types of De-duplication**: Exact duplication (Suffix array), Fuzzy (near) duplication (MinHash, SimHash, Locality Sensitive Hash (LSH))
- **CCNet Pipeline**
    1. Designed for extracting high-quality monolingual pre-training data
    2. This is also helpful for low-resource languages as datasets like Bookcorpus and Wikipedia are only for English
    3. Even if we wish to use Wikipedia dumps for other languages, the size won't be sufficient
    4. Therefore, we need to use CC dumps (years of dumps). This is a compute-intensive task and requires massive parallelization with carefully written code to execute them
    5. Uses paragraph-level de-duplication.
    6. Divide each webpage into three portions: head, middle, and tail. Obtain quality score for each portion using a pre-trained language model on a targeted domain.
    7. Remove low-quality pages based on a threshold.

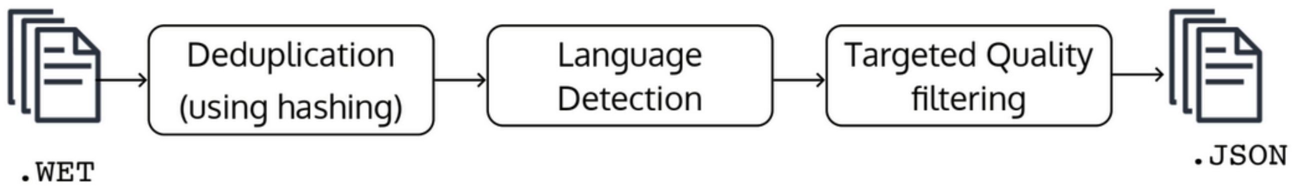


Figure 11: CCNet Pipeline

- **Colossal Clean Common Crawl (C4) Pipeline**: It is an extension of CCNet with additional filtering heuristics. We can get diverse data such as news, stories, webtext by domain filtering.

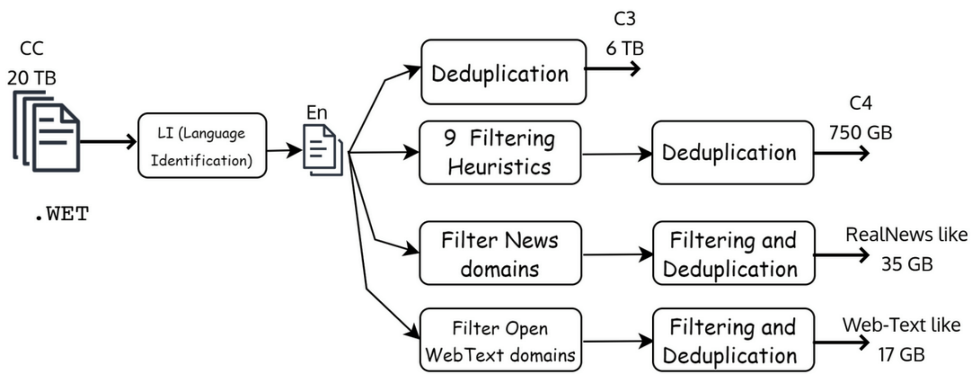


Figure 12: C4 Pipeline

- They released the clean dataset named C4 in the public domain.
- In general, all pre-processing data pipelines follow CCNet pipeline with modifications in the components of the pipeline.

- Pile-CC uses the WARC files instead of WET, which contain raw HTML page,s then applies filtering (using jusText) to extract text content

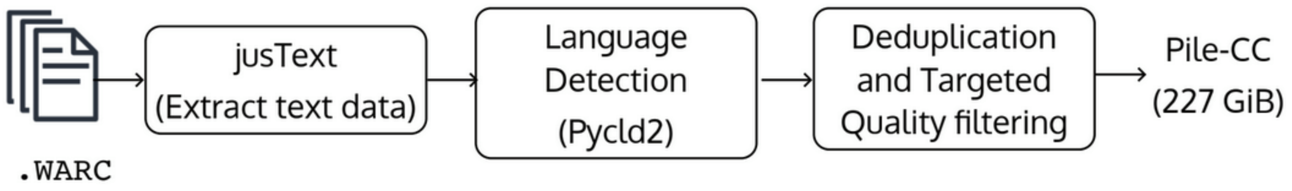

.WARC
jusText (Extract text data)
Language Detection (Pycld2)
Deduplication and Targeted Quality filtering
Pile-CC (227 GiB)

Figure 13: pile-CC Pipeline

It uses fuzzy de-duplication over exact de-duplication algorithm as in C4

- Significant duplicates in CC are exact duplicates. Therefore, one can use fuzzy de-duplication first, followed by exact de-duplication.
- **RefinedWeb (RW)**
  1. 50% of CC was removed after language identification.
  2. 24% of RW-Raw was removed after Quality filtering
  3. 12% of RW-filtered was removed after de-duplication
  4. In total, this stringent approach removes almost 90% of data from a single snapshot of the common crawl
  5. It uses both fuzzy and exact de-duplication techniques.

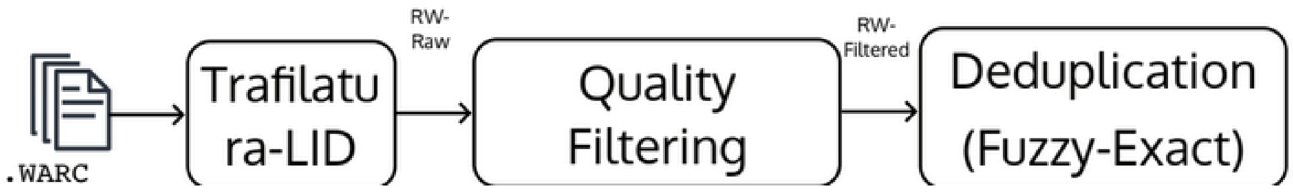


Figure 14: R

- Perfect De-duplication is difficult
- Duplication allows the model to memorize the training data, gives a wrong test performance, susceptible to privacy risks.
- For a 1B parameters model, a hundred duplicates are harmful.
- At 175B, even a few duplicates could have a disproportionate effect.
- All pipelines are leaky
- The recent RefinedWeb pipeline demonstrates that sourcing large-scale, high-quality data from web alone is sufficient to get a better performance.
- Almost all LLM models predominantly use web data.

7 Fast Attention and Efficient Mechanisms

7.1 Complexity of Attention Mechanism

- Enabling the models to attend to longer sequences is crucial.
- One of the primary factors is the computational complexity of the self-attention mechanism.
- To compute QK^T, we require $O(T^2 d)$ operations, where T is the context window size and d is the size of the projected embeddings.
- The number of tokens T is typically greater than embedding dimension d.
- Computational complexity of $softmax$ is $O(T^2)$

- Finally, the attention score matrix A is multiplies with the value matrix V. This has a complexity of $O(T^2d)$
- Notations
  1. B: Batch size
  2. $|\nu|$: Number of tokens in the vocabulary
  3. T: Context (sequence) length
  4. d_{ff}: Dimension of feed-forward layer
  5. d_{model}: Dimension of input embedding, $d_{att} = 512 = \frac{d_{ff}}{4}$
  6. n_h: Number of attention heads
  7. N: Number of parameters in the model
- For a single sample, the memory requirement is

$$mha = (Q_{T\times d} + K^T_{d\times T} + V_{T\times d})n_h = 3n_h(T \times d) = 3T \times d_{model}$$

$$Z = QK^T = Bn_hT^2$$

$$A = softmax(QK^T) = Bn_hT^2$$

  For a batch of B samples $= 3BTd_{model}$
- Memory scales up linearly with respect to batch size B and quadratically with respect to the length of the context window T.

7.2 Fast Attention Mechanisms

- A study on BERT in NLP tasks concluded that the neighboring inner products are extremely important in self-attention and not all heads attend to all tokens.
- This gives rise to a plethora of approaches that approximate the full attention.

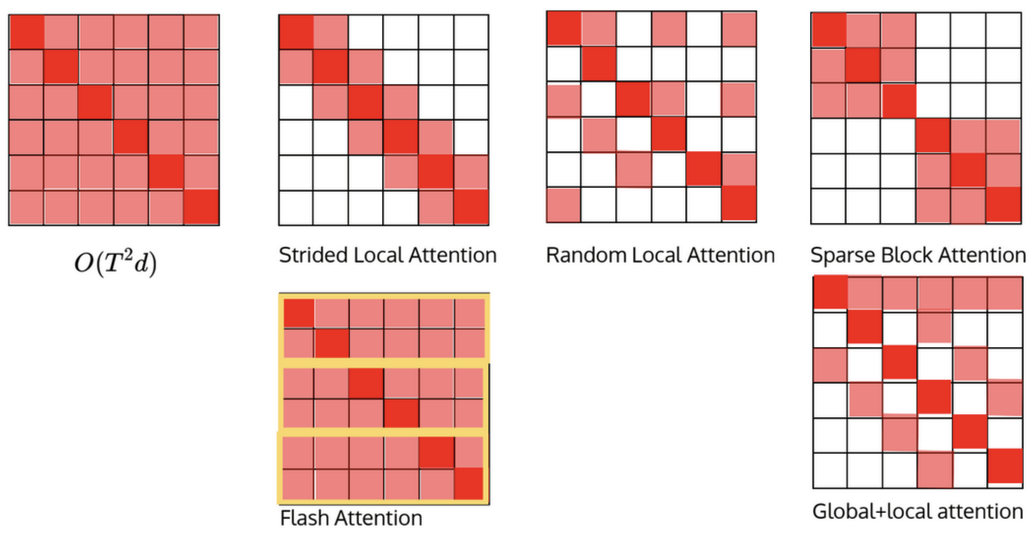


Figure 15: Fast Attention Approaches

- In full attention every word attends to every other word in the input sequence.
- One way to make it sparse is by attending a subset of words instead of all words

$$O(c \times T \times d) \text{ where } c < T$$

- **Strided Local Attention**: Every word attends to a window of c words ($\lfloor\frac{c}{2}\rfloor$ to the left and $\lfloor\frac{c}{2}\rfloor$ words to the right). Linear computation complexity $O(cTd)$.
- We could vary the size of the window for each layer in the model such that the top-most layer uses the full (global) attention.

- **Dilated Attention**: We could also use dilated attention (like dilated convolutions) by using different dilation rates in different attention heads. Note that, the computational complexity is the same as Strided Local Attention.
- **Global+Local Attention**: A few words are global, every other word attend to a window of c words. Tokens like $[CLS]$ require full attention to get good performance in downstream tasks. We can manually set tokens that require global attention based on the task. Time complexity reduces to $O((cT + T)d)$
- **Global+Random+Local Attention**: The BIGBIRD model proposes adding random tokens. Time complexity reduces to $O((cT + T + T)d)$
- **Local Block Attention**: Another way is to divide the sequence into n blocks and do the computations independently. Typically, $n = 2, 3, 4$, therefore, for a context length of 512, the block sizes will be $256, 128, 64$. These are reasonable context sizes for the majority of NLP tasks. We could also use different patterns in each attention head/layer. We can permute these blocks in each attention head so that it can effectively capture long-range dependencies. Time complexity reduces to $O(\frac{T^2 d}{n})$
  Consider the permuation, $\pi = perm(0, 1, 2, \cdots, n-1)$
  Let the kth element of π be $\pi(k)$. We define the masking matrix M of size $T \times T$

$$M_{ij} = \begin{cases} 1 & \text{if } \pi(\lfloor \frac{in}{T} \rfloor) = \lfloor \frac{jn}{T} \rfloor \\ 0 & \text{otherwise} \end{cases}$$

  Block attention is given by

$$Z = softmax(QK^T \odot M)$$

  Using these we can compute Block Attention as follows

$$\text{Block-wise Attention}(Q, K, V, M) = \begin{bmatrix} softmax(Q_0 K^T_{\pi(0)})V_{\pi(0)} \\ \vdots \\ softmax(Q_n K^T_{\pi(n-1)})V_{\pi(n-1)} \end{bmatrix} = O(\frac{T}{n} \times \frac{T}{n} \times n \times d)$$

- We can extend the same concept to multi-head attention by allowing each head to use a different masking matrix. This would be called **Blocked multi-head attention**.
- Blockwise sparsity captures both local and long-distance dependencies in a memory-efficient way using different permutations.
- Using blockwise sparse attention reduces the training time significantly for longer sequences.
- Note, however, that the perplexity score increases as the number of blocks n increases.
- There exists a low-rank approximation of QK^T and we can let the network learn it
  Suppose $Q, K, V \in \mathbb{R}^{T \times d}$
  Introduce two learnable linear projection matrices

$$\begin{aligned} &E, F \in \mathbb{R}^{k \times T} \\ &A = softmax(\frac{Q(EK)^T}{\sqrt{d}}) \in \mathbb{R}^{T \times k} \\ &O = AFV \end{aligned}$$

  The computational complexity is $O(kTd)$, where k is the rank and can be set according to the error ϵ

7.3 Fast Interference

- At each time step, we compute the key-values for all previous queries as well.
- This is a waste of compute and moreover increases the latency.
- Moreover, processing an LLM request can be $10\times$ more expensive than a traditional keyword query.
- **KV Caching**: In the first time step, we start with a query, compute its key-value pair for the given input token when running the model in autoregressive mode. Store the key value in cache (usually GPU RAM, which has HBM).
- For the new query q_2 in time step 2, we reload k_1, v_1 from HBM and compute k_2, v_2.
- In this way, we trade off the memory for compute and the compute scales linearly $O(n)$

- Since KV-caching uses GPU RAM where the model weights are stored, we cannot let the memory for KV-caching grow indefinitely.

$$\text{KV cache per token (in B)} = 2 \times num\_layers \times (num\_heads \times dim\_head) \times precision\_in\_bytes$$

- A practical solution is to drop the keys and values of tokens that occurred in the distant past (act similarly to local windowed attention). The other approach is to introduce model-level modifications
- **Multi-Query Attention (MQA)**: Select one hyperparameter out of three hyperparameters that we could exploit in the formula to reduce the memory requirement. Share all the key-value pairs across the heads, setting $num\_heads = 1$. This greatly reduces the cache memory. However, the performance degrades significantly.
- **Grouped-Query Attention (GQA)**: A middle ground between Multi-Head and Multi-Query attention, where a group of queries share key-value pairs that result in performance close to MHA.
- **Paged Attention**: Typically, fixes and contiguous memory is reserved for KV-caching. However, we do not know the length of the sequence that the model generates a priori. The memory during the generation process is wasted if the model generates a sequence that is much smaller than the pre-fixed length. A solution is to use a block of non-contiguous memory locations, which is what Paged Attention does.
- References
  1. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness
  2. Fast Transformer Decoding: One Write-Head is All You Need
  3. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints
  4. Efficient Memory Management for Large Language Model Serving with PagedAttention

8 Position Encoding Methods

8.1 Limitations of Absolute Position Encoding

- Positional encoding helps to capture the order of tokens in the input sequences.
- One approach is to encode the absolute positions, we have done this by adding the position vector p_i with the word embedding x_i

  $$h_i = x_i + p_i$$

  The values for the position vector are fixed.
- Why note relative positions? Why not multiply? Why not add PE with the attention score? Why not learn it?
- This is still an active area of research alongside attention mechanisms
- All prominent models like BERT, RoBERTa, GPT-X, BART and OPT use learned APE.
- It is hypothesized that APE does learn to attend to tokens by relative positions.
- Ideally, we expect the model prediction to be consistent for the given input sentence with (learned) absolute position encoding.
- What if we change the starting position of the input by shifting by k units to the right, i.e., the embeddings of the first token added to $(0 + k)$.
- If the model learned to attend to relative positions, then the prediction should not change. Therefore, the performance should not degrade.
- However, it was observed that the performance degrades as k increases.
- This motivated the need to explicitly encode relative positions.
- The other limitation of APE is its inability to extrapolate to a longer sequence beyond the context length of a model.
- Closely related to the above is the "length generalization" ability to extrapolate from shorter training sequences to longer test ones, within the context length of a model.

8.2 Relative Position Encoding Methods

- RPE encodes pairwise relationships between tokens.
- The relative position of the word "is" 0 with respect to the position of the word itself and differs with respect to the position of the words surrounding it.
- We have T relative positions for each token in a sequence of length T and, in general, $2T-1$ embeddings to encode all relative positions, whereas APE requires only T embeddings.
- With a naive implementation, for each token embedding x_i, we need a way to combine (say, addition) T relative position embeddings. For $T=8$

$$h_3 = x_3 + (p_{-3} + p_{-2} + \cdots + p_4)$$

$$h_i = x_i + \sum_{j=1}^{T-1} p_{j-i}$$

- In general, this requires T^2 additions instead of T additions as in APE.
- Let's take a look at the decomposition of the pre-attention computation $e_{ij} = q_i k_j^T$

$$\begin{aligned}
q_i &= h_i W_Q = (x_i + p_i) W_Q \\
k_j &= h_j W_K = (x_j + p_j) W_K \\
e_{ij} &= (x_i + p_i) W_Q W_K^T (x_j + p_j)^T \\
&= (x_i W_Q + p_i W_Q)(W_K^T x_j^T + W_K^T p_j^T) \\
&= \underbrace{x_i W_Q W_K^T x_j^T}_{\text{score without PE}} + \underbrace{x_i W_Q W_K^T p_j^T + p_i W_Q W_K^T x_j^T}_{\text{correlation b/w word and position}} + \underbrace{p_i W_Q W_K^T p_j^T}_{\text{constant from PE}}
\end{aligned}$$

- PE just adds a constant to the pre-attention score. Therefore, we can inject position information in the attention layer directly by adjusting the pre-attention score e_{ij}
- Adding APE to the input embeddings introduces position artefacts.
- Correlation term is uniformly distributed, that is, no correlation.
- We can do better by modifying the pre-attention score e_{ij} directly.
- Adding a constant to the attention score is the simplest approach.

$$e_{ij} = x_i W_Q (x_j W_K + p_{ij}^K)^T \quad \text{where, } p_{ij}^K \in \mathbb{R}^d$$

  p_{ij}^K is also called position bias

- This however does not help the model to generalize to sequence lengths not seen during training because the size of position embedding matrix still depends on T
- Suppose the relative distance information is clipped beyond a certain distance k as follows

$$\begin{aligned}
p_{ij}^K &= w_{clip(j-i,k)}^K \\
clip(j-i,k) &= \max(-k, \min(j-i,k))
\end{aligned}$$

  Then, the size of the position embedding matrix is $(2k+1)$, making it independent of T.

- It is also empirically observed that clipping the distance does not hurt the performance.
- We can compute the attention score as usual

$$\alpha_{ij} = softmax(e_{ij})$$

  Finally, add the relative position information to value vectors

$$z_i = \sum_j \alpha_{ij}(x_j W_V + p_{ij}^V)$$

- The position embeddings p_{ij}^K, p_{ij}^V are shared across heads (not across layers)

- Replacing the APE in vanilla transformer architecture by RPE with $k = 16$ leads to better performance.
- **Transformer-XL**: Recursively extend context

$$e_{ij} = x_i W_Q W_K^T x_j^T + x_i W_Q W_K^T R_{j-1}^T + u W_Q W_K^T x_j^T + v W_Q W_K^T R_{j-i}^T$$

  R_{i-j}^T is the relative distance between the position i and j computed from sinusoidal function (non-learnable)
  u, v are learnable vectors

- **T5-bias**: All that is required is adding a constant

$$e_{ij} = x_i W_Q W_K^T x_j^T + r_{j-i}$$

  $r_{j-i} \in \mathbb{R}$ are learnable scalars, shared across layers. Maximum relative distance is clipped to 128. It also greatly reduces the number of learnable parameters when the model size is scaled.

- **Rotary Position Embedding (RoPE)**: We can generalize the above encoding as $f_q(x_i, i) = x_i W_Q$, $f_k(x_j, j) = x_j W_K$

$$\langle f_q(x_i, i), f_k(x_j, j)\rangle = x_i W_Q W_K^T x_j^T$$

  However, we want the inner product between token embeddings to encode the relative position information. That is we want it to be

$$\langle f_q(x_i, i), f_k(x_j, j)\rangle = g(x_i, x_j, j - i)$$

  Let us assume the following functional form

$$f_q(x_m, m) = x_m W_Q e^{jm\theta} \qquad f_k(x_n, n) = x_n W_K e^{jn\theta}$$

  Here, we multiply a (complex) constant to the affine-transformed word embedding.

$$g(x_i, x_j, j - i) = Re[(x_m W_Q)(x_n W_K)^T e^{j(m-n)\theta}]$$

  θ is a non-zero constant
  Essentially, x_m is rotated by $m\theta$ and x_n is rotated by $n\theta$
  The inner product results in

$$g(x_i, x_j, j - i) = x_m^T W_Q^T \begin{bmatrix} \cos((m-n)\theta) & \sin((m-n)\theta) \\ -\sin((m-n)\theta) & \cos((m-n)\theta) \end{bmatrix} W_K x_n$$

  We can generalize this to $x_m \in \mathbb{R}^d$ as follows

$$f_q(x_m, m) = R_{\Theta,m}^d W_Q x_m$$

  where

$$R_{\Theta,m}^d = \begin{bmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \cdots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \cdots & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{bmatrix}$$

$$\Theta = \{\theta_i = 10000^{-\frac{2(i-1)}{d}}, i \in (1, 2, \cdots, d/2)\}$$

  It is similar to the one we used in sinusoidal embedding
  Then, the pre-attention score is computed by encoding the relative position as follows

$$\begin{aligned} q_m^T k_n &= (R_{\Theta,m}^d W_Q x_m)^T (R_{\Theta,n}^d W_K x_n) \leftarrow \text{APE of } m, n \\ &= x_m^T W_Q^T (R_{\Theta,m}^d)^T (R_{\Theta,n}^d) W_K x_n \\ &= x_m^T W_Q^T R_{\Theta,m-n}^d W_K x_n \leftarrow \text{RPE of } m - n \\ \text{where } R_{\Theta,m-n}^T &= (R_{\Theta,m}^d)^T (R_{\Theta,n}^d) \end{aligned}$$

- Unlike sinusoidal APE, RoPE injects the positional information at every layer of the model and the position embedding is not injected into the value vectors.

8.3 Length Generalization

- We define length generalization as the ability of a model to extrapolate to longer sequences than the ones it has seen during training.
- **Attention with Linear Bias (ALiBi)**: Suppose we train a causal language model with $T = 512$ tokens. During inference, we validate the model by allowing the model to predict next tokens up to T_{valid} tokens. Perplexity score is used to measure the quality of predictions. T_{valid} is varied from 512 to 16000 tokens. We repeat the experiment only by changing the position encoding method and keeping the other aspects as is.
- Increasing the training context window helps T5 Bias RPE more than rotary and sinusoidal APE. This suggests that modifying T5-bias might improve the performance.
- We can hard-code r_{j-i} in T5-bias RPE to increase the inference speed. Simply add a constant (negative bias) to the attention score based on its position.

  $$softmax(q_iK^T + m(-(i-1), \cdots, -2, -1, 0))$$

  m is a head-specific scalar that follows a geometric progression. For $n = 8$ heads, $m_i = \frac{1}{2^i}$, $i = 1, \dots, n$.
- Since it is RPE, position information is added at every layer of the model.
- It is called linear because the bias value increases linearly with respect to the distance.
- We can observe that the constant value added for nearby tokens is larger $(0, -1, -2, ...)$ than the one added for distant tokens say $(-128, -129, ...)$. This effectively acts as a local windowed attention that enables ALiBi PE to predict the next token effectively from the recent past k tokens.
- This works remarkable well even for $T_{valid} = 16K$
- Subsequent studies generalize ALiBi ,such as KERPLE(Kernelized Relative Position Embedding for Length Extrapolation) and improved RoPE, such as xPOS.
- A study NoPos conjectures that using a causal mask implicitly encodes the position information.
- Perplexity measure may not be suitable for all downstream tasks like sorting, addition, summation, reversing, etc.
- Keeping these two observations in mind, one could extend the study on length generalization ability of decoder-only models, to other downstream tasks.
- One study considered sinusoidal APE, RoPE, T5-bias, No Position Encoding (NoPE).
- It is shown theoretically that NoPE (i.e., decoder-only models)
  1. can represent both absolute and relative PEs
  2. learns to use relative PE in practice
- Despite all these studies, length generalization remains a challenge for transformer-based models.
- Is there an inherent limitation in the Transformers design preventing effective length generalization?
- A recent study "Transformers Can Achieve Length Generalization But Not Robustly" suggest so.