Introduction to Large Language Models

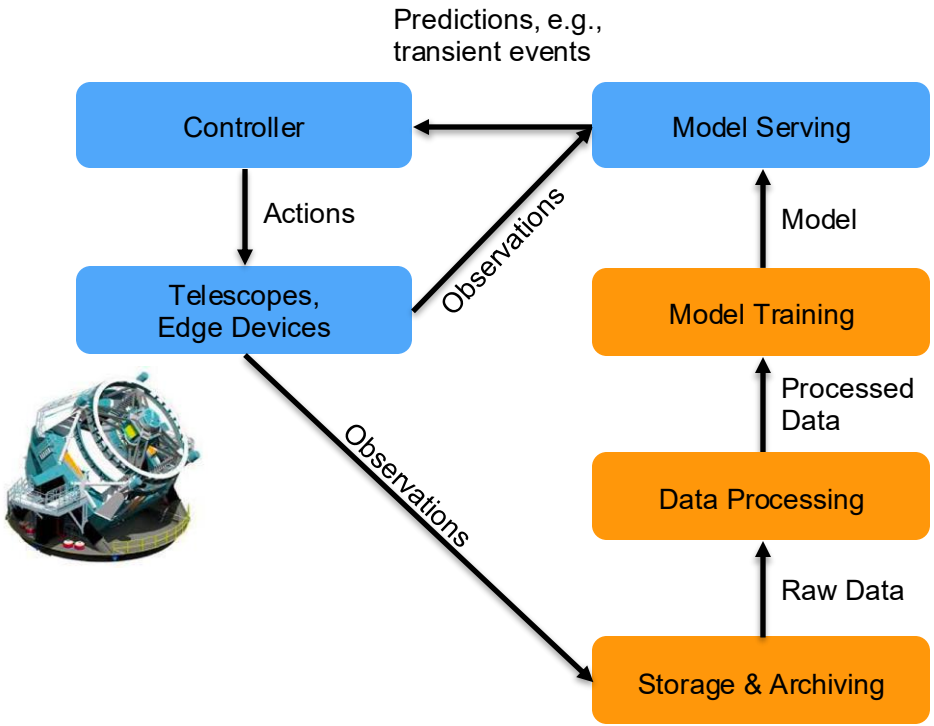
April 29, 2025

Zhao Zhang

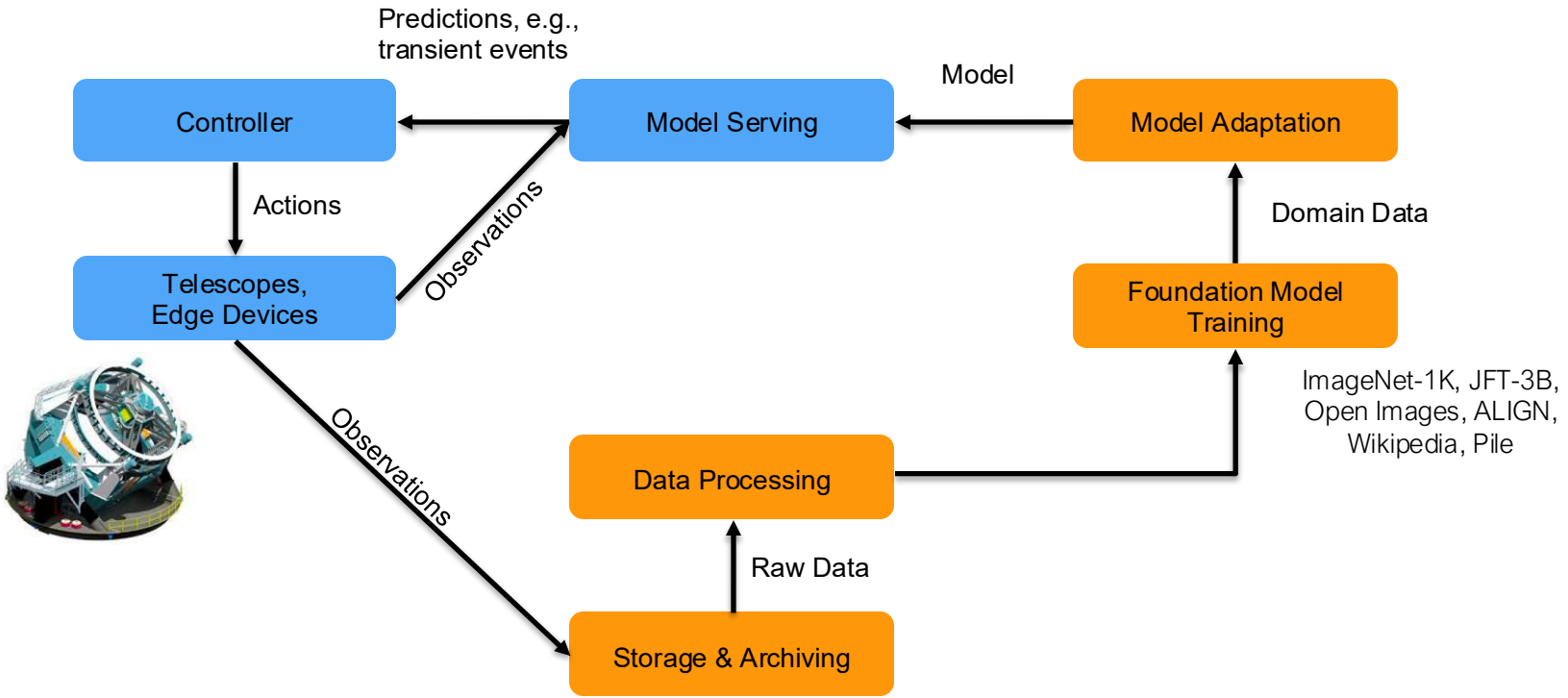
Overview

1. Using LLMs in Science and Engineering
2. High-level overview of the workflow of GPT2 model.
3. Tokenization
4. Layers in GPT-2 model
    a. Embedding layer
    b. Positional encoding layer
    c. Decoder layer
        i. Layernorm
        ii. Multi-head self-attention mechanism
        iii. MLP (or feedforward layer)
5. Hands-on with OpenFold

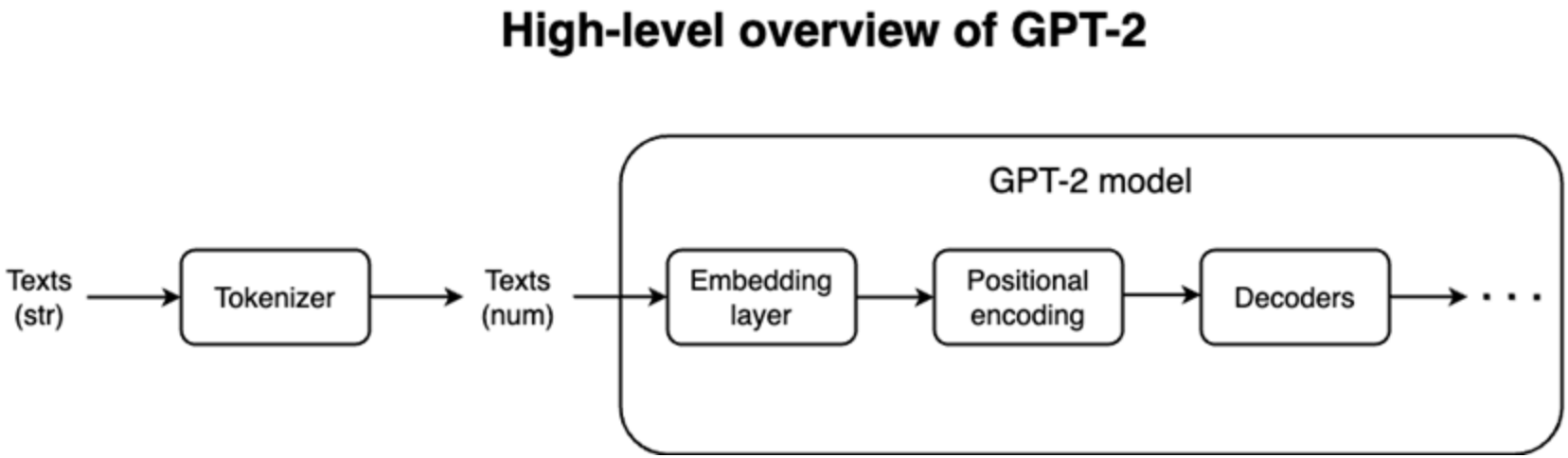
ML/DL in Science not So Long Ago



ML/DL in Science not So Long Ago



High-level overview



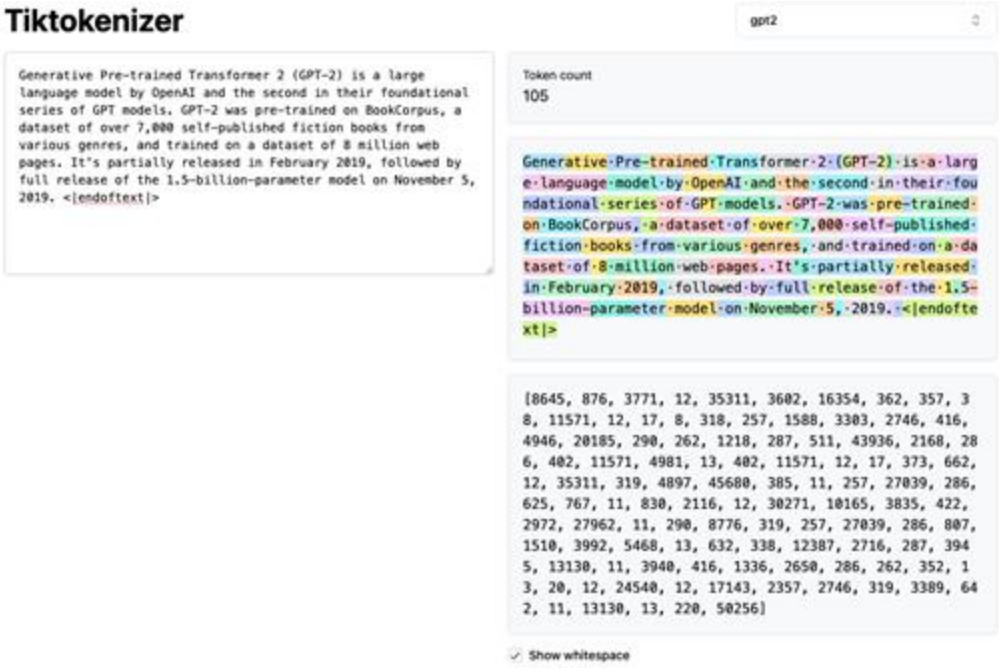
Tokenization

Tokenizer

- Why do we need a tokenizer?
  - ML/DL can only handle tensors.
  - Tokenizer converts a string of words into many numbers.
- Tokenizer is not part of the GPT-2 model.
  - **But it is very important!**
  - It determines how the model sees the strings, which can cause significant differences on model's inference performance.
- Tokenizer is trained from the pretraining dataset.
  - So, different LLMs have their own tokenizer.

Tokenization

- One tokenizer has two parts
  - Encoder and decoder.
  - Encoder converts string to numbers.
  - Decoder does the reverse ops.
- The simplest tokenizer
  - Encode every char in a string into UTF-8 format.
  - However, this is computationally intractable because we have attention mechanism which has O(n^3) time complexity.
- To try and tokenize different texts using different tokenizers: [link]



GPT-2’s tokenization strategy

- **Byte Pair Encoding (BPE)** is the core algorithm behind GPT-2 tokenizer.
- The source code can be found here.
- The insight is very simple: Continue merging pairs of UTF-8 encoded bytes.
- Here is an example:
  - Suppose we have already converted every char in a string into its byte representation.
    - For example, “aaabdaaabac” -> [97, 97, 97, 98, 100, 97, 97, 97, 98, 97, 99].
  - Then we group every char with its right adjacent char.
    - [(aa),(aa),(ab),(bd),(da),(aa),(aa),(ab),(ba),(ac)]
    - [(97,97),(97,97),(97,98),(98,100),(100,97),(97,97),(97,97),(97,98),(98,97),(97,99)]
  - We find the most frequent pair, which in this case, is (aa).
  - Every (aa) pair is replaced by a new special char, say “Z”.
    - “aaabdaaabac” -> “ZabdZabac”.
    - We give index 256 (because UTF-8 encoded value ranges from 0 to 255) to “Z” and store this info into a lookup table.
  - Then we continue the steps above with the new str.
    - Every step will create a new index, and we can choose when to stop.

GPT-2 tokenizer details

- GPT-2 tokenizer works at the word-level, not the string-level.
  - I.e. OpenAI team first use regex operations to break a string into many word-like chunks, and then perform encoding on these chunks.

```
# Should haved added re.IGNORECASE so BPE merges can happen for capitalized versions of contractions
self.pat = re.compile(r"""'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+""")
```

  - These are chunks instead of words because 1) there are usually numbers in the text, 2) if a word has a leading white space, that white space is grouped with the word (e.g. “hello world” -> “hello”, “ world”), 3) punctuations are usually separated from word, and 4) “I’ve” -> (“I”,“’ve”.
- They ran the BPE algo for 50000 iteration on WebText, which creates a lookup table of 50256 items.
  - They added one more special token as then 50257th item: <|endoftext|>, which separates two sentences.
- To check its lookup table: link.

GPT-2 tokenizer encoding

1. Use regex to break a string to chunks.
2. For each chunk, encode it to UTF-8 format.
3. Use the lookup table to compress the number of tokens.
4. At the end of each sentence, we add the special token <|endoftext|>.
5. Finally, we put all of them together to form a tensor of tokens.
6. Note that these all happen before the pretraining.
    a. This is what "megatron-lm/tools/preprocess_data.py" does.

GPT-2 tokenizer decoding

- Decoding is needed when we want to generate a sentence.
- It is the reverse of encoding.
- Given a list of tokens (i.e. integers), we need to convert them into a string.
    - This is also an iterative process.
    - We need to decompose one token into a pair of tokens until we every token cannot be decomposed anymore.
    - Finally, we use the inverse of the lookup table to convert each token to a char.

Layers in GPT-2 model

How to construct the pretrianing as an unsupervised learning problem?

Consider pre-training as an optimization problem

- GPT-2 was trained using unsupervised learning method.
  - No manually crafted labels needed.
- Given an unsupervised corpus of tokens U = {u_1, …, u_n}, we use a standard language modeling objective to maximize the following likelihood:

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta)$$

  Where *k* is the context window, and *P* is modeled using a neural network w/ parameters \theta.

Another way to understand the loss func.

- To get the probability of generating a target sentence, we multiply the conditional probability of generating each target “word”:

$$P(U) = \prod_i P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta)$$

- One common trick for simplifying optimization problems is to take the log of the a product and make it a summation problem (min still be the same):

$$\begin{aligned} \log(P(U)) &= \log \left(\prod_i P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta) \right) \\ &= \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta) \end{aligned}$$

- log(p) can be calculated from cross entropy loss (torch.nn.CrossEntropyLoss).

How to generate prob.

- OpenAI team used a multi-layer Transformer decoder.
- This model applies a multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens

$$h_0 = UW_e + W_p$$
$$h_l = \texttt{transformer\_block}(h_{l-1}) \forall i \in [1, n]$$
$$P(u) = \texttt{softmax}(h_n W_e^T)$$

  where W_e is the embedding layer, W_p is the positional encoding layer, and there are *n* decoders in this model.

- Note that we need W_e both for the input and output.

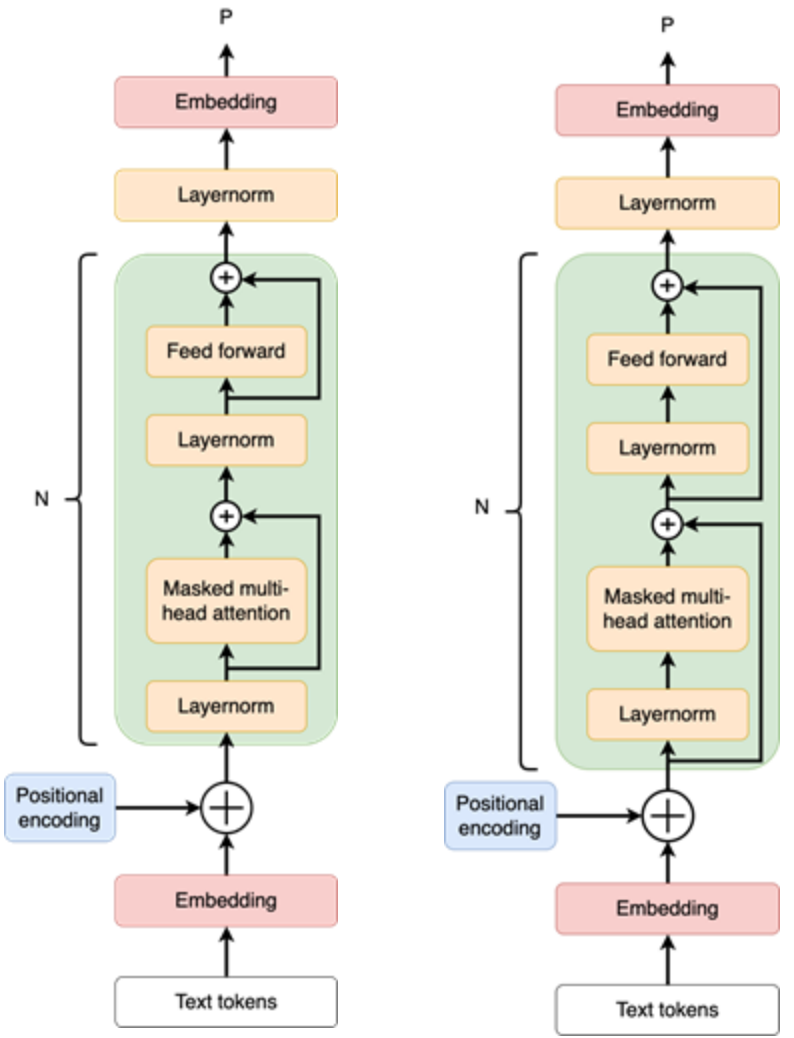
A more concrete example

- This is an oversimplified example.
- Hopefully, it can convey the basic ideas of input and output shapes.
- Assume we have a training corpus as:
  - “Generative Pre-trained Transformer 2 (GPT-2) is a large language model by OpenAI and the second in their foundational series of GPT models.”
- We first need to break a corpus into blocks/chunks:
  - Suppose we have block size (or context size *k*) of 8.
  - [“Generative Pre-trained Transformer 2 (GPT-2) is a large language”, “model by OpenAI and the second in their”, “foundational series of GPT models.”]
  - If we focus on the 1st chunk:
    - Input: [[“Generative”], [“Generative”, “ Pre-trained”], …, [“Generative”, “ Pre-trained”, “ Transformer”, “ 2”, “ (“, “GPT”, “-”, “2”, “)”, “ is”, “ a”, “ large”, “ language”]]
    - Output: [“Pre-trained”, “Transformer”, …, “model”].
    - Output is simply the next word of the input words in their original corpus.

GPT-2’s architecture

GPT-2's block diagram

- GPT-2's architecture is very similar to GPT-1.
- However, they made some corrections.
  - Layer norm is moved to the input of each sub-block.
  - An additional layer norm is added to the end of decoders.
- We will cover **embedding layer**, **positional encoding layer**, **multihead attention block**, and **layer norm** in more details in the following slides.
- Code can be found here.



Post-layernorm residual

Pre-layernorm residual

Code snippet

- A very simple implementation of the decoder layer (link)

```
class Block(nn.Module):
    def __init__(self, n_ctx, config, scale=False):
        super(Block, self).__init__()
        nx = config.n_embd
        self.ln_1 = LayerNorm(nx, eps=config.layer_norm_epsilon)
        self.attn = Attention(nx, n_ctx, config, scale)
        self.ln_2 = LayerNorm(nx, eps=config.layer_norm_epsilon)
        self.mlp = MLP(4 * nx, config)

    def forward(self, x, layer_past=None):
        a, present = self.attn(self.ln_1(x), layer_past=layer_past)
        x = x + a
        m = self.mlp(self.ln_2(x))
        x = x + m
        return x, present
```

Embedding layer

After tokenization…

- After converting strings to tokens, can we directly use these tokens for the attention mechanism?
- No! We need to represent each token by a vector.
- Why bother? If two “words” have similar semantic meaning (e.g. “good” and “nice”), we also want them to have similar mathematical representations (maybe in a higher dimensional space).
- To know more about it, check the distributed representation of words paper.

Example of vector representations of words

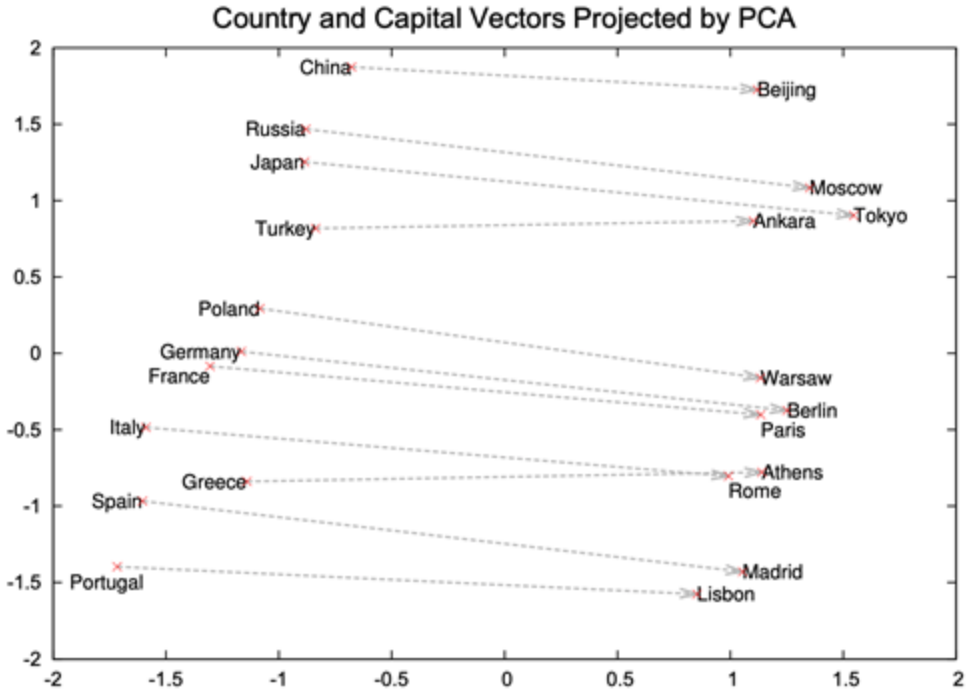
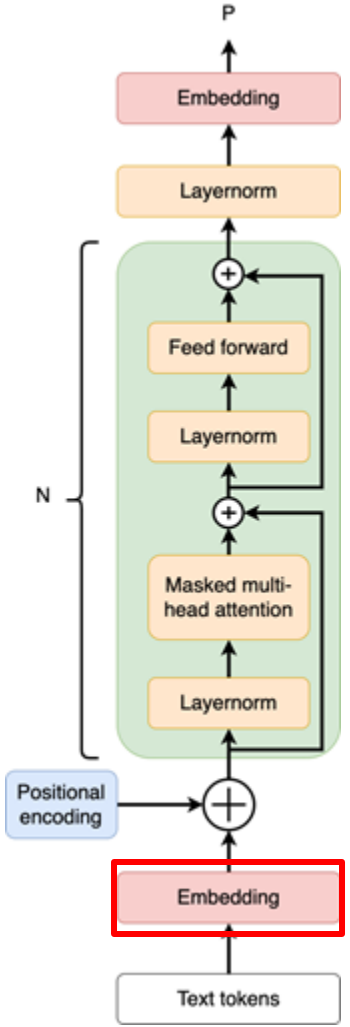


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

Embedding layer

- How to convert tokens to vectors?
- This is where embedding layer comes into play.
- Embedding layer is a lookup table, which converts tokens into their vector rep.
- Tokens are indices here.
- Embedding layer is also trainable.
- Check [torch.nn.Embedding] to see the implementation details.



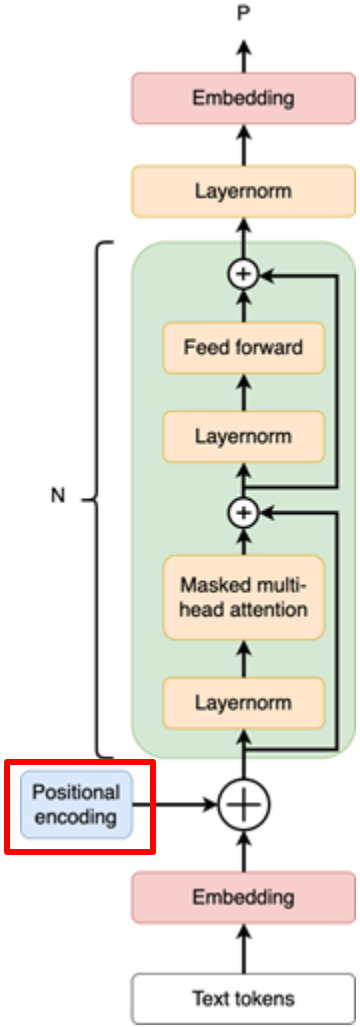
Padding

- From the example in page 18, we see that the number tokens in each sequence is not the same.
- This is a problem for matrix operations that we will see later in the self-attention section.
- Therefore, padding is needed.
- Padding to the right by token=0 is enough, since we only want each sentence to have the same number of tokens.
- However, this brings up other problems when we talk about self-attention and normalization.

Positional encoding layer

Positional encoding layer

- After the embedding layer, we need a positional encoding layer.
- Why? Because we want to let model have the positional info of “words” in the string, so that we can perform operations on them in parallel.
- There are many ways to construct the positional encoding layer, for example, in the transformer paper, they use cos/sin function.
- In GPT-2 paper, they use [torch.nn.Embedding] as the positional encoder, making it trainable.



Layer normalization

Why normalization?

- It provides better model performance in terms of accuracy.
  - The gradients weights in one layer are highly dependent on the outputs of the neurons in the previous layer especially if these outputs change in a highly correlated way.
  - Batchnorm alleviate this problem in the backprop process.
- The most commonly used norm is batch normalization.
  - We normalize across all the training examples in the current batch.

$$\mathrm{BatchNorm}(u) = \gamma \frac{u - \mu}{\sigma + \epsilon} + \beta$$

$$\mu = \frac{1}{B}\sum_{b=1}^{B} u_b,\ \sigma = \sqrt{\frac{1}{B}\sum_{b=1}^{B}(u_b - \mu)^2}$$

where μ and σ keep the same dim as the number of features in a batch, and γ and β are trainable parameters.

Layer normalization

- Suppose every example has d-dim

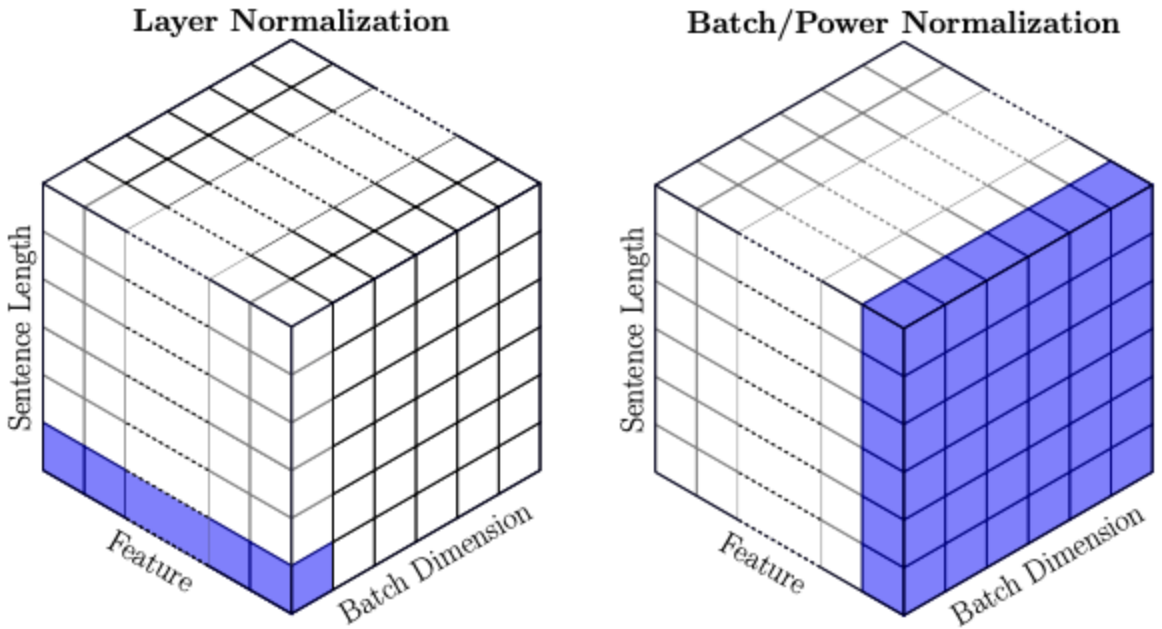
$$\text{LayerNorm}(u) = \gamma \frac{u - \mu}{\sigma + \epsilon} + \beta$$

$$\mu = \frac{1}{d}\sum_{i=1}^{d} a^i,\ \sigma = \frac{1}{d}\sqrt{\sum_{i=1}^{d}(a^i - \mu))}$$

  where μ and σ are 1-dim values.

- For sentences, we perform layernorm on all the features in one “word”, which is usually the last dim in K.shape, Q.shape, and V.shape.
- Torch document: link, and the paper.

Visualization



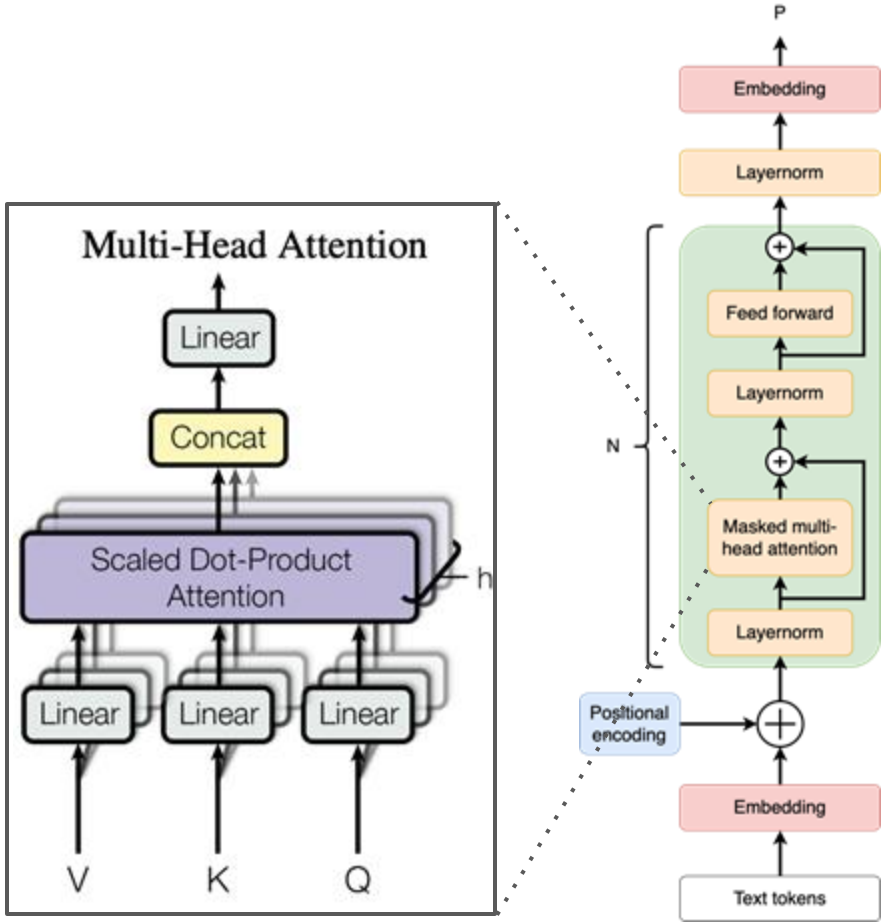
Why layer norm instead of batch norm?

- **Sentences differ in length.**
- A minibatch has sentences with different length **->** μ and σ in batch norm changes frequently across different mini-batches.
- μ and σ learned in training might not be suitable for unseenly long sentences in inference.

Multi-head self-attention mechanism

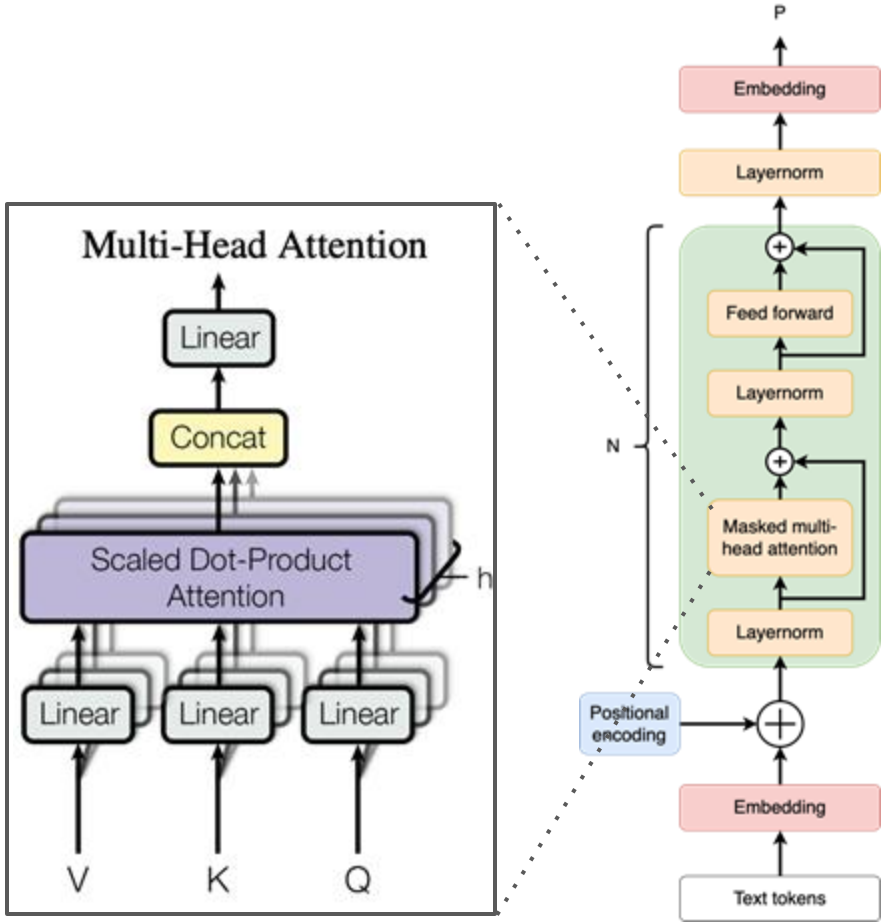
Multi-head Attention - cont.

- Key (K), Value (V), and Query (Q) are the same.
- But, after passing them through the Linear layers (feed forward layers), they will differ.
- K, V, and Q are splitted along the d_model dim, where d_model is the dim of vectors stored in embedding layers and the output dim of each decoder block.



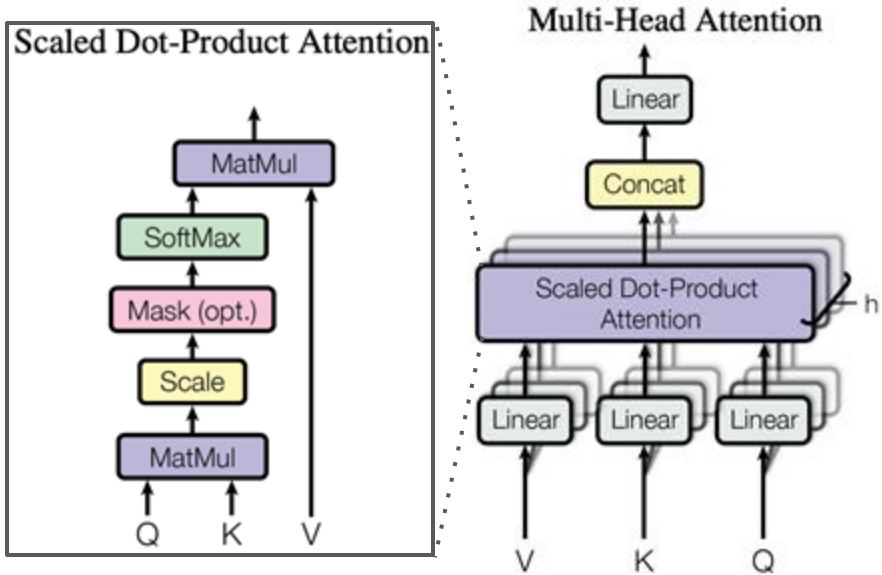
Multi-head Attention - cont.

- After the embedding layers, the input has three dim: (batch, sequence_len, d_model).
- After the split, each of them has shape: (batch, sequence_len, num_head, d_model//num_head).
- All these dims are hyper-params.
- This multi-head split can help reduce the computation in each attention block significantly.



Masked self-attention

- torch.matmul(Q,K) measures similarities between vect. in Q and K.
- Scaling by sqrt(d_model//num_head) to avoid having all prob concentrate on one entry in the matmul result.
- Mask is a lower-triangular matrix, which masks the query from future key content.
- Softmax gives the prob.
- This whole process can be seen as computing the weighted average sum of vectors in V.



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

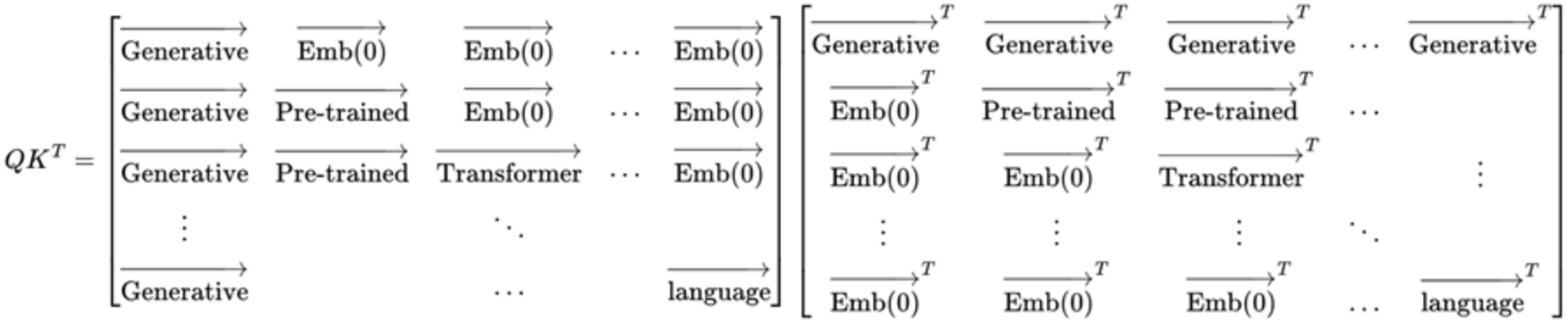
A more concrete example

- Let's reuse the example in page 18 and assume batchsize = 1.
  - Input: [["Generative"], ["Generative", " Pre-trained"], …, ["Generative", " Pre-trained", " Transformer", " 2", " (", "GPT", "-", "2", ")", " is", " a", " large", " language"]]
- Let's ignore the linear transformation and consider Q=K=V.
- Let's also ignore multihead attention for simplicity.
- Let's visualize Q, K, and V in matrix form:

$$Q = K = V = \begin{bmatrix} \overrightarrow{\text{Generative}} & \overrightarrow{\text{Emb}(0)} & \overrightarrow{\text{Emb}(0)} & \cdots & \overrightarrow{\text{Emb}(0)} \\ \overrightarrow{\text{Generative}} & \overrightarrow{\text{Pre-trained}} & \overrightarrow{\text{Emb}(0)} & \cdots & \overrightarrow{\text{Emb}(0)} \\ \overrightarrow{\text{Generative}} & \overrightarrow{\text{Pre-trained}} & \overrightarrow{\text{Transformer}} & \cdots & \overrightarrow{\text{Emb}(0)} \\ \vdots & & \ddots & & \\ \overrightarrow{\text{Generative}} & & \cdots & & \overrightarrow{\text{language}} \end{bmatrix}$$

Matmul of Query and Key

- For torch.matmul(Q, K^T):



where \vec{Emb(0)} represents the embedding results of token=0.

Need an attention mask

- Without an attention mask, we will have an attention score from the “future”:

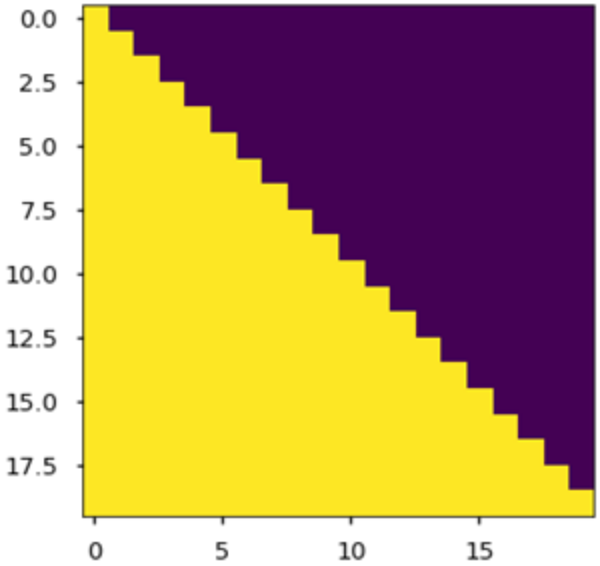
$$Q[0,:]K^T[:,1] = \left[\overrightarrow{\text{Generative}}\ \overrightarrow{\text{Emb}(0)}\ \overrightarrow{\text{Emb}(0)}\ \cdots\ \overrightarrow{\text{Emb}(0)}\right] \begin{bmatrix} \overrightarrow{\text{Generative}}^T \\ \overrightarrow{\text{Pre-trained}}^T \\ \overrightarrow{\text{Emb}(0)}^T \\ \vdots \\ \overrightarrow{\text{Emb}(0)}^T \end{bmatrix}$$

- We want the model to guess the next “word” given only the existing context.
- In other words, vectors in key and query should not see the following texts. Otherwise, it is cheating!
- Therefore, we need an attention mask, which should filter out all the attention scores computed using future information.

Attention mask

- An attention mask can be as simple as a lower triangular matrix, where the lower entries are 1s and the upper entries are 0s.
- Code snippet (link)

```
max_positions = config.max_position_embeddings
self.register_buffer(
    "bias",
    torch.tril(torch.ones((max_positions, max_positions), dtype=torch.bool)).view(
        1, 1, max_positions, max_positions
    ),
    persistent=False,
)
```



Yellow=1, purple=0

Some final reminders on MLP block

- The output from multi-head blocks are concatenated together.
- The activation they used in GPT-2 is GeLU instead of ReLU.
  - GeLU is continuous and differentiable everywhere.

```
def gelu(x):
    return 0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))
```

- Two layers in MLP block:
  - Hidden size = 4 x d_model.
  - Output size = d_model.
- After the MLP layer, the output will have the same shape as the input, so that we can easily pass it to the next decoder layer without worrying about dims.

Fine-tuning

Training strategy:

1. Fine-tune a pretrained model (e.g. GPT-3).
2. Train reward model.
3. Perform reinforcement learning on SFT.

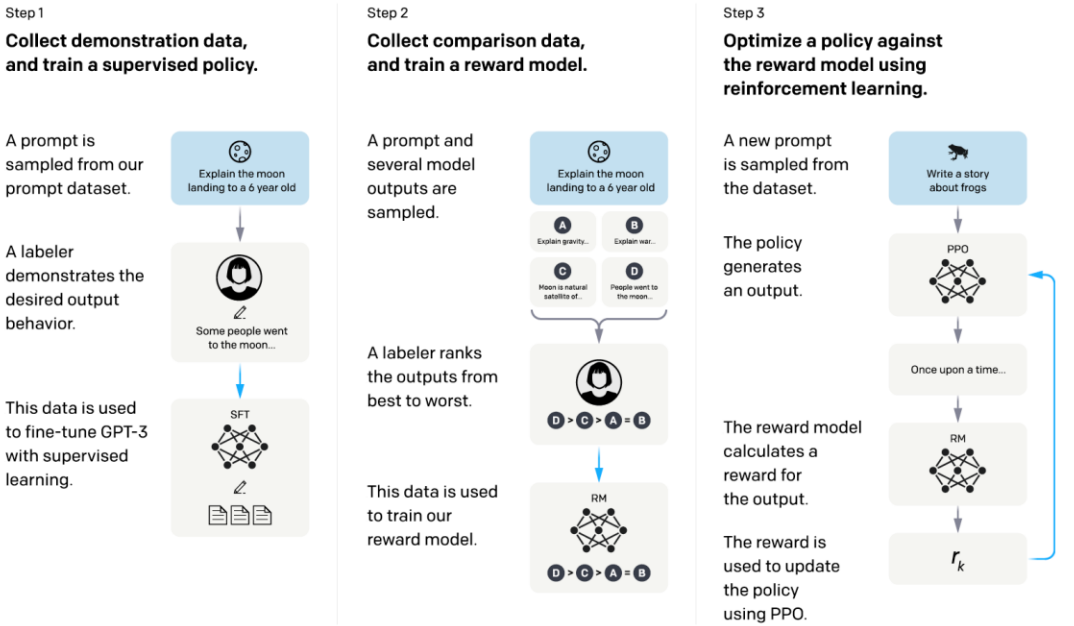
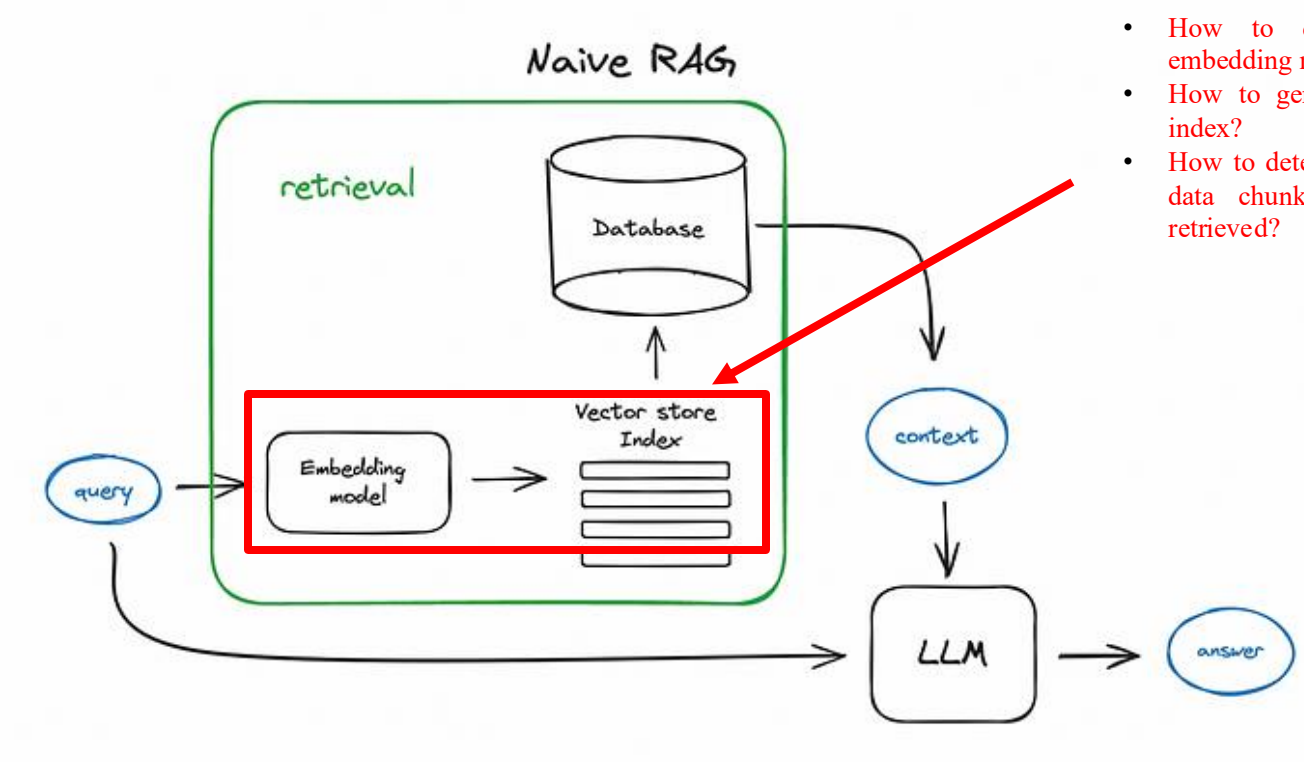


Figure 2: A diagram illustrating the three steps of our method: (1) supervised fine-tuning (SFT), (2) reward model (RM) training, and (3) reinforcement learning via proximal policy optimization (PPO) on this reward model. Blue arrows indicate that this data is used to train one of our models. In Step 2, boxes A-D are samples from our models that get ranked by labelers. See Section 3 for more details on our method.

Retrieval Augmented Generation



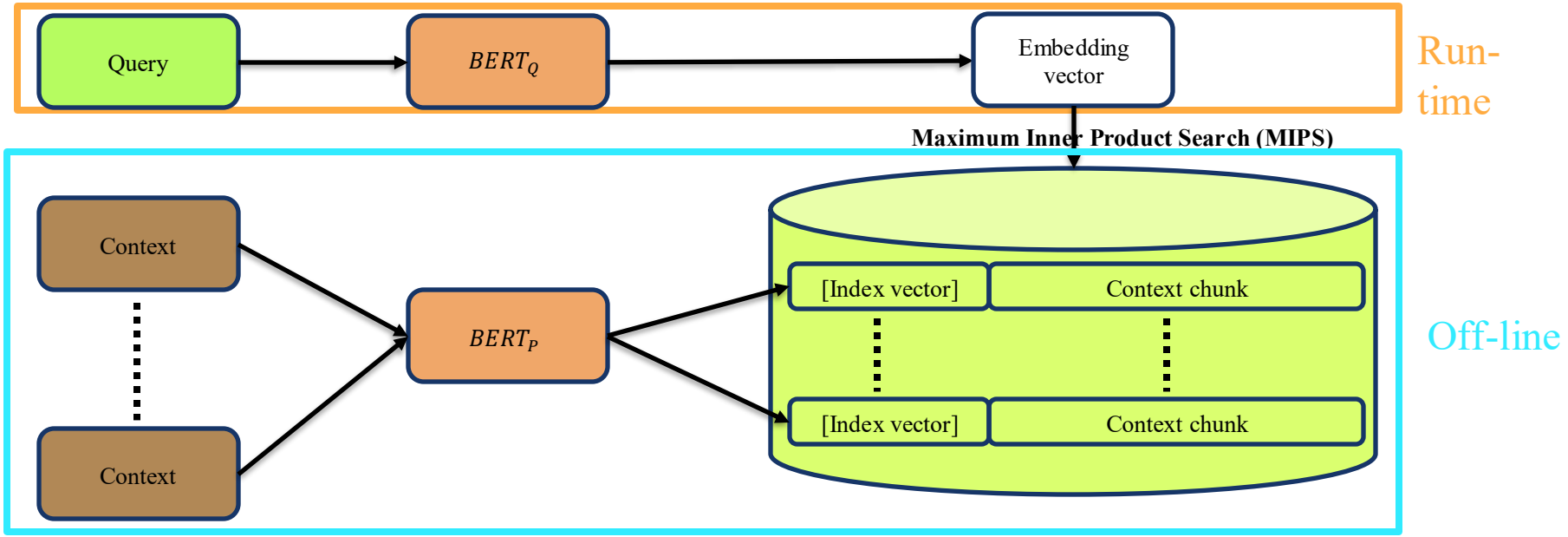
- How to design such embedding model?
- How to generate vector index?
- How to determine which data chunk should be retrieved?

Motivation

- Existing 2-stage retrieval has huge performance degradation in practice
  - A context retriever first selects a small subset of passages that may contain the answer.
    - E.g. TF-IDF or BM25 matches the keywords.
    - Keywords matching often fails when two sentences only contain synonyms.
  - Then a machine reader examines and identifies which of them has the answer.
- This is caused by sparse vector representation.
  - Such as one-hot encoding.
- However, this can be fixed by dense vector representation.
  - The vector rep. of two synonyms can have high similarity.
  - These dense rep. are usually learnable -> flexible and task-specific.
  - But this learning process is difficult at that time.
- So, how to train a better dense embedding model using only pairs of Q&A without additional pre-training?

Solution - DPR

- Dense Passage Retriever (DPR)
  - Also known as two-tower model.
  - Can provide the top-k most relevant passages in the database at run-time.
  - Use BERT as the encoding/embedding model.



Solution - DPR

- Dense encoder $E_P(\cdot)$ maps text/context to a d-dim dense continuous vector.
- Another encoder $E_Q(\cdot)$ maps the query to a d-dim vector.
  - $d = 768$ and both BERT are base, uncased.
- The similarity is measured by the inner product of $E_P(p)$ and $E_Q(q)$
  - $\text{sim}\,(p, q) = E_P(p)^T E_Q(q)$
- By traversing and computing the inner product between all the indexes in a database, we can find top-k context that are most relevant.
  - Have sub-linear implementation.
- Other similarity measurement method can be used
  - E.g. cosine similarity (same as dot prod if unit vectors) and Euclidean distance.
- This method is now widely used in recommendation systems.

Training

- Given a training dataset $\mathcal{D} = \{< q_i, p_i^+, p_{i,1}^-, \dots, p_{i,n}^- >\}_{i=1}^m$.
  - q_i: i-th query, p_i^+: the relevant context, $p_{i,1}^-, \dots, p_{i,n}^-$: n irrelevant context.
- Target is to minimize the negative log likelihood.
  - $L\left(q_i, p_i^+, p_{i,1}^-, \dots, p_{i,n}^-\right) = -\log \frac{e^{sim\left(q_i, p_i^+\right)}}{e^{\text{sim}\left(q_i, p_i^+\right)} + \sum_{j=1}^n e^{sim\left(q_i, p_j^-\right)}}$.
- Negative sample selection
  - Every query has only one positive example but can have a large pool of negative examples.
  - Need to perform sampling: 1) random, 2) use result returned by BM25 (not contain answer), 3) other positive samples paired with other questions:
    - Positive and negative pairs
    - In-batch negatives
      - Assume B examples in a mini-batch and each has a positive passage.
      - Let $\mathbf{Q}$ and $\mathbf{P}$ be $(B \times d)$ matrices of question and embeddings in a batch.
      - Then $\mathbf{S} = \mathbf{Q}\mathbf{P}^T$ is a $(B \times B)$ matrix of similarities scores, and only when i=j, (q_i, p_j) is a pos. example.
      - This creates $\boldsymbol{B}$ training example where each example has $\boldsymbol{B}$**-1** negative examples.
- In-batch negative with one negative from BM25 works the best.

Dataset

- Context/passage generation
  - Extract text from English Wikipedia and break it into chunks (i.e. passages) of 100 words.
  - Totally, 21,015,324 passages.
- Question answering datasets
  - Natural Questions (NQ)
  - TriviaQA
  - WebQuestions
  - CuratedTREC
  - SQuAD v1.1

Dataset	Train		Dev	Test
Natural Questions	79,168	58,880	8,757	3,610
TriviaQA	78,785	60,413	8,837	11,313
WebQuestions	3,417	2,474	361	2,032
CuratedTREC	1,353	1,125	133	694
SQuAD	78,713	70,096	8,886	10,570

Table 1: Number of questions in each QA dataset. The two columns of **Train** denote the original training examples in the dataset and the actual questions used for training DPR after filtering. See text for more details.

Training setup

- In-batch negative with one negative from BM25.
- Batch size of 128.
- Question and passage encoders training epochs
  - 40 epochs for large datasets (NQ, TriviaQA, SQuAD).
  - 100 epochs from small datasets (TREC, WQ).
- Learning rate: 10^{-5}.
- Optimizer: Adam.
- Linear learning rate scheduler with warm-up.
- Dropout rate: 0.1.

Results

Training	Retriever	Top-20					Top-100				
		NQ	TriviaQA	WQ	TREC	SQuAD	NQ	TriviaQA	WQ	TREC	SQuAD
None	BM25	59.1	66.9	55.0	70.9	68.8	73.7	76.7	71.1	84.1	80.0
Single	DPR	78.4	79.4	73.2	79.8	63.2	85.4	**85.0**	81.4	89.1	77.2
	BM25 + DPR	76.6	79.8	71.0	85.2	**71.5**	83.8	84.5	80.5	92.7	**81.3**
Multi	DPR	**79.4**	78.8	**75.0**	**89.1**	51.6	**86.0**	84.7	**82.9**	93.9	67.6
	BM25 + DPR	78.0	**79.9**	74.7	88.5	66.2	83.9	84.4	82.3	**94.1**	78.6

Table 2: Top-20 & Top-100 retrieval accuracy on test sets, measured as the percentage of top 20/100 retrieved passages that contain the answer. *Single* and *Multi* denote that our Dense Passage Retriever (DPR) was trained using individial or combined training datasets (all the datasets excluding SQuAD). See text for more details.

- Single: train model on each dataset separately
- Multi: train model on all dataset combined (excluding SQuAD).
- BM25+DPR: a linear combination
  - First, get 2 set of top-200 passages based on BM25 and DPR separately.
  - Then rerank the union w/ a new ranking function: $\text{BM25}(q,p) + \lambda \cdot \text{sim}(q,p)$, where $\lambda = 1.1$.

Results

- Sample efficiency
  - DPR > BM25 when use 1,000 examples for train.
  - More examples used for training leads to higher top-k retrieval accuracy.

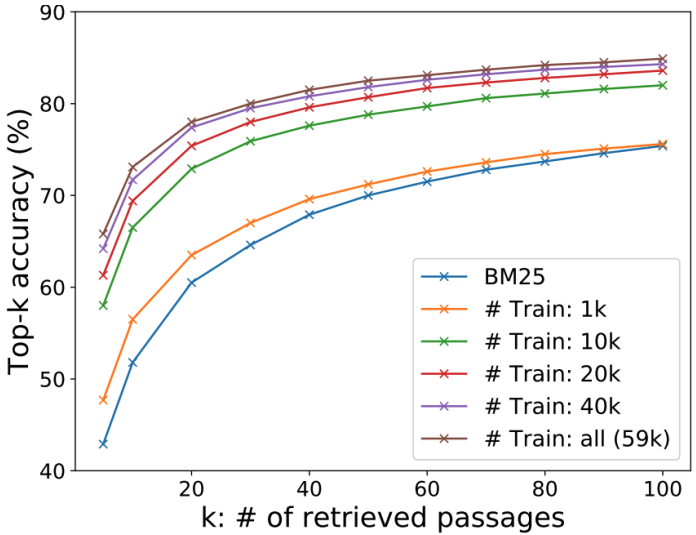


Figure 1: Retriever top-k accuracy with different numbers of training examples used in our dense passage retriever vs BM25. The results are measured on the development set of Natural Questions. Our DPR trained using 1,000 examples already outperforms BM25.

Results

- In-batch negative training
  - Easy and memory friendly.
  - Gold: negative questions are other positives from the same training batch.
  - When $k \geq 20$, random, BM25, and gold have similar performance.
  - W/ IB used, gold (#N=7) improves substantially.
  - Accuracy continues to improve as #N increases.
  - Adding one negative example from BM25 greatly improves accuracy.
    - BM25's negative example is a "hard" example because BM25 gives it high score, but it actually does not contain the answer.

Type	#N	IB	Top-5	Top-20	Top-100
Random	7	✗	47.0	64.3	77.8
BM25	7	✗	50.0	63.3	74.8
Gold	7	✗	42.6	63.1	78.3
Gold	7	✓	51.1	69.1	80.8
Gold	31	✓	52.1	70.8	82.1
Gold	127	✓	55.8	73.0	83.1
G.+BM25$^{(1)}$	31+32	✓	65.0	77.3	84.4
G.+BM25$^{(2)}$	31+64	✓	64.5	76.4	84.0
G.+BM25$^{(1)}$	127+128	✓	**65.8**	**78.0**	**84.9**

Table 3: Comparison of different training schemes, measured as top-k retrieval accuracy on Natural Questions (development set). #N: number of negative examples, IB: in-batch training. G.+BM25$^{(1)}$ and G.+BM25$^{(2)}$ denote in-batch training with 1 or 2 additional BM25 negatives, which serve as negative passages for all questions in the batch.

Results

- Similarity measurement
  - L2 and dot product gives the same performance, and both are superior to cosine.
- Loss function
  - Triplet hinge loss has comparable performance to negative log likelihood.
  - $L(q_i, p_i^+, p_i^-) = \max\{0, \text{sim}(q_i, p_i^-) - \text{sim}(q_i, p_i^+) + m\}$ (m: hyper-parameter)
- Cross-dataset generalization
  - DPR generalizes well when trained on one dataset and then apply it to a different dataset.
  - But there are 3-5 points performance loss, compared to the best performing model fine-tuned on that dataset.

Results

- End-to-end QA System

Training	Model	NQ	TriviaQA	WQ	TREC	SQuAD
Single	BM25+BERT (Lee et al., 2019)	26.5	47.1	17.7	21.3	33.2
Single	ORQA (Lee et al., 2019)	33.3	45.0	36.4	30.1	20.2
Single	HardEM (Min et al., 2019a)	28.1	50.9	-	-	-
Single	GraphRetriever (Min et al., 2019b)	34.5	56.0	36.4	-	-
Single	PathRetriever (Asai et al., 2020)	32.6	-	-	-	**56.5**
Single	$\text{REALM}_{\text{Wiki}}$ (Guu et al., 2020)	39.2	-	40.2	46.8	-
Single	$\text{REALM}_{\text{News}}$ (Guu et al., 2020)	40.4	-	40.7	42.9	-
Single	BM25	32.6	52.4	29.9	24.9	38.1
	DPR	**41.5**	56.8	34.6	25.9	29.8
	BM25+DPR	39.0	57.0	35.2	28.0	36.7
Multi	DPR	**41.5**	56.8	**42.4**	49.4	24.1
	BM25+DPR	38.8	**57.9**	41.1	**50.6**	35.8

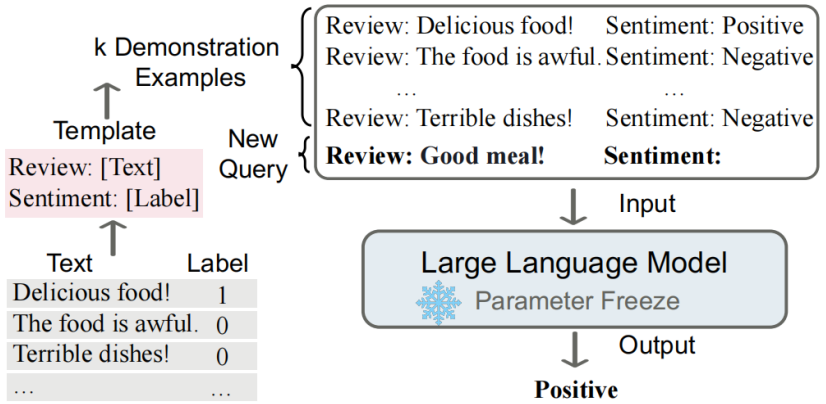
Table 4: End-to-end QA (Exact Match) Accuracy. The first block of results are copied from their cited papers. $\text{REALM}_{\text{Wiki}}$ and $\text{REALM}_{\text{News}}$ are the same model but pretrained on Wikipedia and CC-News, respectively. *Single* and *Multi* denote that our Dense Passage Retriever (DPR) is trained using individual or combined training datasets (all except SQuAD). For WQ and TREC in the *Multi* setting, we fine-tune the reader trained on NQ.

In-context Learning

In-Context Learning (ICL)

Definition (Dong, Q., et al., 2022): In-context learning is a paradigm that allows language models to learn tasks given only a few examples in the form of demonstration.

It is a task adaptation strategy that does not update the weights of the pre-trained model.



Reference: Dong, Q., et al. (2022). "A survey on in-context learning." arXiv preprint arXiv:2301.00234.

Fine-tuning VS. RAG

Use Case	Dynamic Vs Static data	External Knowledge	Model Customisation	Reducing Hallucinations	Transparency	Recommendation
Summarization (Specialized Domain & Style)	NA	No	Fine-tuning for adapting style	Less critical due to context	Context offers transparency	Fine-Tuning
Q/A System on Organizational Knowledge	RAG supports frequent updates	Yes	Depending on requirements	Critical due to lack of domain knowledge	RAG offers transparency	RAG (with possible fine-tuning)
Customer Support Chatbots	RAG supports frequent updates	Yes	Fine-tuning for adapting tone and politeness	Critical due to lack of domain knowledge	RAG offers transparency	Fine-Tuning + RAG
Code Generation System	Dynamic codebases benefit RAG	RAG for external codebases	Fine-tuning for code style	Critical for code correctness	RAG offers transparency	Fine-Tuning + RAG

Hands-on with OpenFold and OPT-125M Fine-tuning

Introduction

Approaches for Protein Structure Prediction:

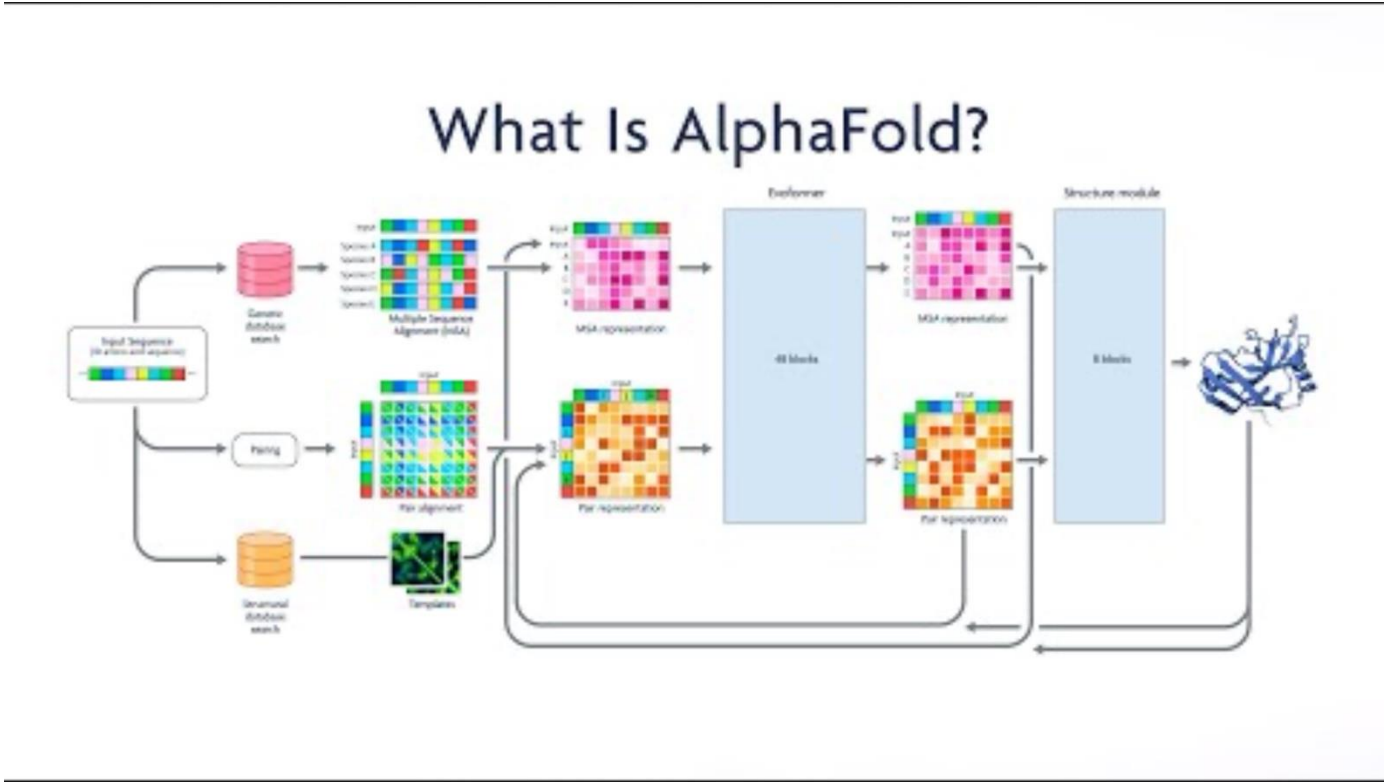
- Experiment - high cost of time and finance
- Computation - high throughput at a low cost
  - AlphaFold (CNN-based model) - demonstrate that DNN can be a efficient solution for protein structure prediction.
  - AlphaFold 2 (Transformer-based model) - the first model to achieve atomic accuracy

Challenge:

1. Limited global batch size for accuracy guarantee. (11 days to train on 128 Google TPUv3)
2. Huge memory consumption exceeds what current GPUs can handle.

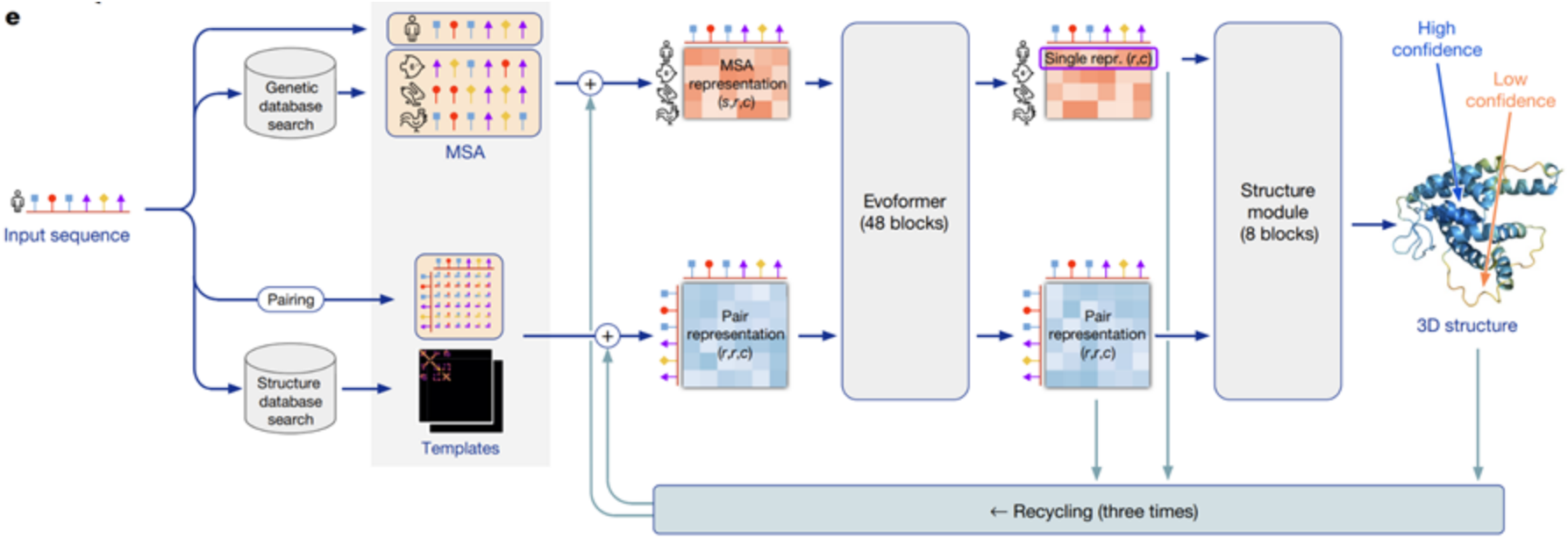
We refer AlphaFold as the transformer-based AlphaFold 2 model in the following slides.

AlphaFold


What Is AlphaFold?
Evoformer
Structure module
Input Sequence
Genetic database search
Multiple Sequence Alignment (MSA)
MSA representation
Pairing
Pair alignment
Pair representation
Structural database search
Templates
48 blocks
8 blocks

Background

- AlphaFold



Background

- Evoformer

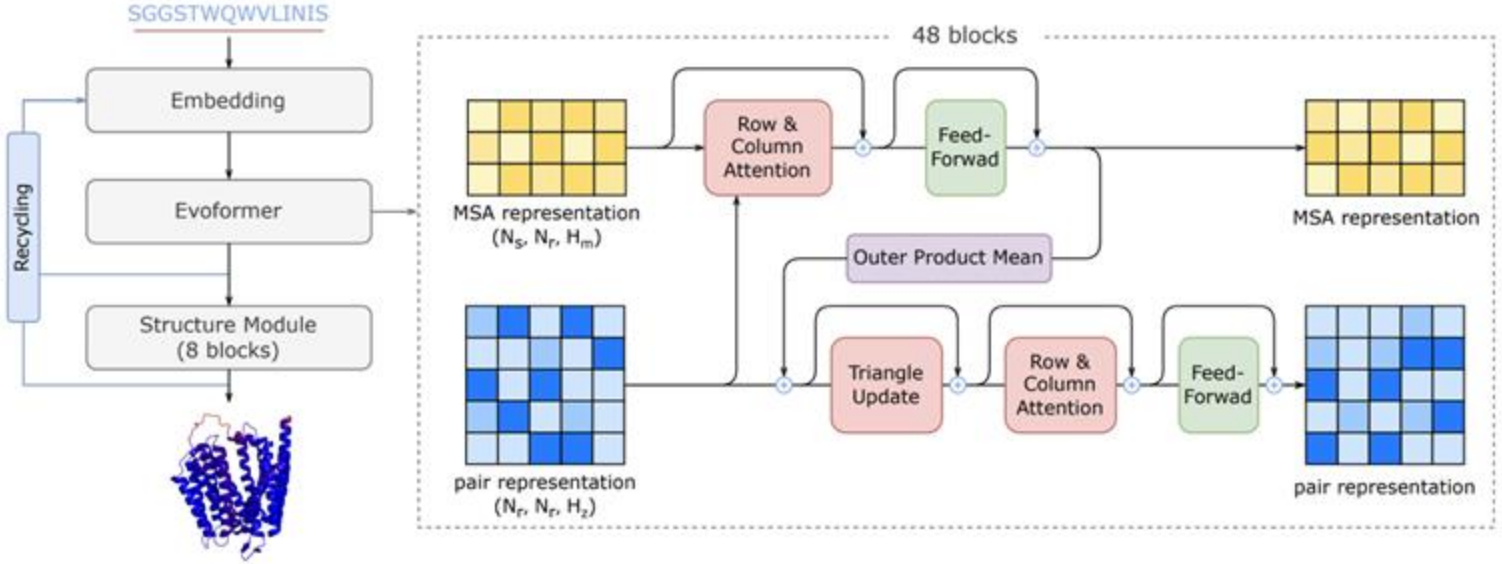


Fig. 1. The Architecture of AlphaFold Model. The amino acid sequence is encoded into MSA and pair representation after Embedding layer, then feeding into Structure Module after 48 Evoformer blocks. In Evoformer, MSA and pair representation were processed by *MSA Stack* and *Pair Stack*, respectively. In addition to this, there is a communication mechanism that allows information to be exchanged between the two representation.

Background

- Training Details

TABLE I
DETAILS OF ALPHAFOLD MODEL TRAINING.

Model	Initial Training	Fine-tuning
Residues sequence N_r	256	384
Number of sequences N_s	128	512
Batch size	128	128
Precision	Bfloat16	Bfloat16
Training samples ($\times 10^6$)	$\approx$ 10	$\approx$ 1.5
Training time	$\approx$ 7 days	$\approx$ 4 days

Thank you!

References

- Megatron-LM: https://github.com/NVIDIA/Megatron-LM
- Huggingface GPT-2: https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling_gpt2.py
- PyTorch version of OpenAI's GPT-2: https://github.com/graykode/gpt-2-Pytorch/blob/master/GPT2/model.py
- Illustration of GPT-2: https://jalammar.github.io/illustrated-gpt2/