When AI meets life sciences
当人工智能遇上生命科学

2025 清华大学 AI创新实践课程

张强锋，清华大学

Dec 8, 2025

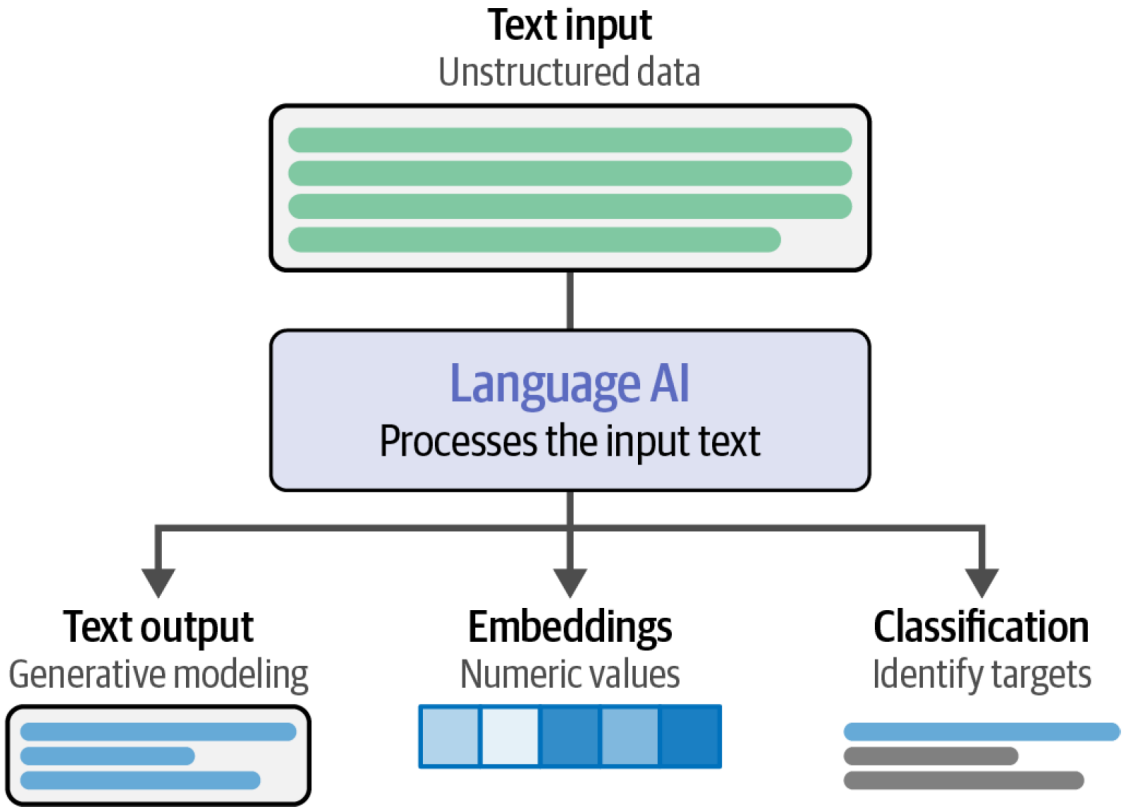
Outline

- The history of Natural Language Processing
- How to train or fine-tune a large language model
  - Data preparation and tokenization
  - Build a GPT model
  - Training and finetuning
- A GPT-style LLMs: evo
- BERT-style LLMs: from ESM-2 to ESM3
- Chat with biology foundation model

Sequence data (language) modeling

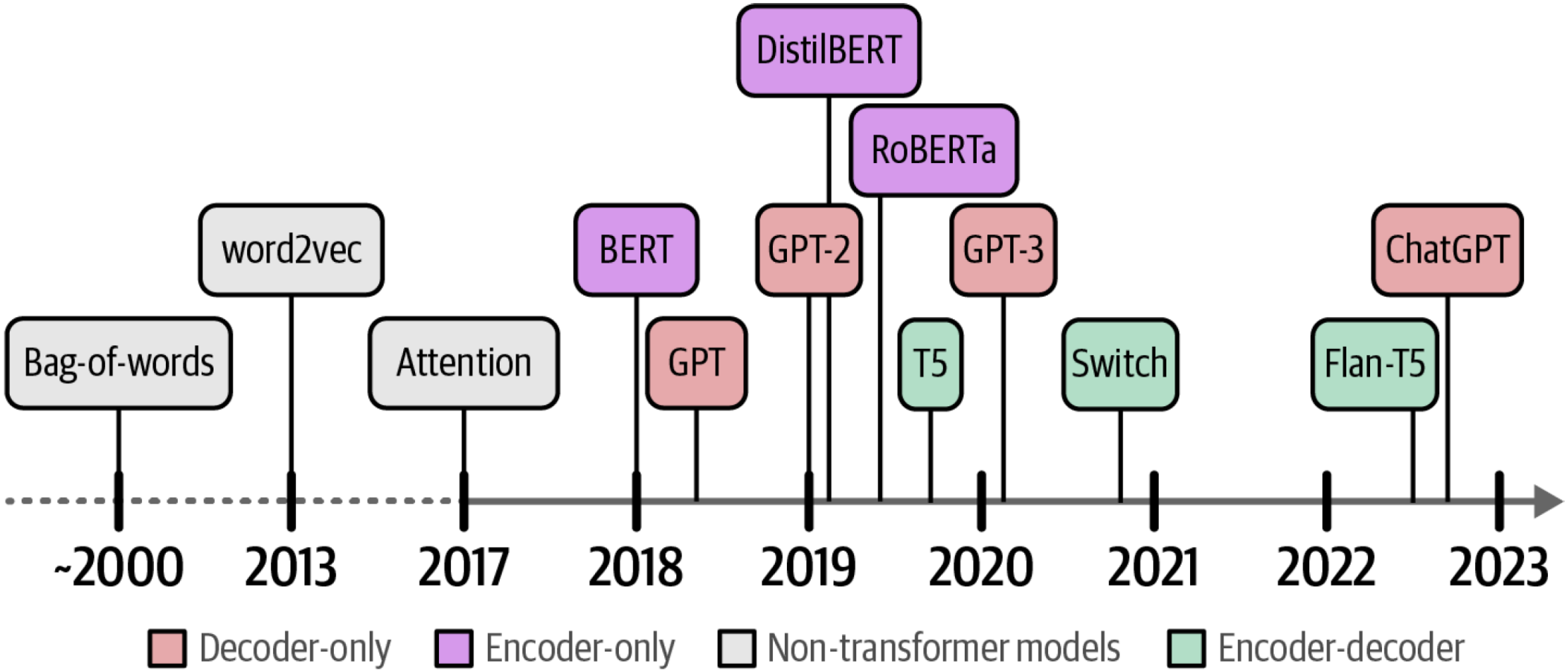


Language AI for sequence data modeling





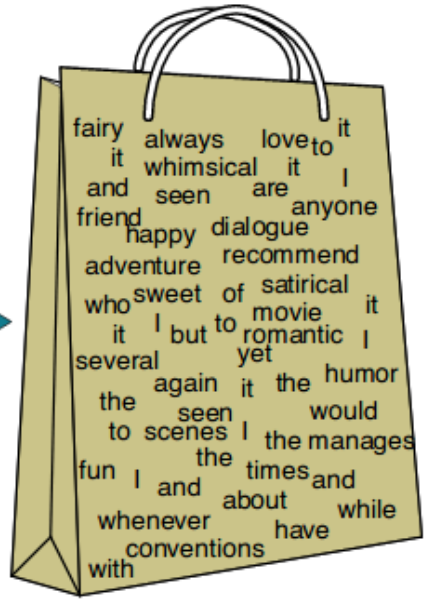
A recent history of language AI



Bag-of-words

- The **bag-of-words (BoW)** model is a model of text which uses an unordered collection (a "bag") of words.
- Limitation: disregards word order (and thus most of syntax or grammar)

I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!



it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
…	…

Yesterday, investors were rallying, but today, they are retreating.

Yesterday, investors were retreating, but today, they are rallying.

N-gram models

Text: "In consciousness studies the consciousness studies are fundamental"

Unigram	In	consciousness	studies	the	are	fundamental

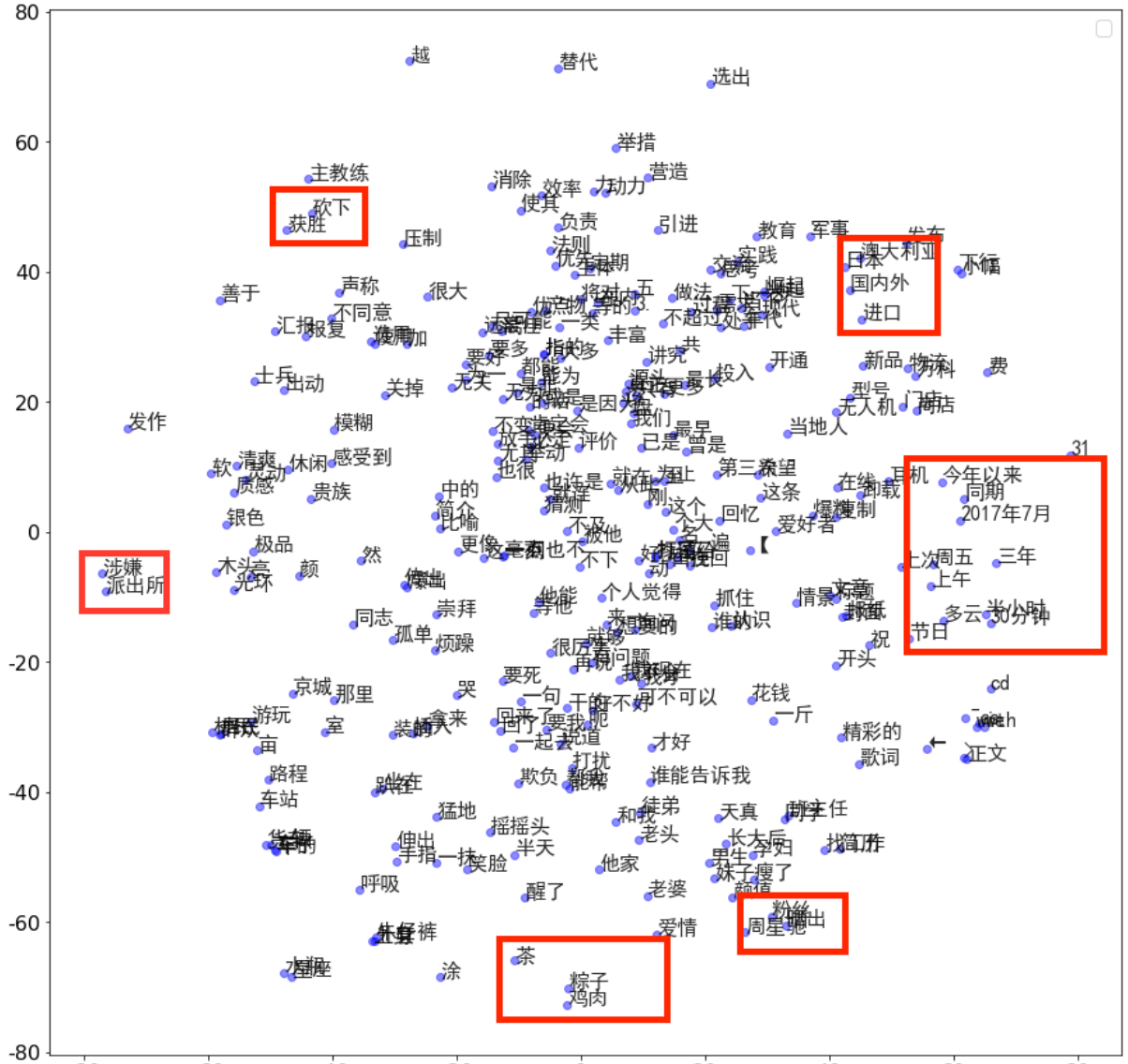
Bigram	In consciousness	consciousness studies	studies the	the consciousness	studies are	are fundamental

Trigram	In consciousness studies	consciousness studies the	studies the consciousness	the consciousness studies	consciousness studies are	studies are fundamental

© AIML.com Research

- For biological sequence: codon preference, sequence motif

Where is the meaning: word vector

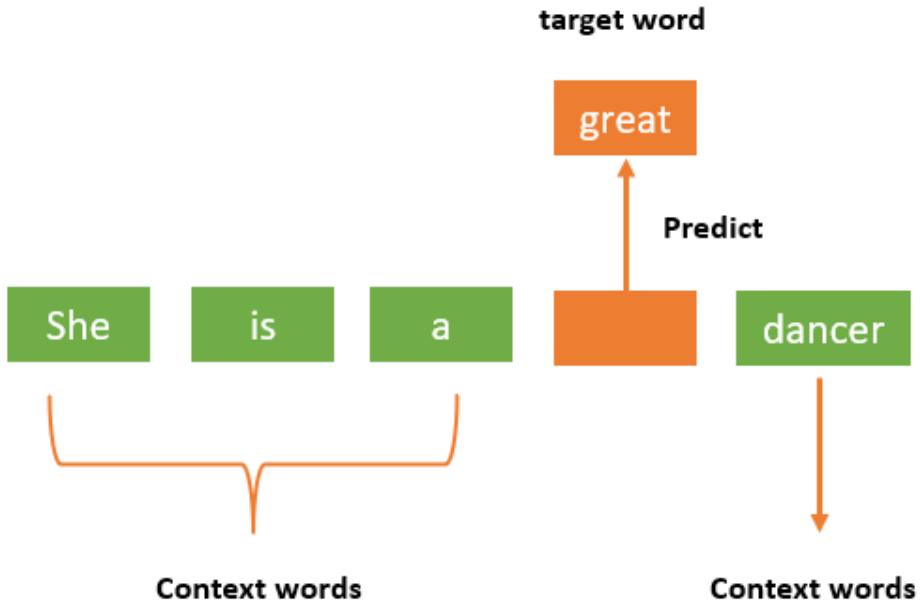


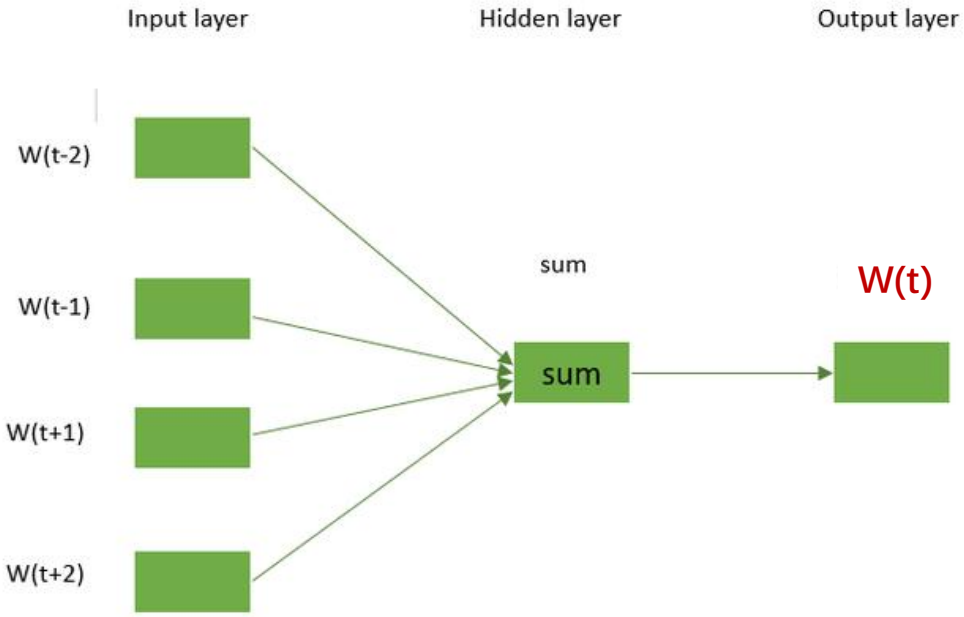
Where is similarity

Word embedding

- **one-hot embedding**
- **Word2Vec**
  - **CBOW**
  - **Skip-gram**
  - **FastText**
- **GloVe**

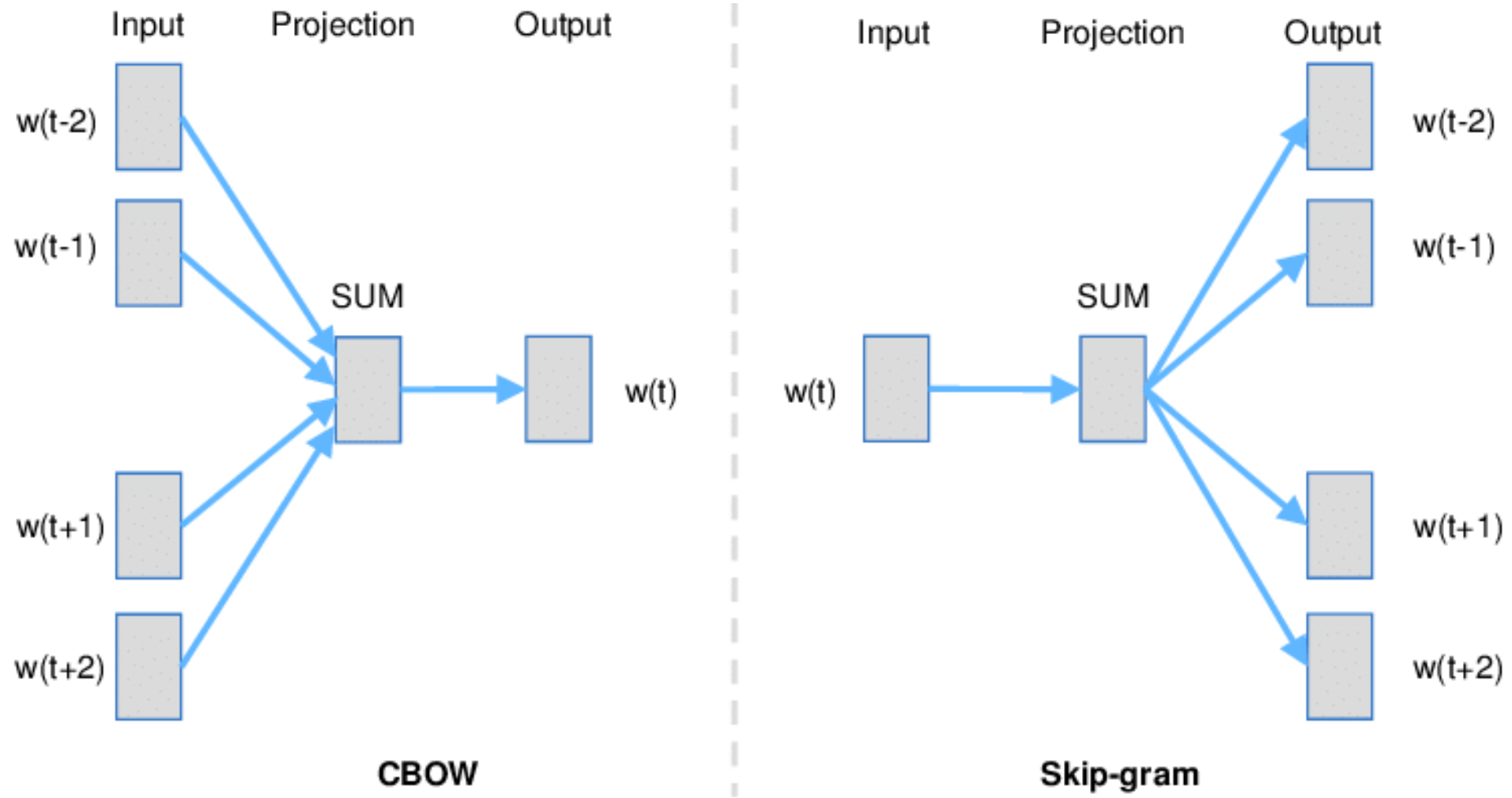
CBOW (Continuous Bag of Words)





The real goal is not to complete the prediction task, but to learn the useful by-product that emerges from the process—the word embedding matrix.

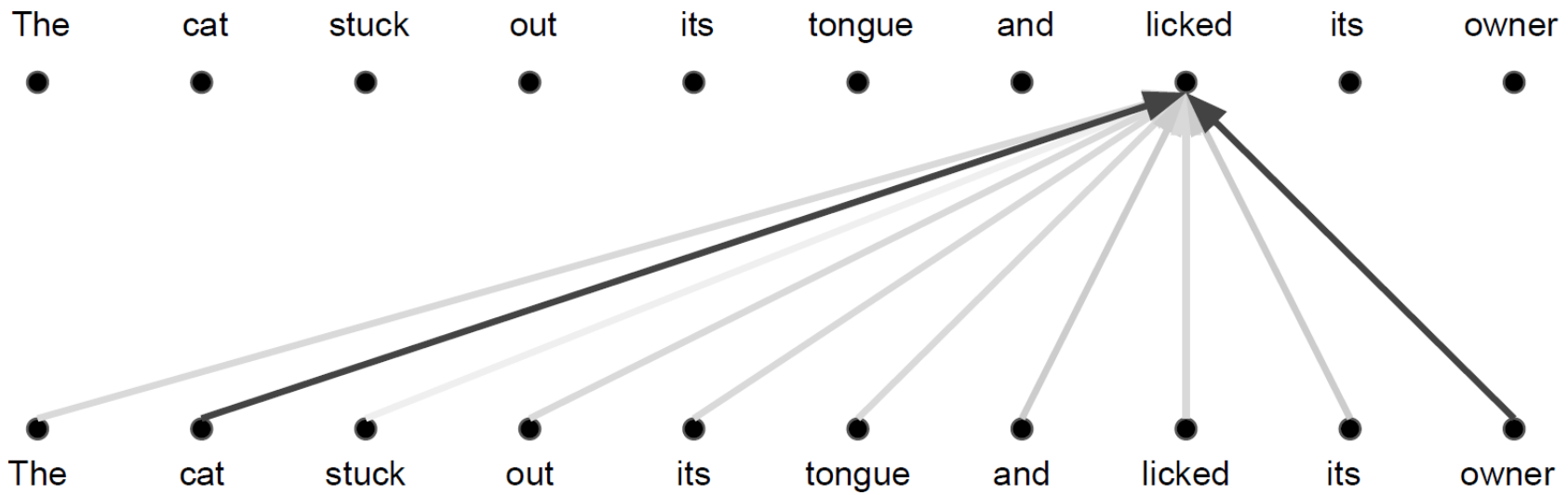
CBOW vs. Skip-gram



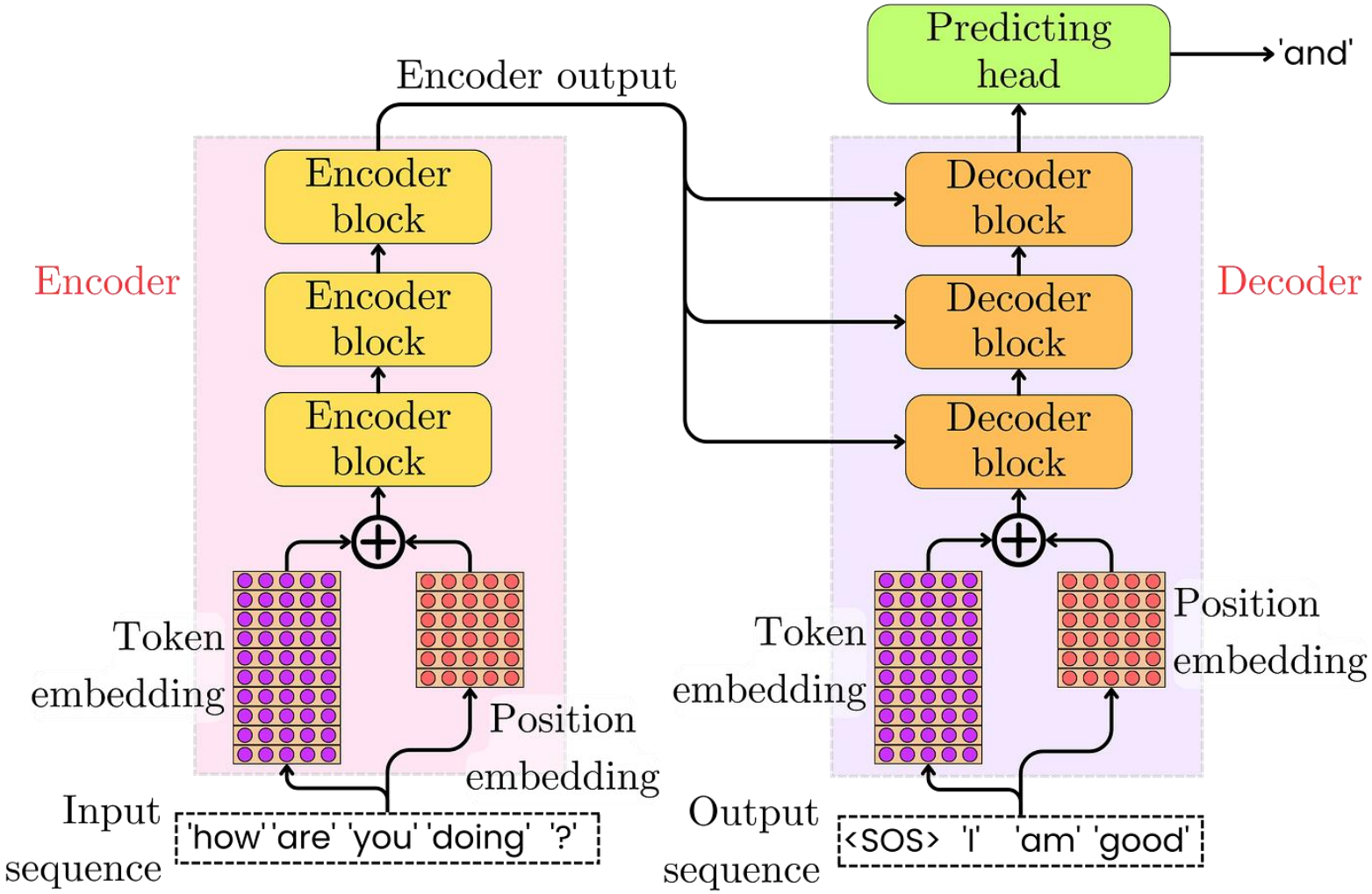
Attention is all you need



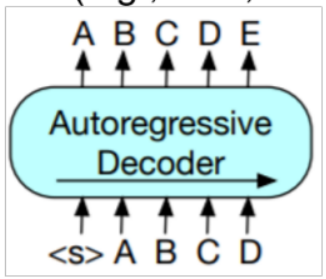
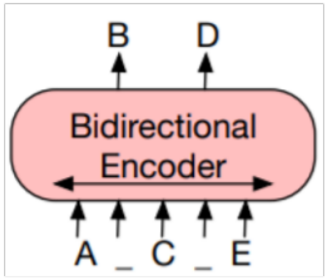
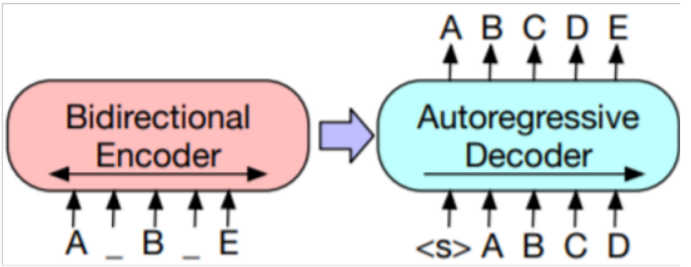
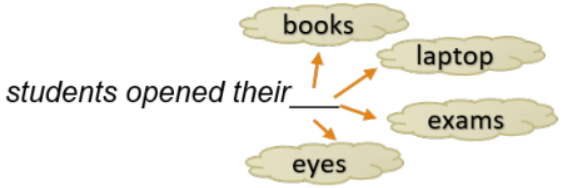
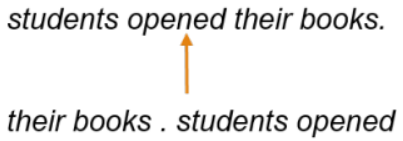
- $Attention(Q, K, V) = Softmax(\frac{QK^\top}{\sqrt{d_k}})V$
  - Query $\mathrm{Q} = [q_1, \ldots, q_n]$, key $\mathrm{K} = [k_1, \ldots, k_n]$, value $\mathrm{V} = [v_1, \ldots, v_n]$
  - Compute weights with each others in the whole sequence



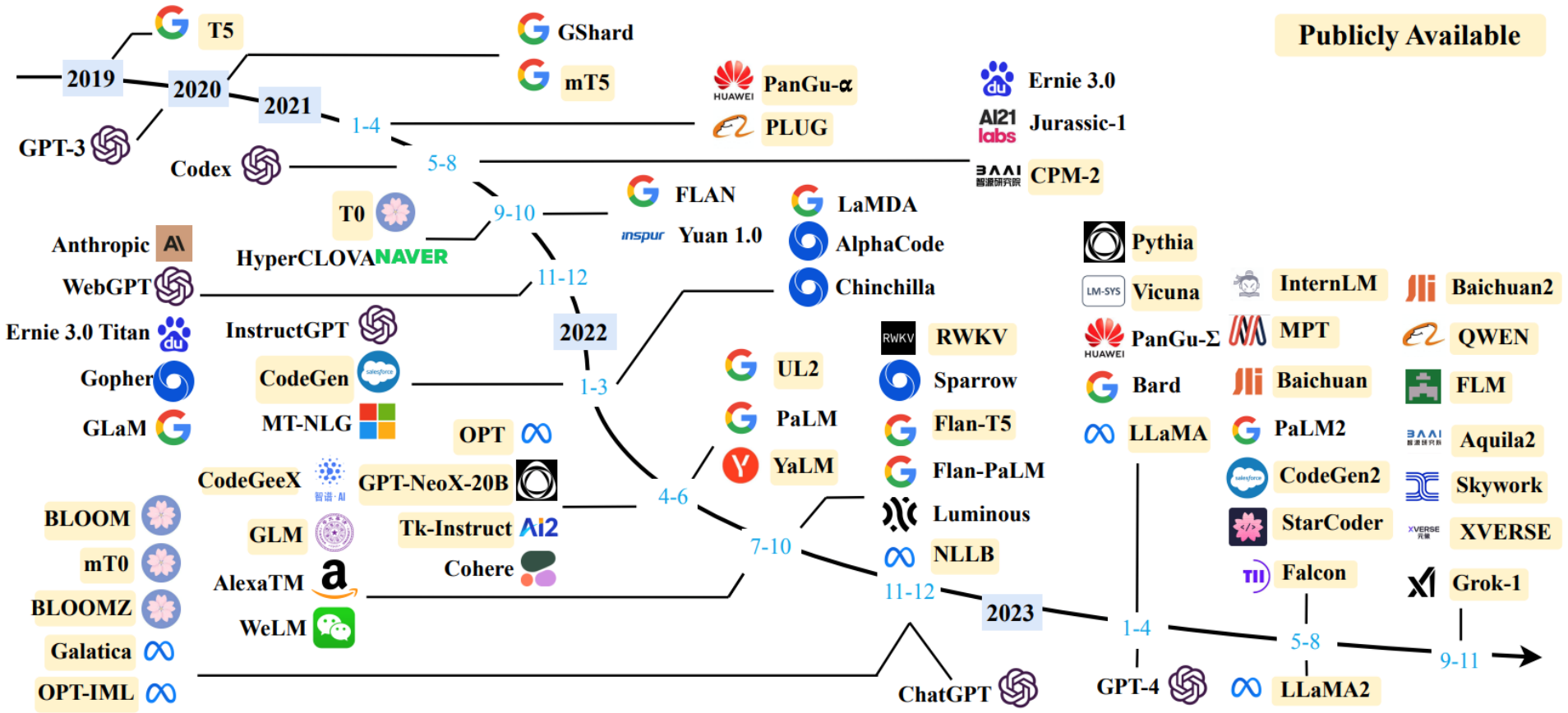
Transformer architecture



Current large language models

	Autoregressive language model (e.g., GPT, GPT-2/3)	**Masked** language model (e.g., BERT, RoBERTa, XLM-R)	**Encoder-Decoder** (e.g., BART, T5)
Model & illustration			
Training objective	Predicting what word comes next given previous words	Predicting masked words given other words in the sequence	Corrupting a sequence and then predicting the original sequence
Example			

Large language models for natural language



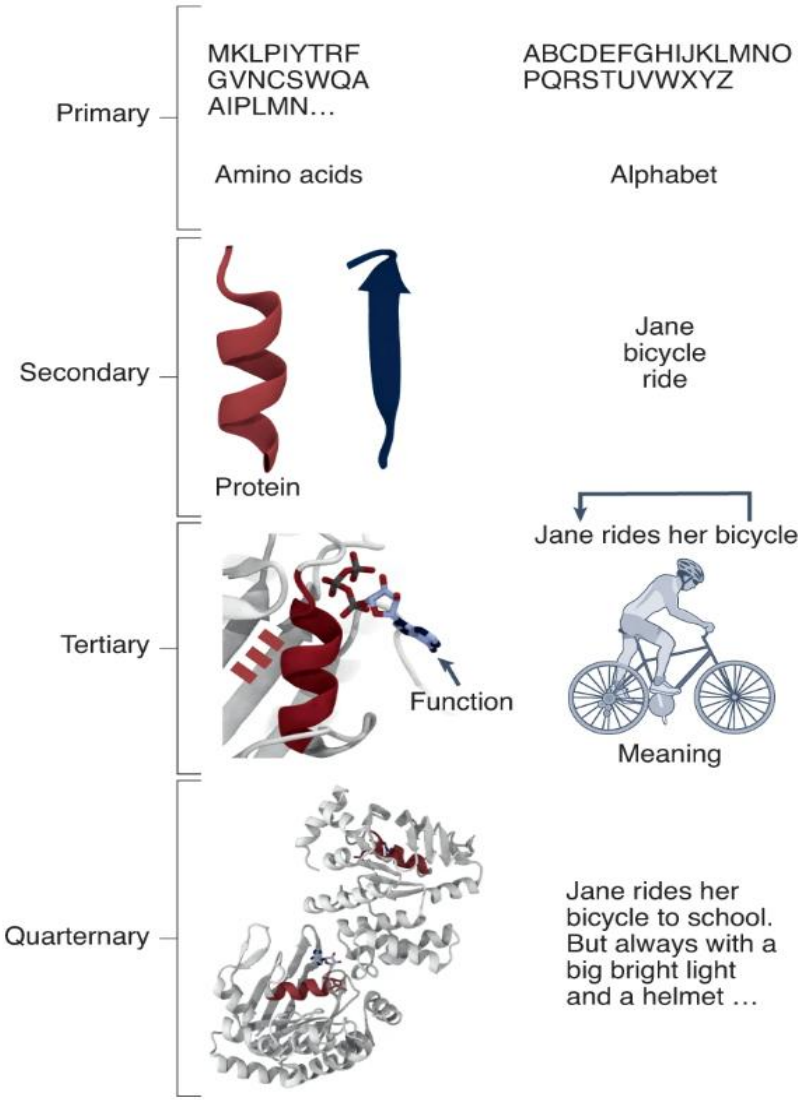
From natural language to life language - Taking proteins as an example

Sequence

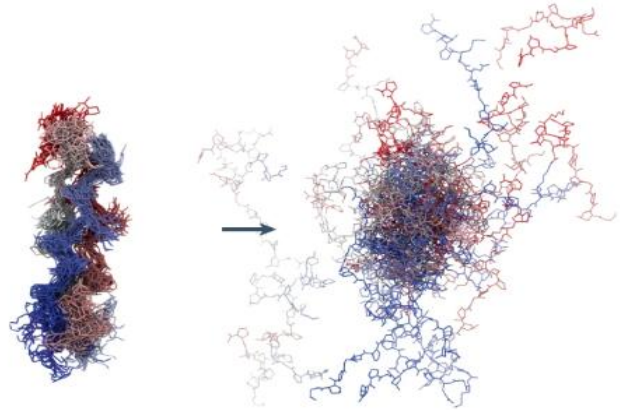
Secondary structure

Tertiary structure

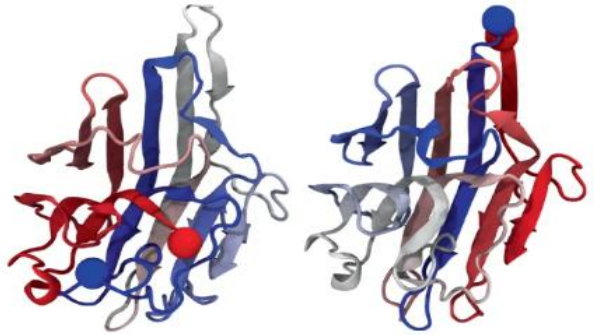
Quaternary structure



Your manuscript is nowt publishable



I am happy when it rains. When it rains, I am happy.



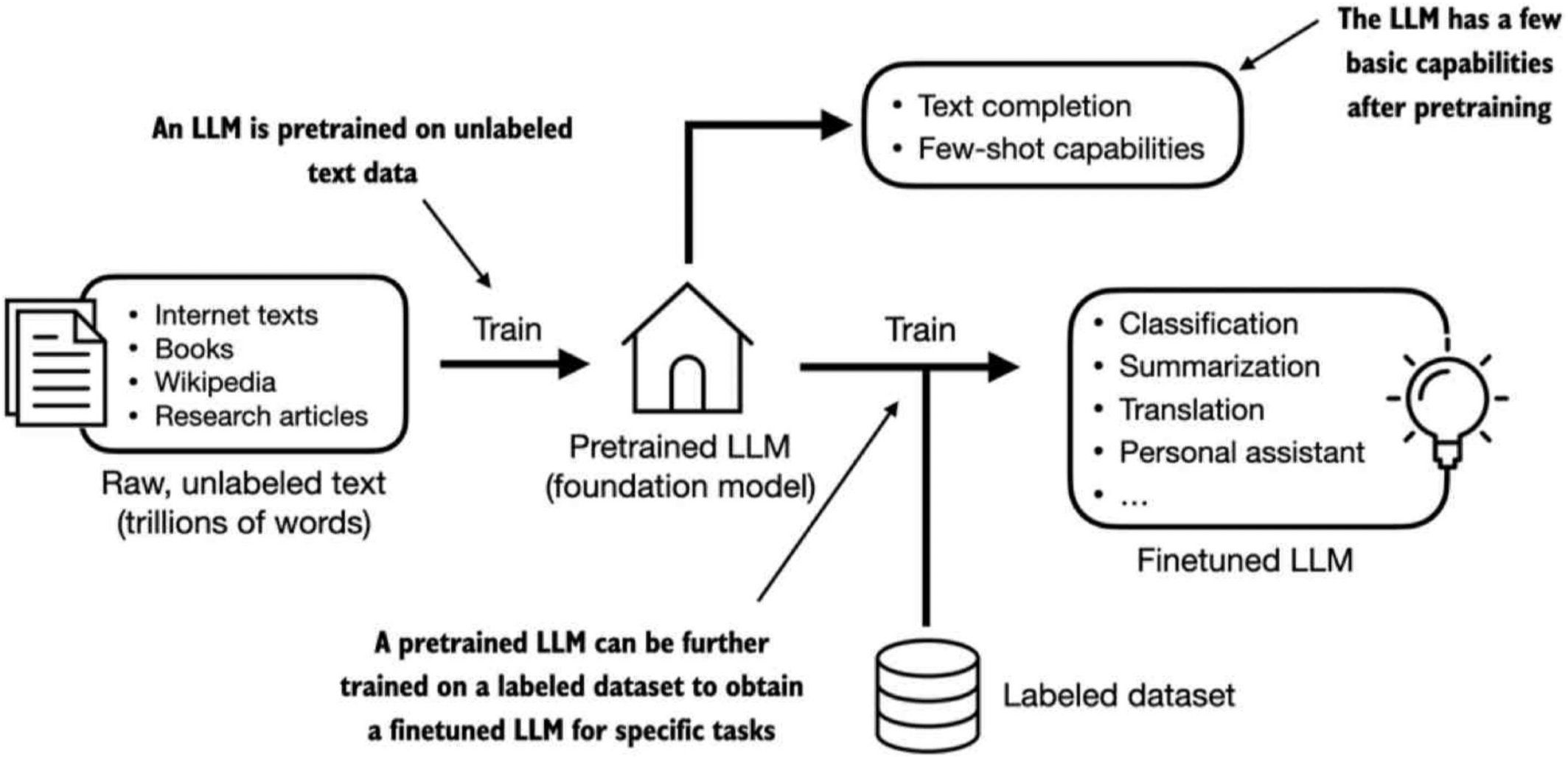
Colourless green ideas sleep furiously.



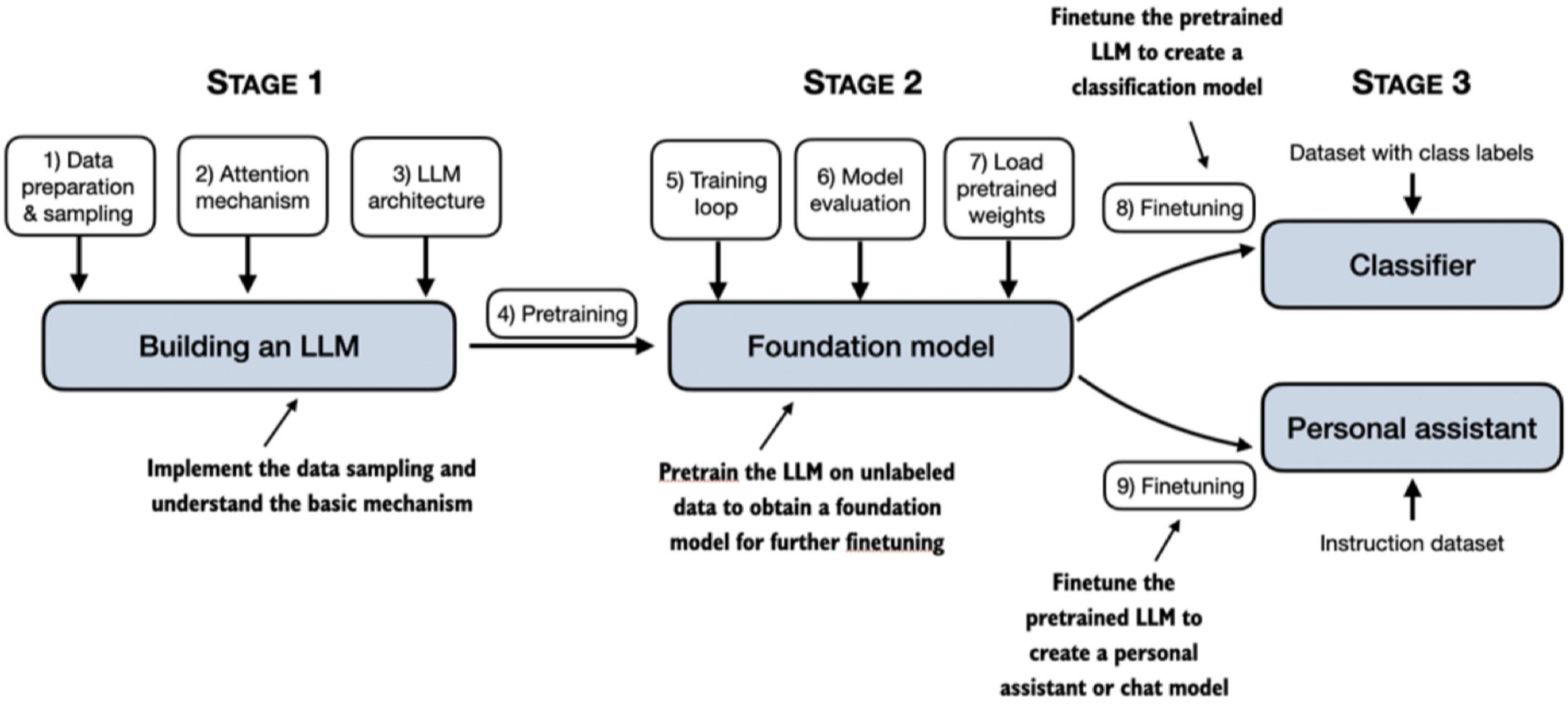
Traditional methods vs LLM for biological sequence modeling

Traditional methods	LLM
Lightweight model	Huge and computation-heavy
Small amount of data	Large scale training data
Rule-based, feature engineering, statistical models (e.g. HMM)	End-to-end self-supervised learning
Strong interpretability but limited generalization	Capture complex biological patterns, generalizable

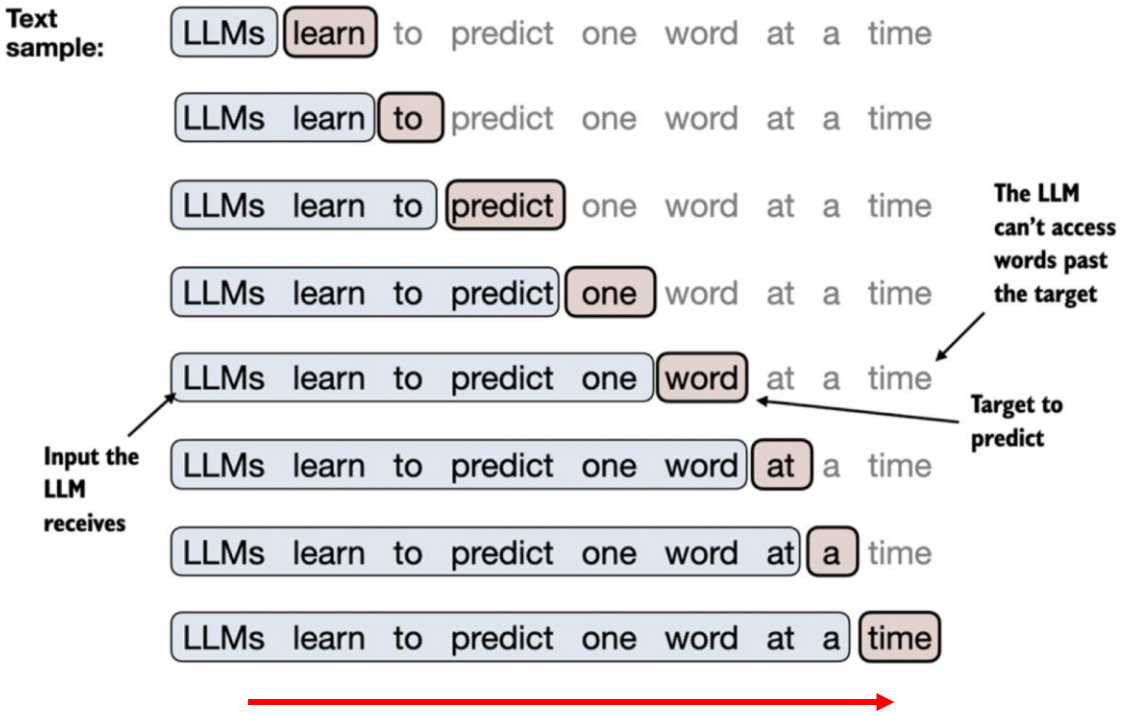
Build a Large Language Model



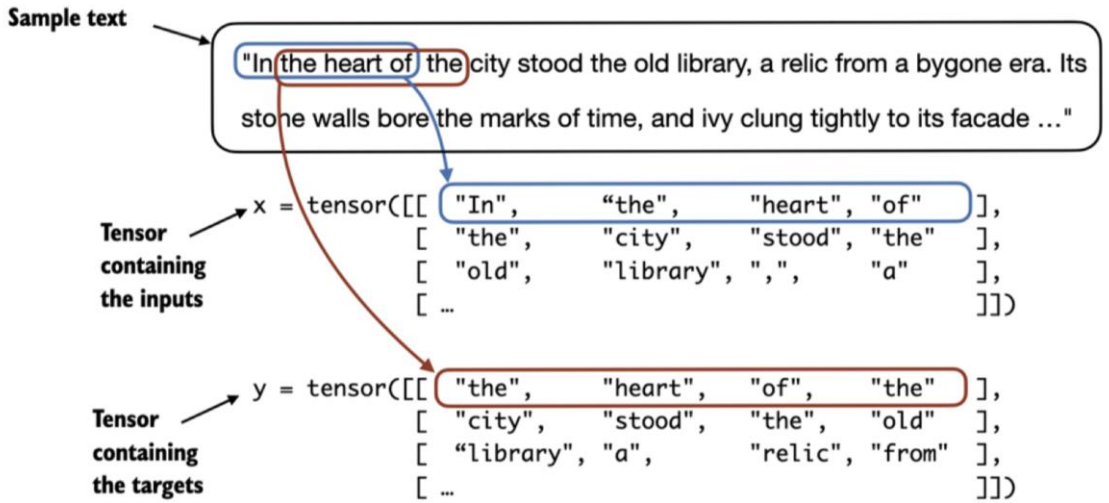
Three stages to build an LLM and finetune it

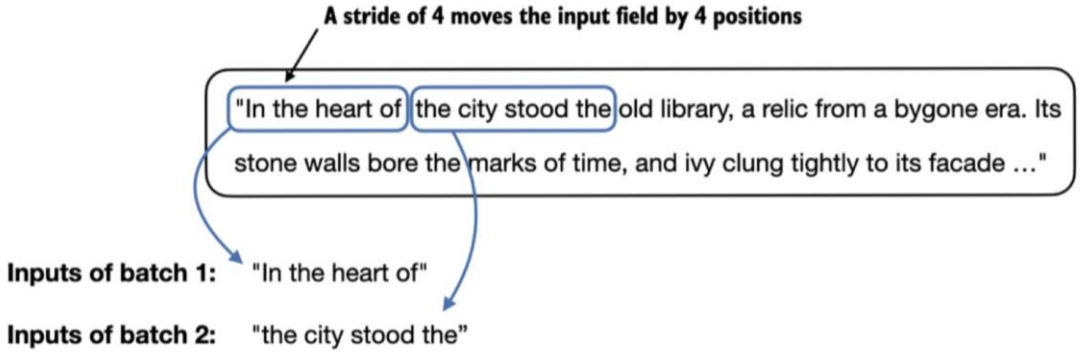


Build text data input-label pairs



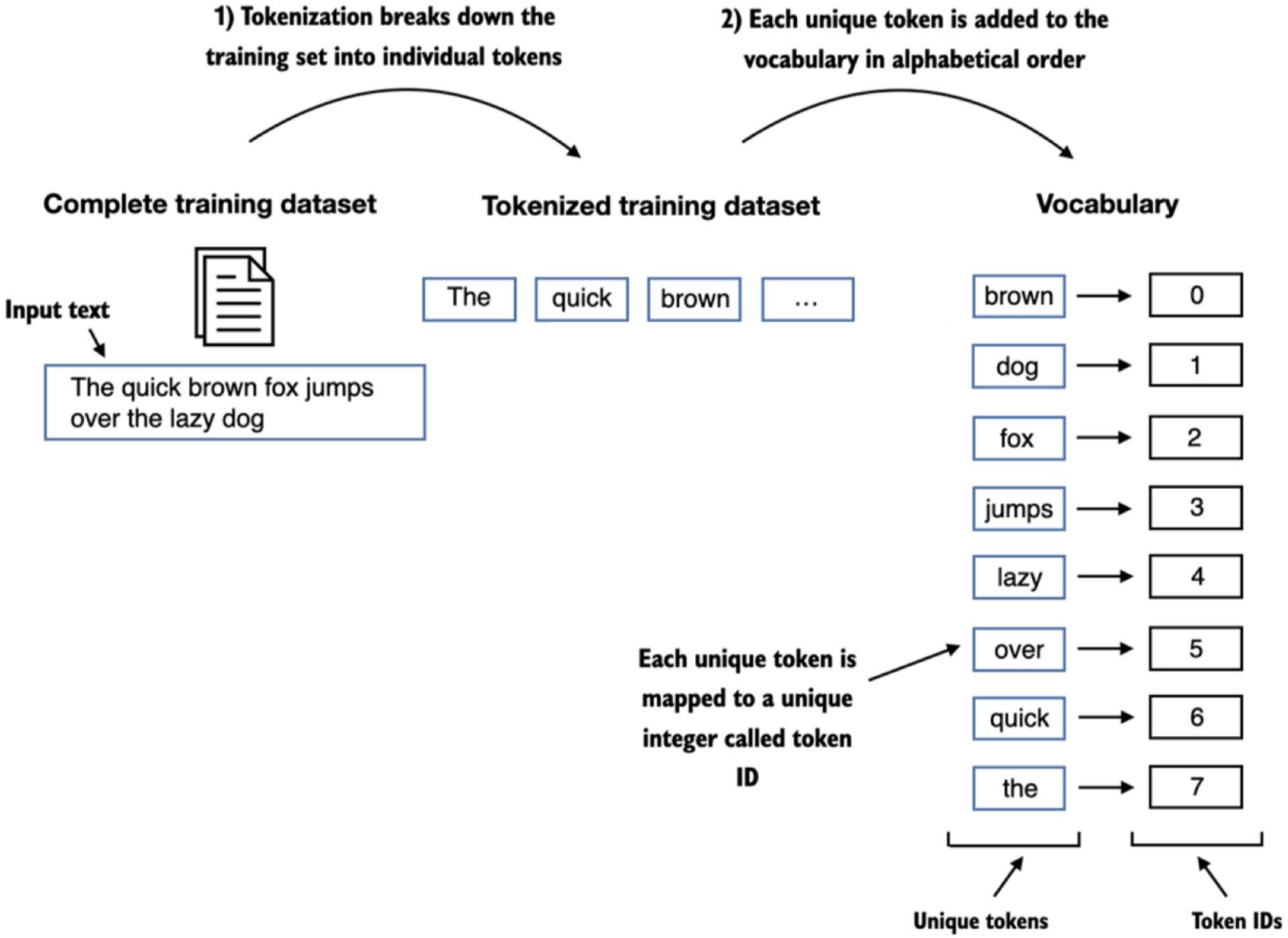
Input and label are random cropping from whole text

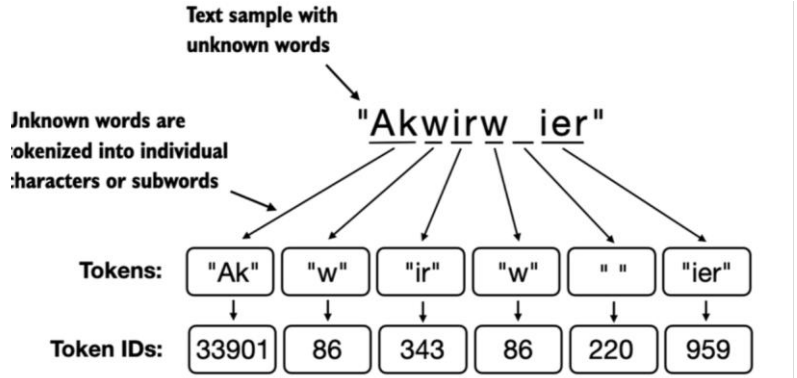




Moving the sliding windows to random cropping input batches

Tokenization of text data

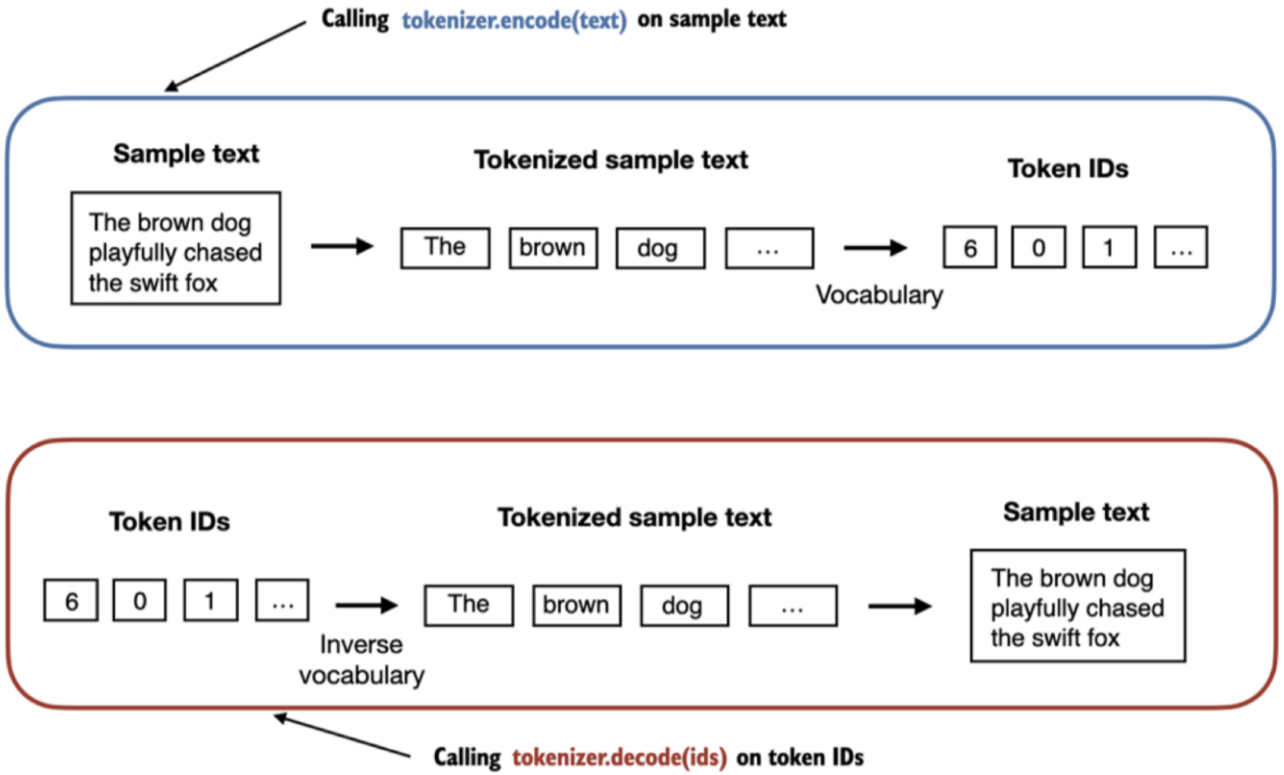




Construct a one-to-one mapping between the text vocabulary and the token ID vocabulary:

- Word vocabulary encoding
- Byte pair encoding

Tokenization of text data




```python
class SimpleTokenizerV1:
      def __init__(self, vocab):
          self.str_to_int = vocab                                          #A
          self.int_to_str = {i:s for s,i in vocab.items()}                 #B

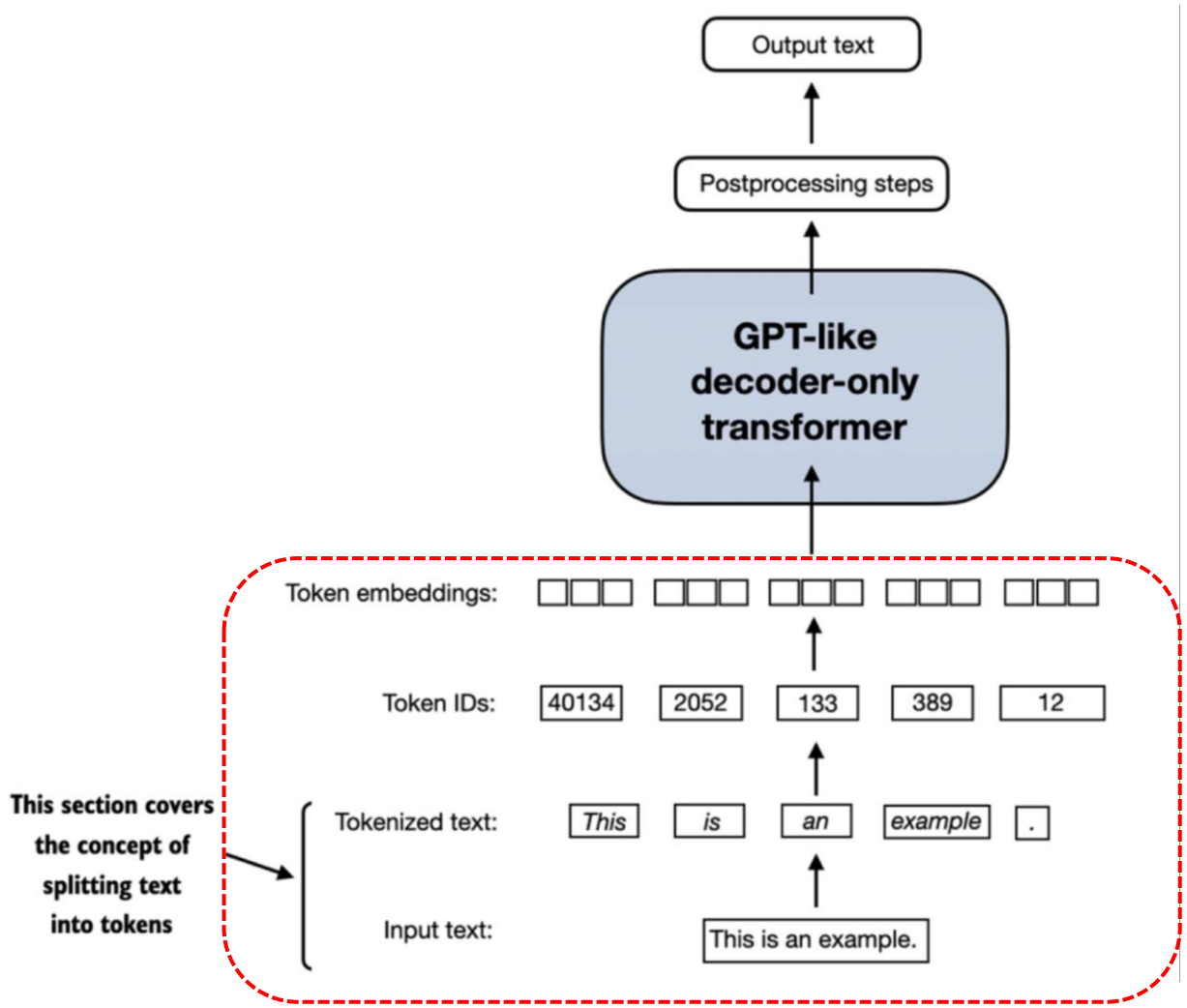
      def encode(self, text):                                              #C
          preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
          preprocessed = [item.strip() for item in preprocessed if item.strip()]
          ids = [self.str_to_int[s] for s in preprocessed]
          return ids

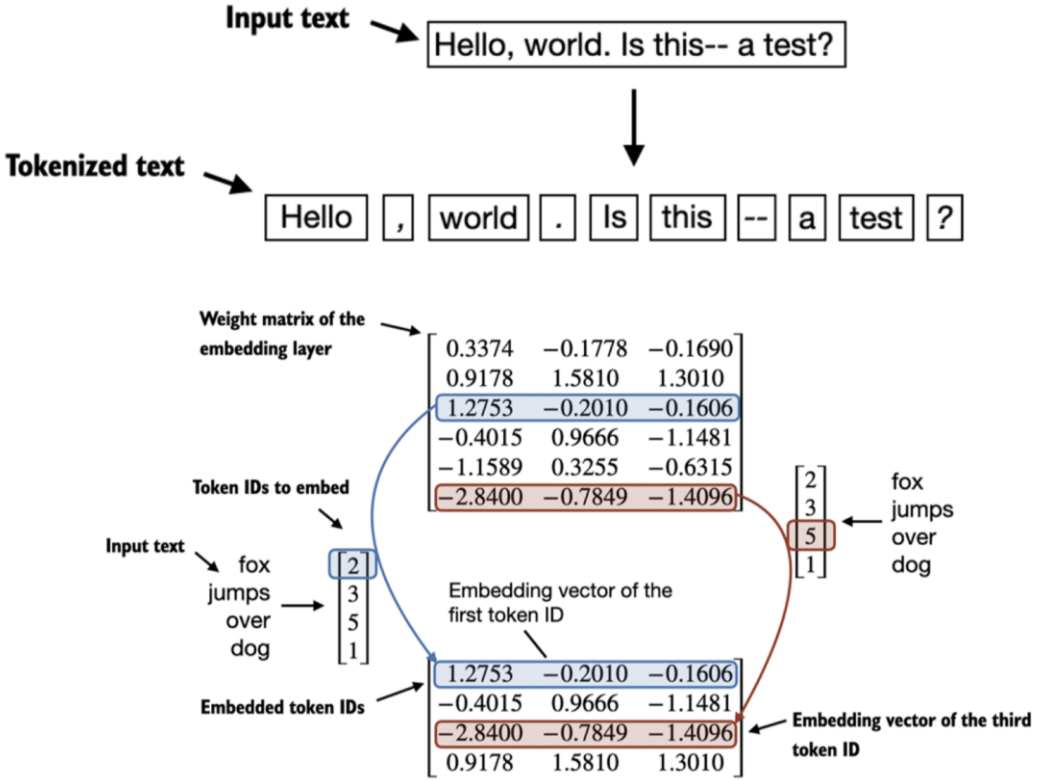
      def decode(self, ids):                                               #D
          text = " ".join([self.int_to_str[i] for i in ids])

          text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)                  #E
          return text
```

Tokenization is reversible

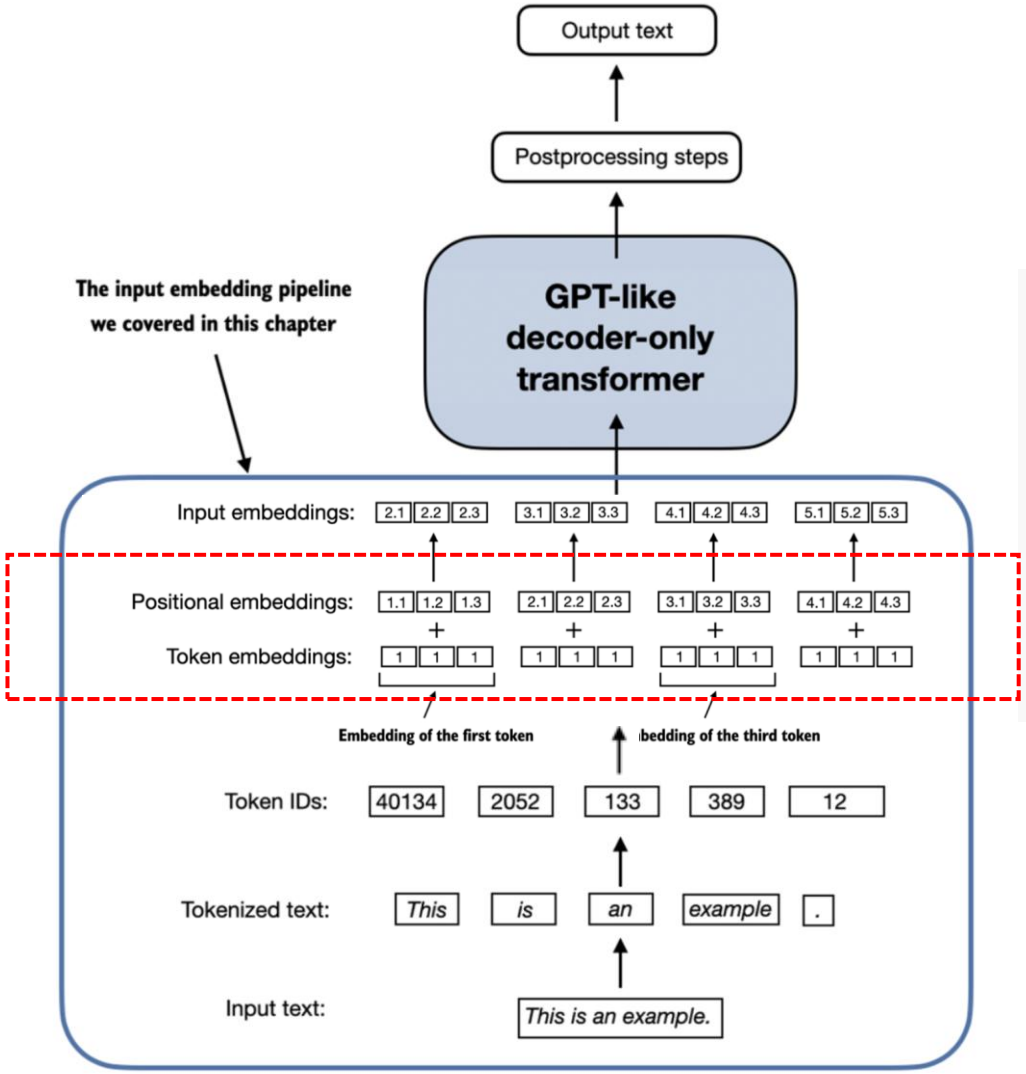
From input text to token embeddings





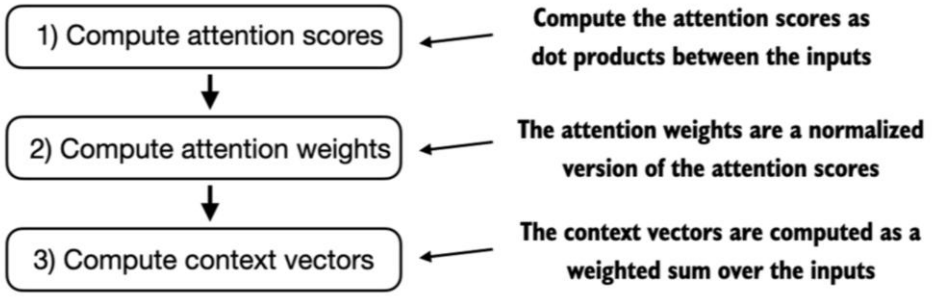
The embedding layer converts identical tokens into identical vectors.

How to add positional embeddings



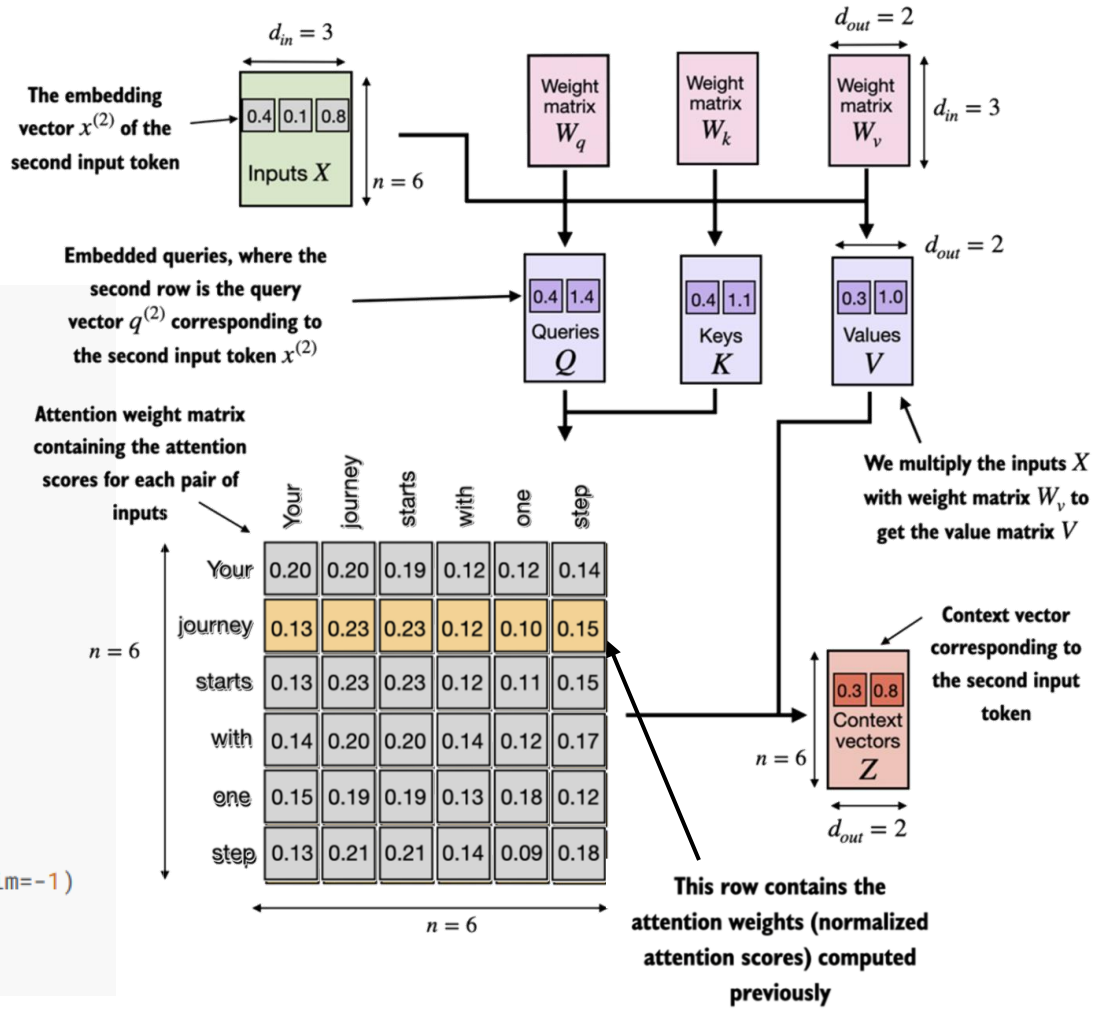

```
vocab_size = 50257
output_dim = 256
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
token_embeddings = token_embedding_layer(inputs)
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
input_embeddings = token_embeddings + pos_embeddings
```

Implement the self-attention mechanism

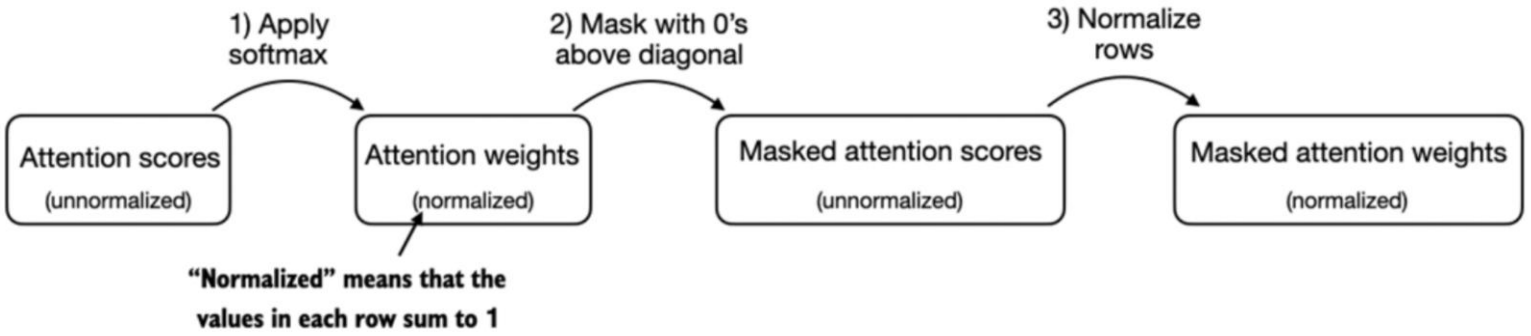


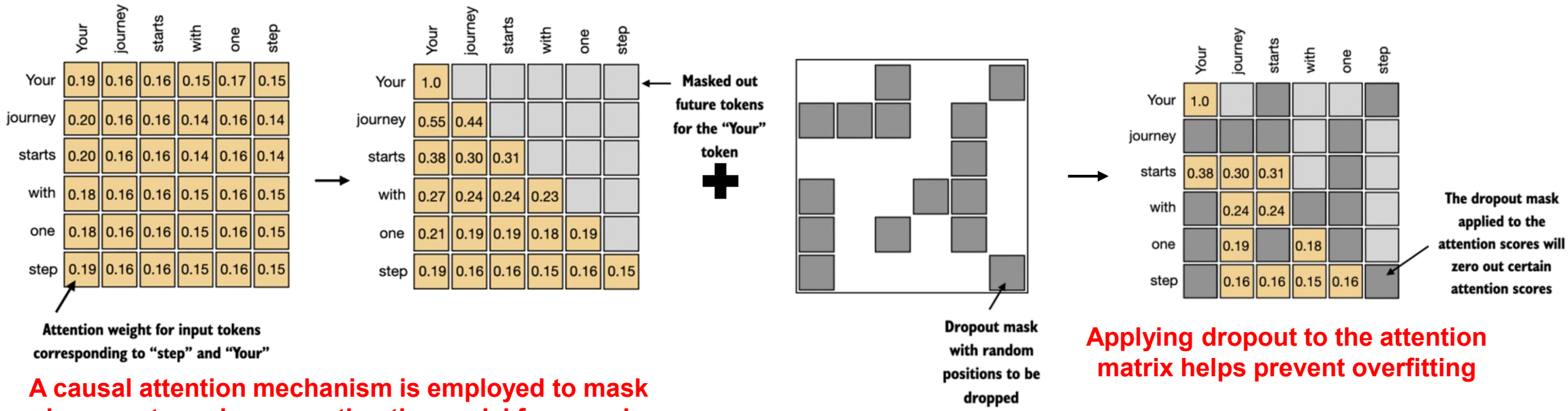

```python
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        context_vec = attn_weights @ values
        return context_vec
```



Mask and dropout of attention weight

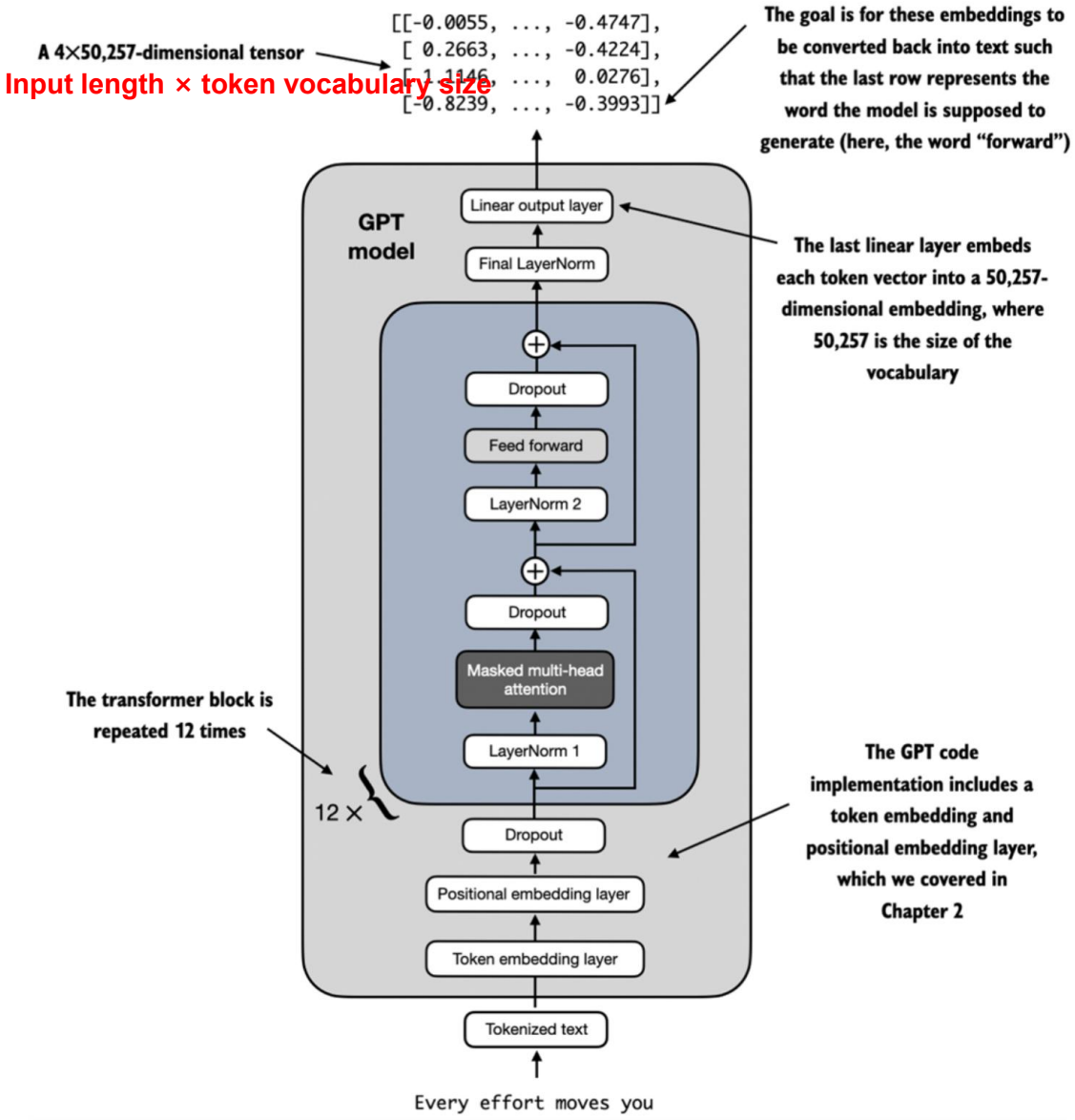




A causal attention mechanism is employed to mask subsequent words, preventing the model from seeing any tokens that appear after the current word.

Applying dropout to the attention matrix helps prevent overfitting

Build the GPT model framework




```python
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

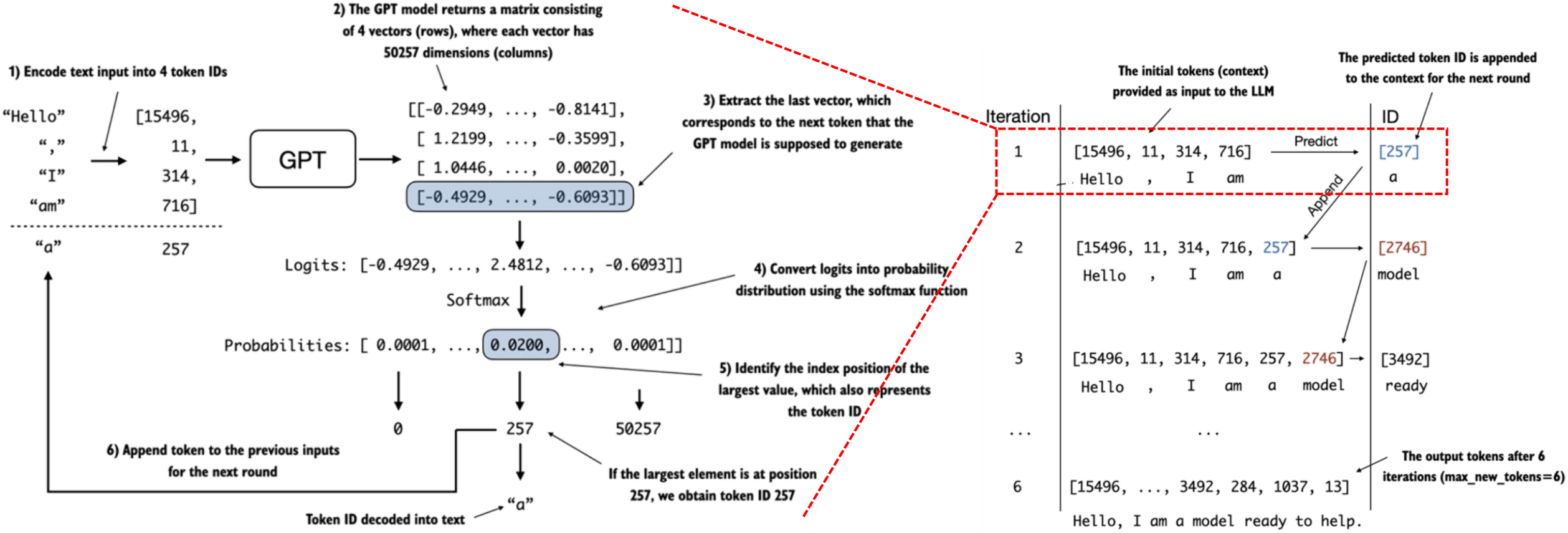
        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

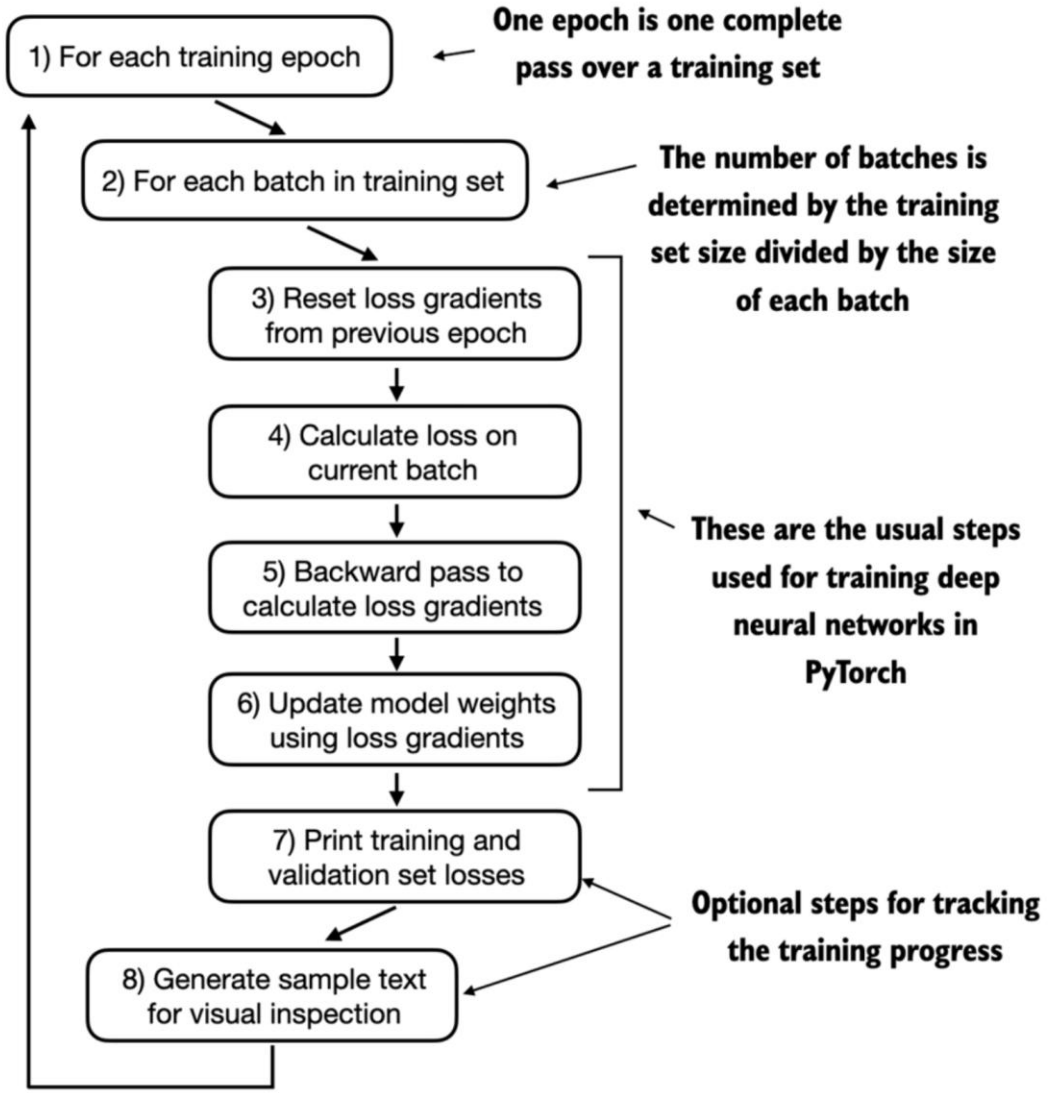
    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)

        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device))
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Processing GPT outputs, from probability matrix to text



Training of LLMs



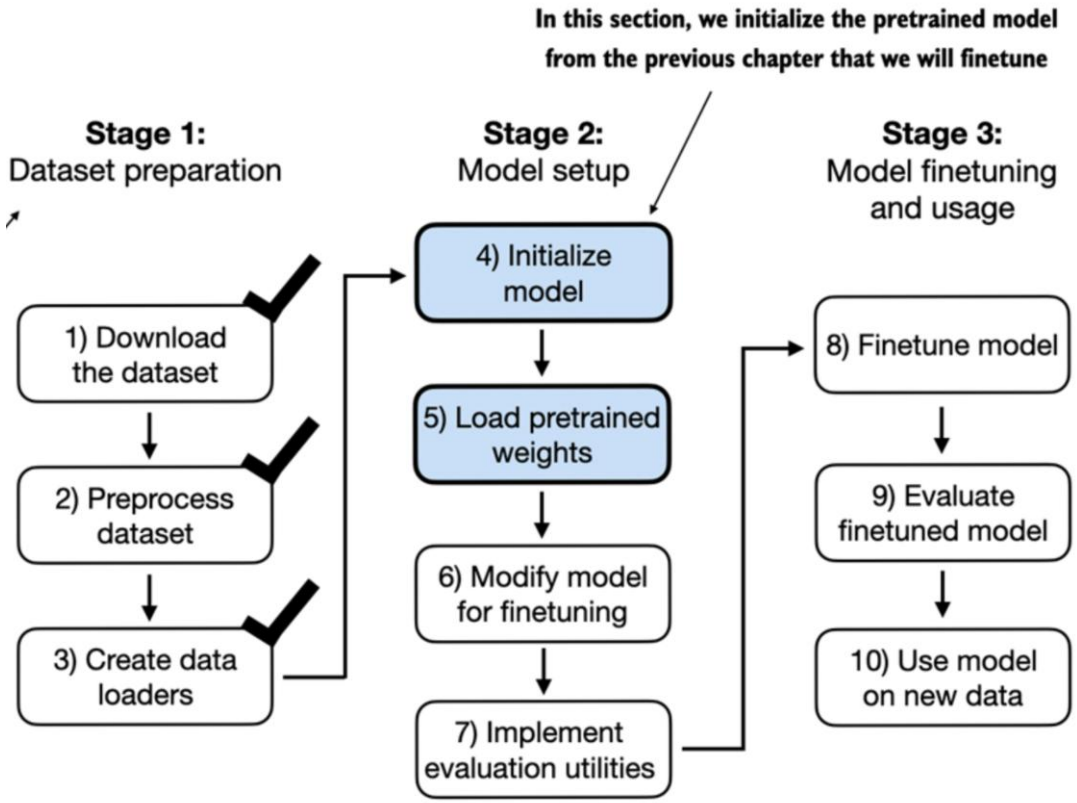

```python
def train_model_simple(model, train_loader, val_loader, optimizer, device, num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []                    #A
    tokens_seen, global_step = 0, -1

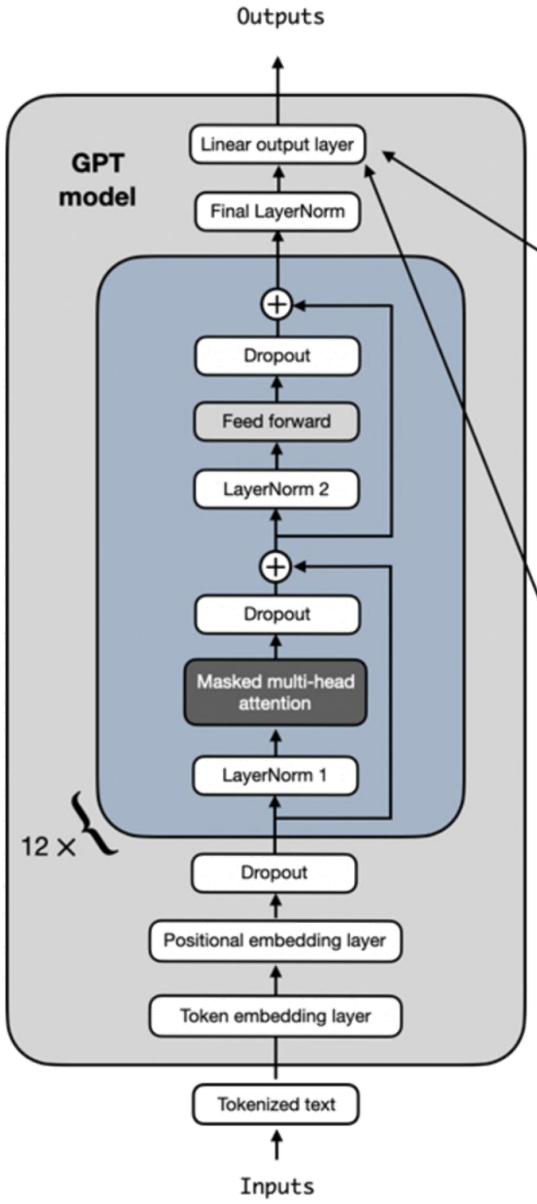
    for epoch in range(num_epochs):                                              #B
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()                                                #C
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward()                                                      #D
            optimizer.step()                                                     #E
            tokens_seen += input_batch.numel()
            global_step += 1

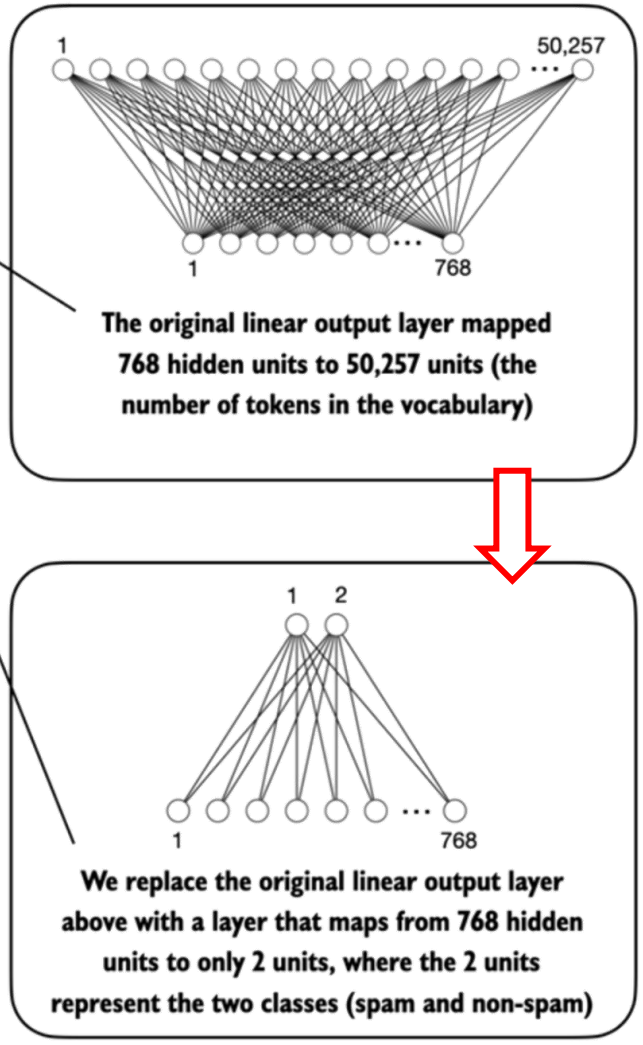
            if global_step % eval_freq == 0:                                     #F
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, Val loss {val_loss:.3f}")

        generate_and_print_sample(                                               #G
            model, tokenizer, device, start_context
        )
    return train_losses, val_losses, track_tokens_seen
```

Finetune of LLMs







The downstream task is a classification task

Finetune of LLMs

```
for param in model.parameters():
    param.requires_grad = False

# Listing 6.7 Adding a classification layer
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(
    in_features=BASE_CONFIG["emb_dim"],
    out_features=num_classes
)

for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

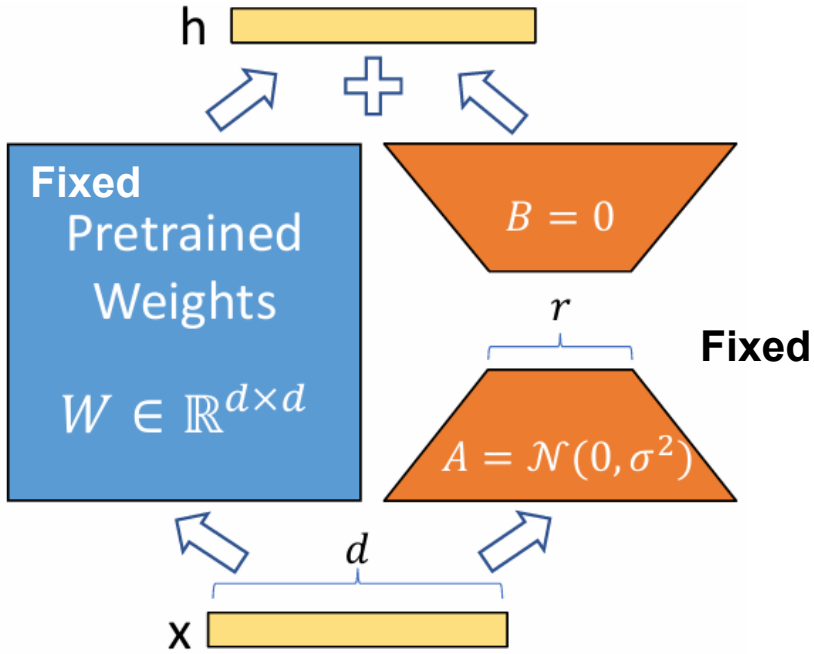
I. Freeze all parameters of the pre-trained model.

II. Modify certain layers of the model, such as the output layer.

III. Determine which layers can have their parameters fine-tuned, such as the last layer of the Transformer and the output layer as shown in the code.

IV. Sometimes, different learning rates or optimizers can also be used to adjust the fine-tuning degree of different layers.

LoRA



$$W' = W + \alpha A \cdot B$$

LoRA (Low-Rank Adaptation) freezes the pre-trained model weights and injects trainable low-rank matrices into Transformer layers, greatly reducing the number of parameters to be trained and effectively preventing overfitting, enabling efficient fine-tuning of large models.

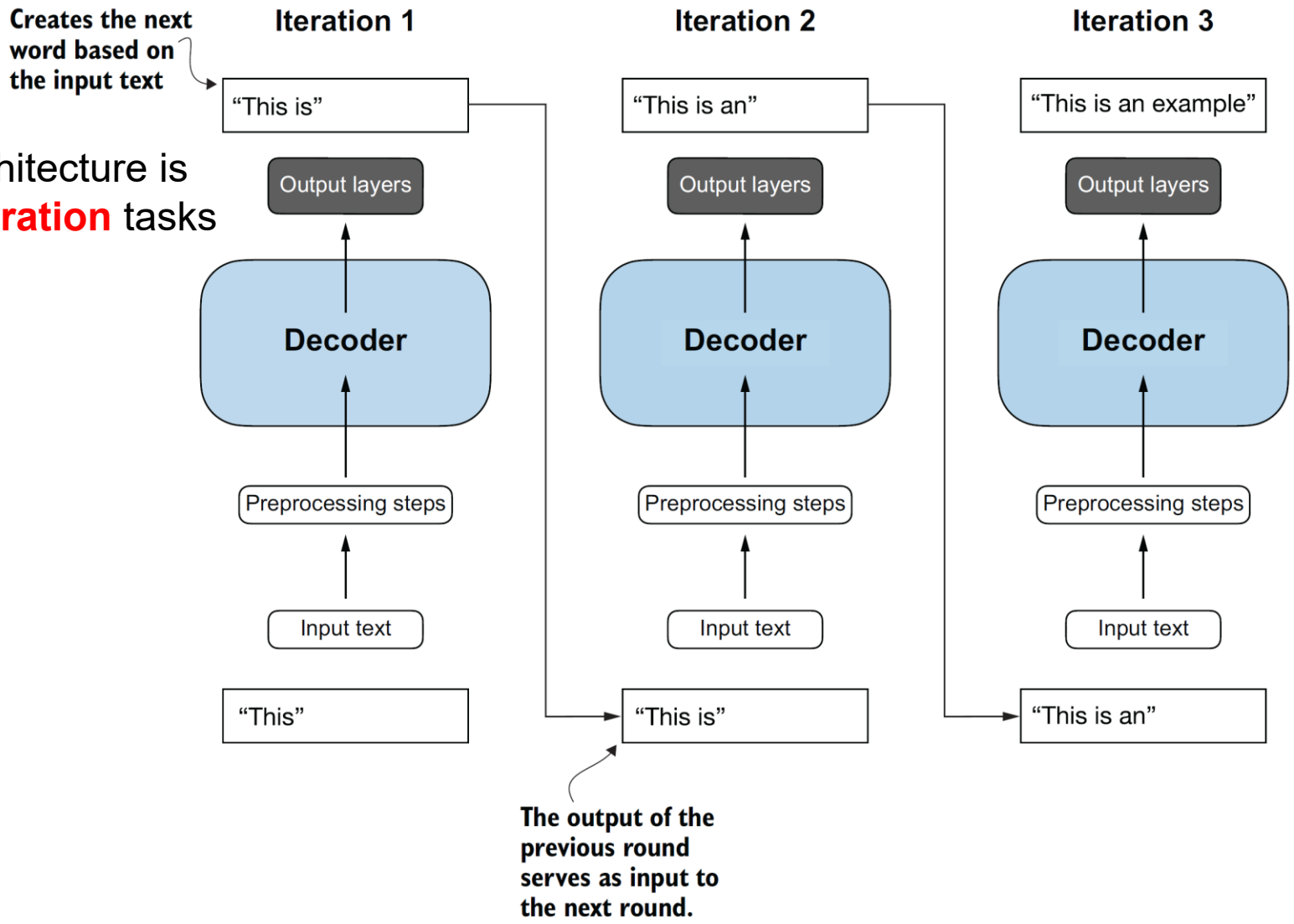
```python
from peft import LoraConfig, TaskType

lora_config = LoraConfig(
        r=64,
        lora_alpha=32,
        lora_dropout=0.05,
        bias="none",
        target_modules=['up_proj', 'gate_proj', 'q_proj', 'o_proj', 'down_proj',
        task_type=TaskType.CAUSAL_LM,
        inference_mode=False  # 训练模式
    )
from peft import get_peft_model
try:
    from openmind import Trainer
except:
    from transformers import Trainer

# 用于确保模型的词嵌入层参与训练
model.enable_input_require_grads()
# 应用 PEFT 配置到模型
model = get_peft_model(model,lora_config)
model.print_trainable_parameters()
```

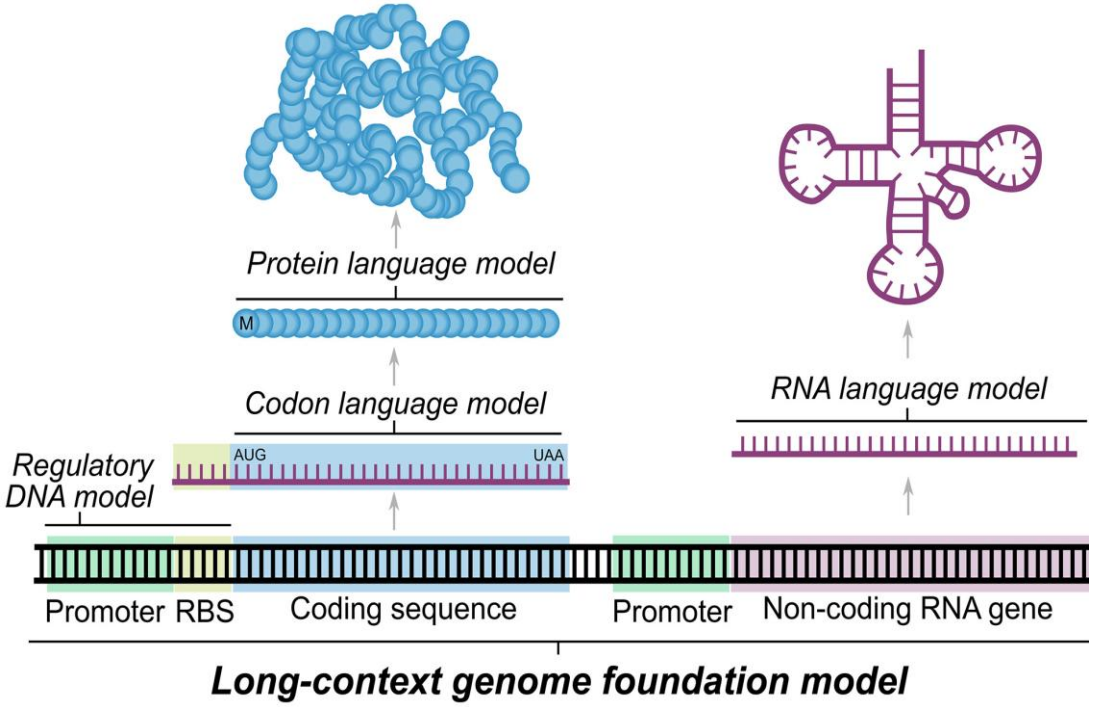
GPT-style architecture for sequence generation

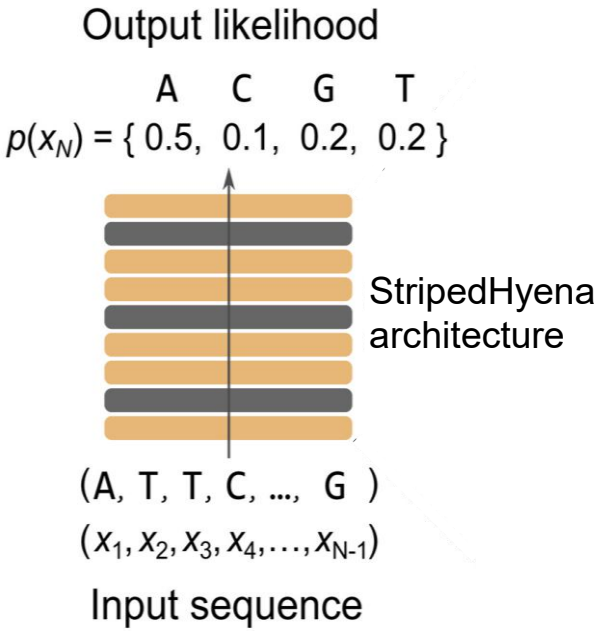
The GPT-style architecture is conducive to **generation** tasks



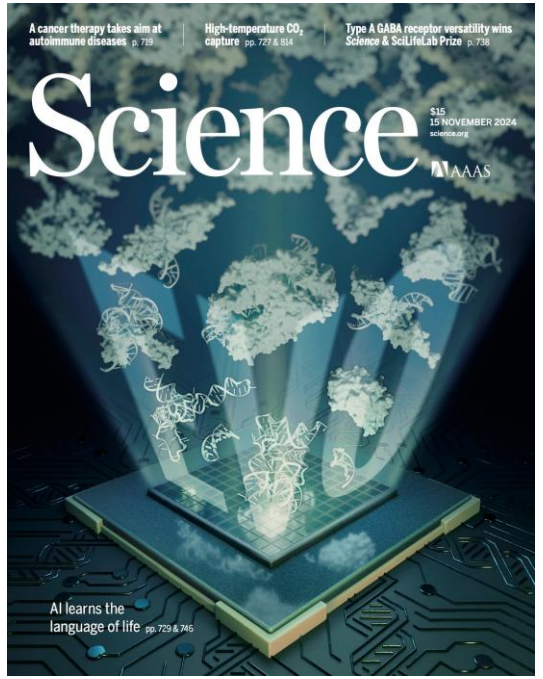
Evo: genomic foundation model for sequence modeling

Evo: genomic foundation model across prokaryotic life



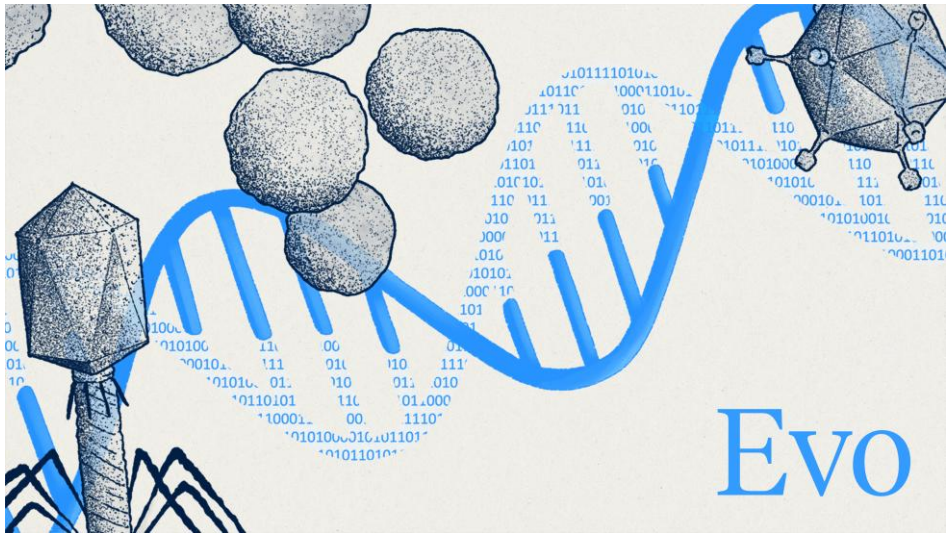


StripedHyena architecture



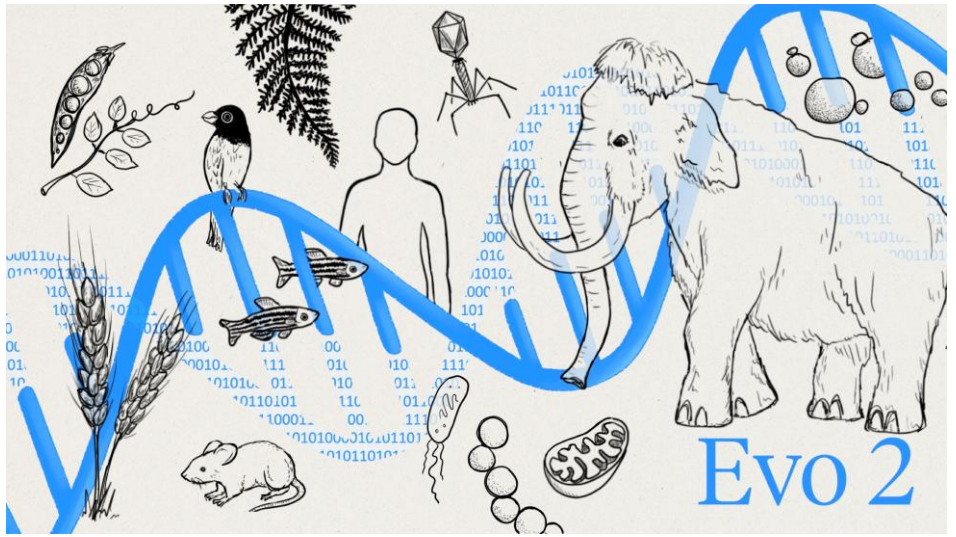
From Evo to Evo2

- StripedHyena model architecture
- Training data: 2.7 million prokaryotic and phage genomes, 340B tokens
- Context length: 8k, 131k
- Parameters: 7 billion



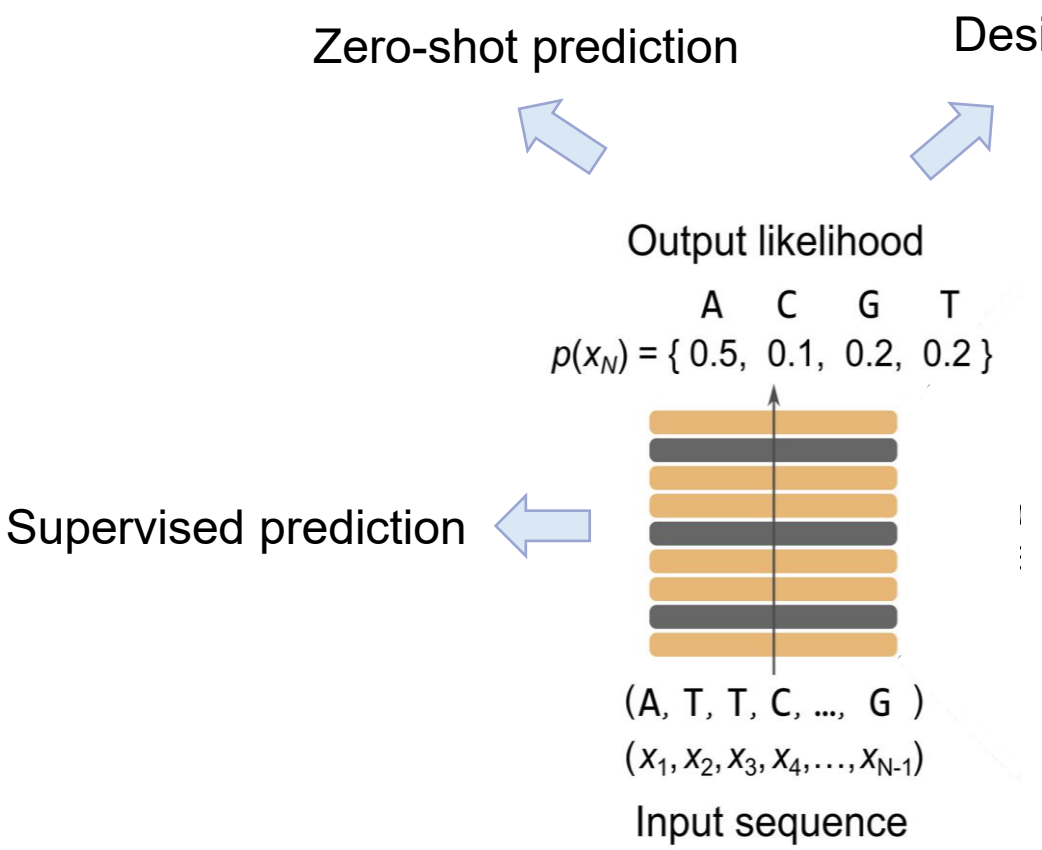
Science, 2024

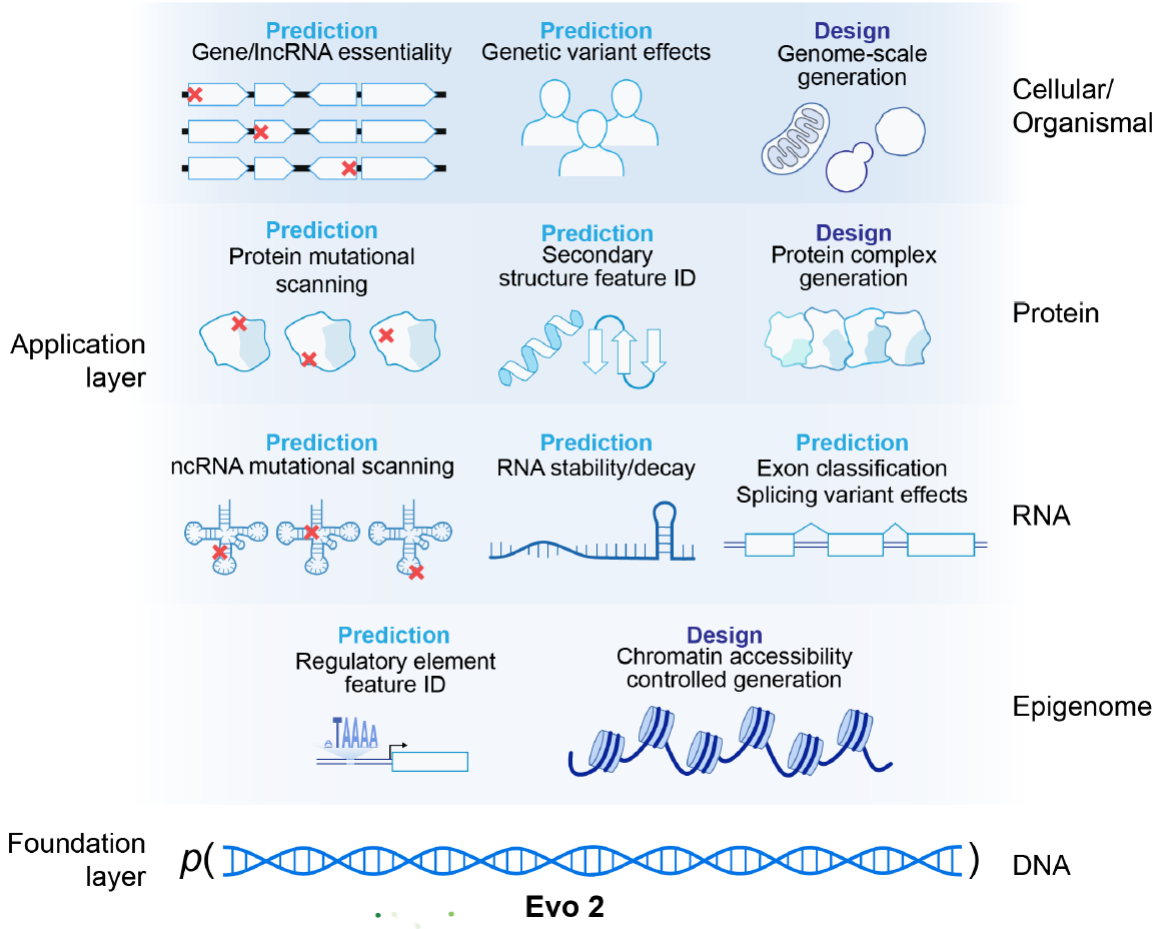
- StripedHyena2 model architecture
- Training data: bacteria, archaea, eukarya, and bacteriophage genomes, 9.3T tokens
- Context length: 8k, 1M
- Parameters: 7 billion, 40 billion



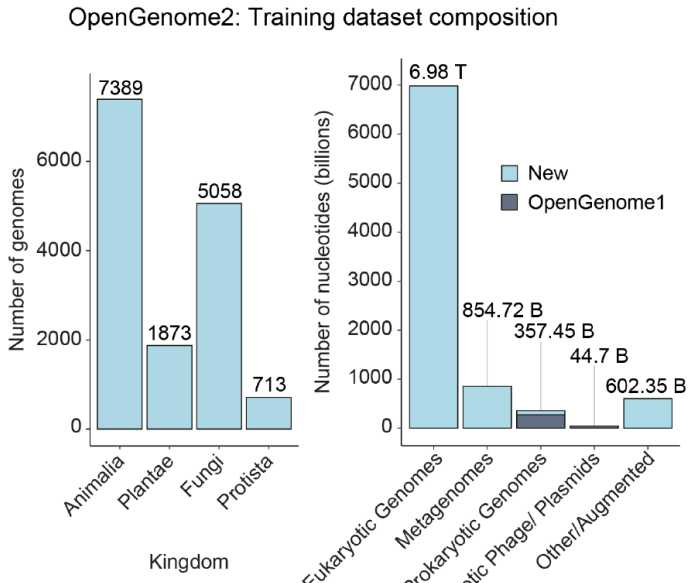
bioRxiv, 2025

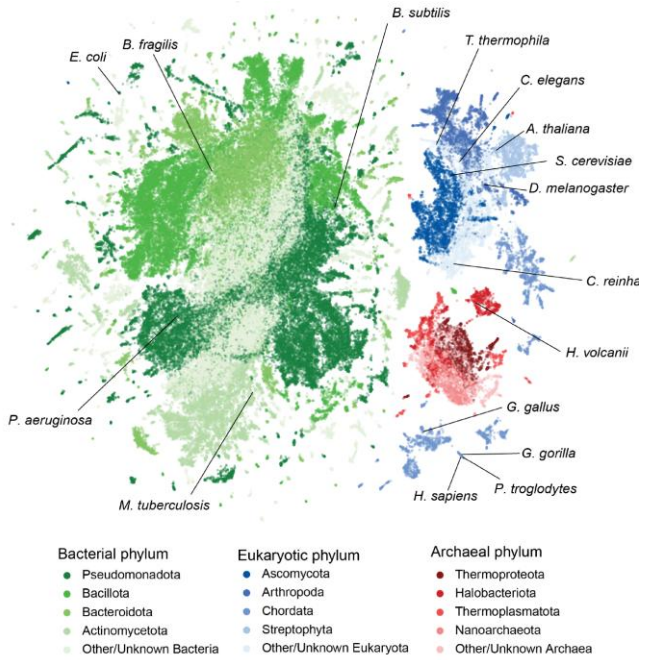
Overview of Evo2

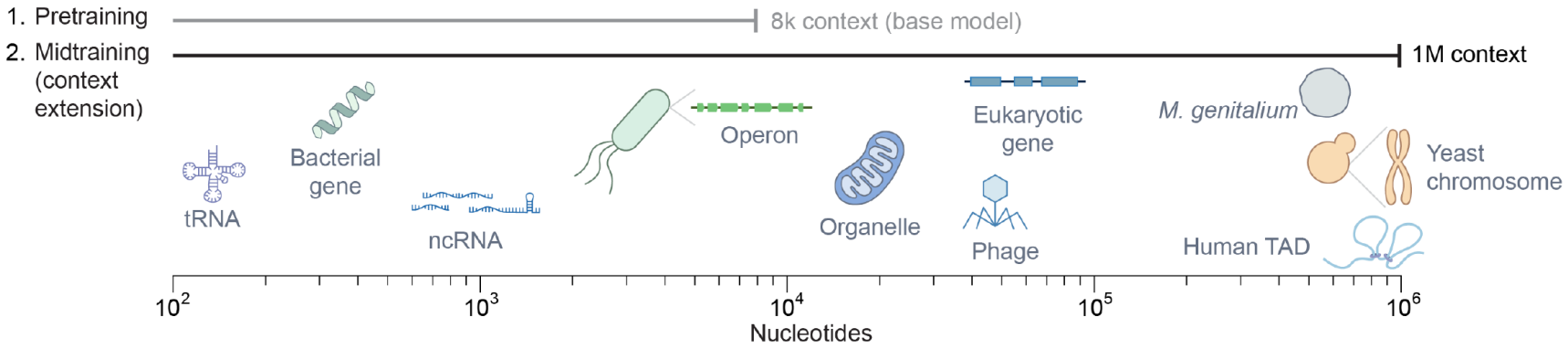




Training data and two-phase training strategy





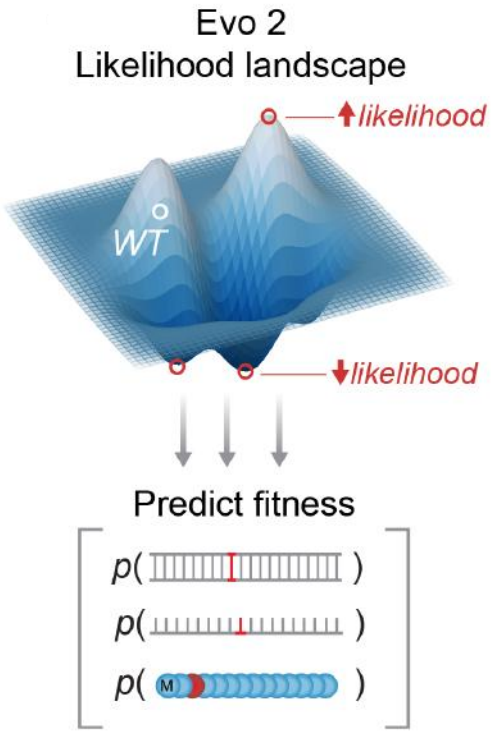


Learn functional genetic elements

Learn relationship between elements across long genomic distances

Zero-shot prediction

- **Zero-shot prediction:** By learning the likelihood of sequences across vast evolutionary training datasets, biological sequence models can learn how mutational effects correlate with biological functions without any task-specific finetuning or supervision.



Forward

Evo 2 can be used to score the likelihoods across a DNA sequence.

```
import torch
from evo2 import Evo2

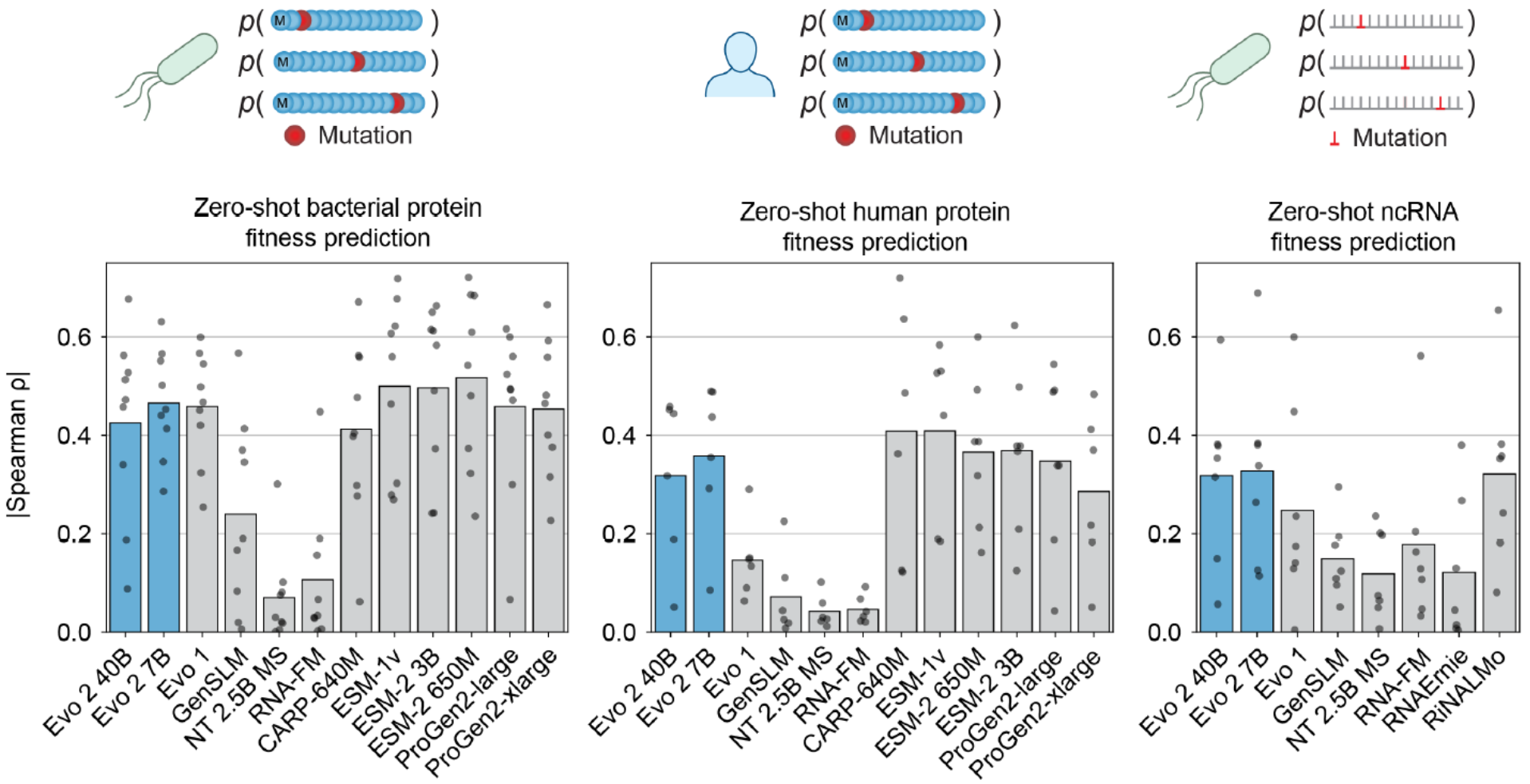
evo2_model = Evo2('evo2_7b')

sequence = 'ACGT'
input_ids = torch.tensor(
    evo2_model.tokenizer.tokenize(sequence),
    dtype=torch.int,
).unsqueeze(0).to('cuda:0')

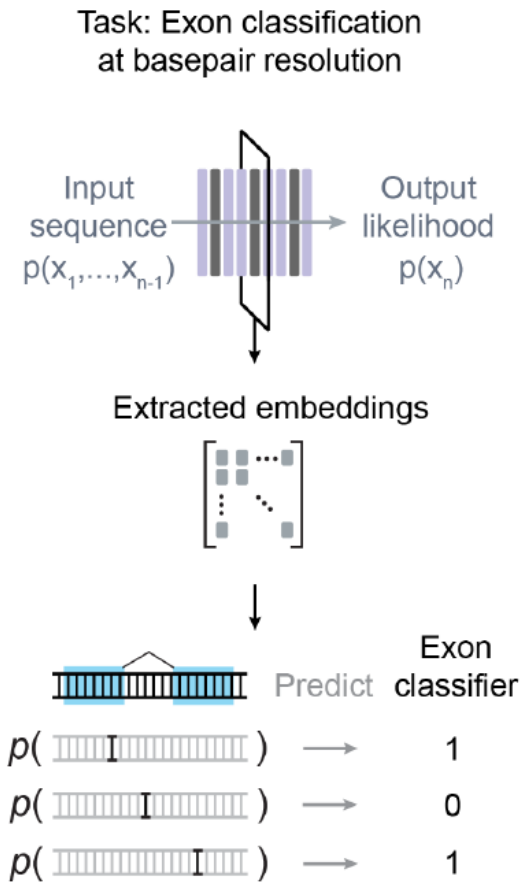
outputs, _ = evo2_model(input_ids)
logits = outputs[0]

print('Logits: ', logits)
print('Shape (batch, length, vocab): ', logits.shape)
```

Evo 2 learns mutational effect on protein and RNA



Use Evo2 embedding for supervised prediction



Embeddings

Evo 2 embeddings can be saved for use downstream. We find that intermediate embeddings work better than final embeddings, see our paper for details.

```
import torch
from evo2 import Evo2

evo2_model = Evo2('evo2_7b')

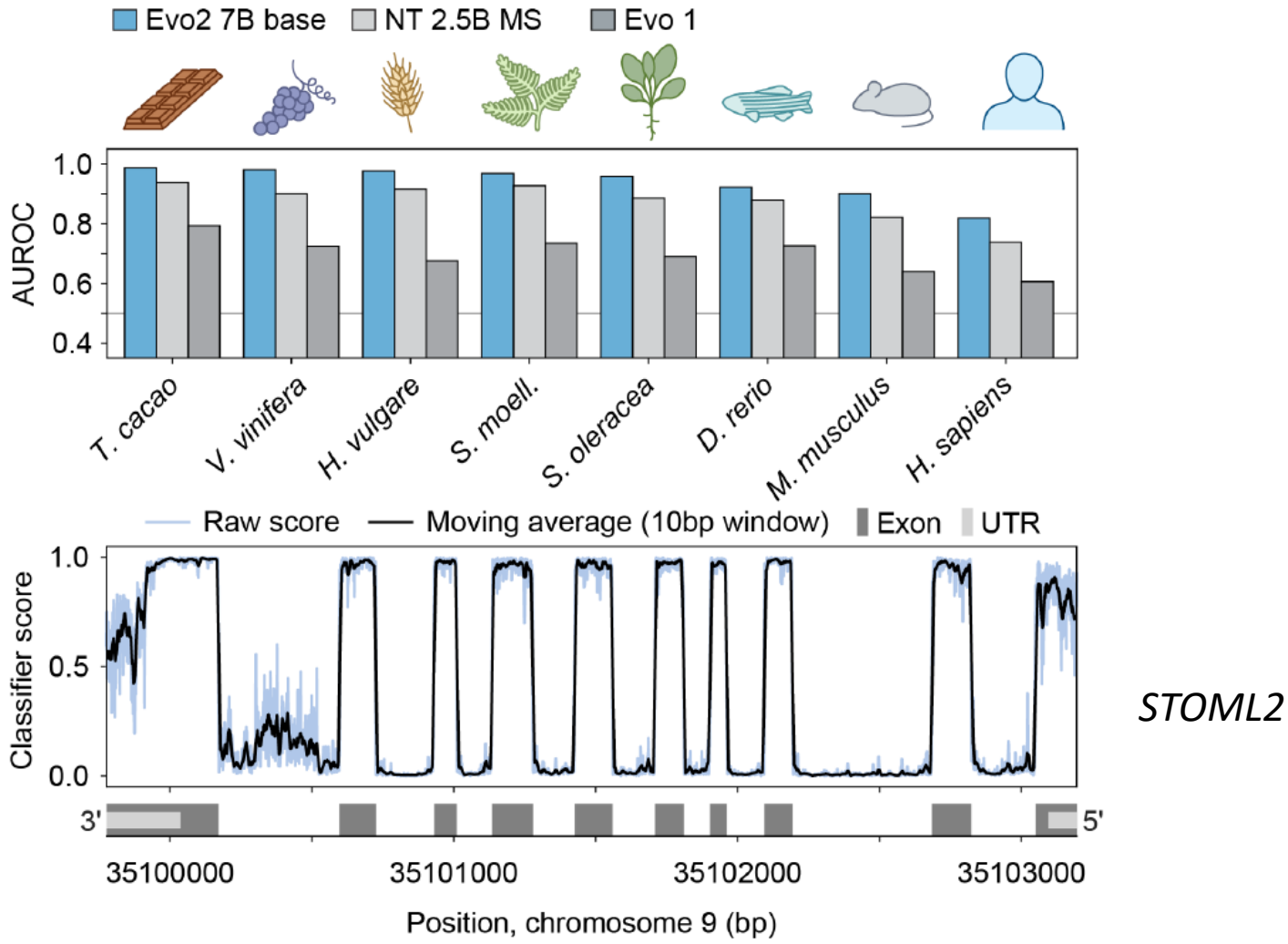
sequence = 'ACGT'
input_ids = torch.tensor(
    evo2_model.tokenizer.tokenize(sequence),
    dtype=torch.int,
).unsqueeze(0).to('cuda:0')

layer_name = 'blocks.28.mlp.l3'

outputs, embeddings = evo2_model(input_ids, return_embeddings=True, layer_names=[layer_name])

print('Embeddings shape: ', embeddings[layer_name].shape)
```

Evo 2 captures the canonical exonic and intronic patterns



STOML2

Evo2 for sequence generation

Generation

Evo 2 can generate DNA sequences based on prompts.

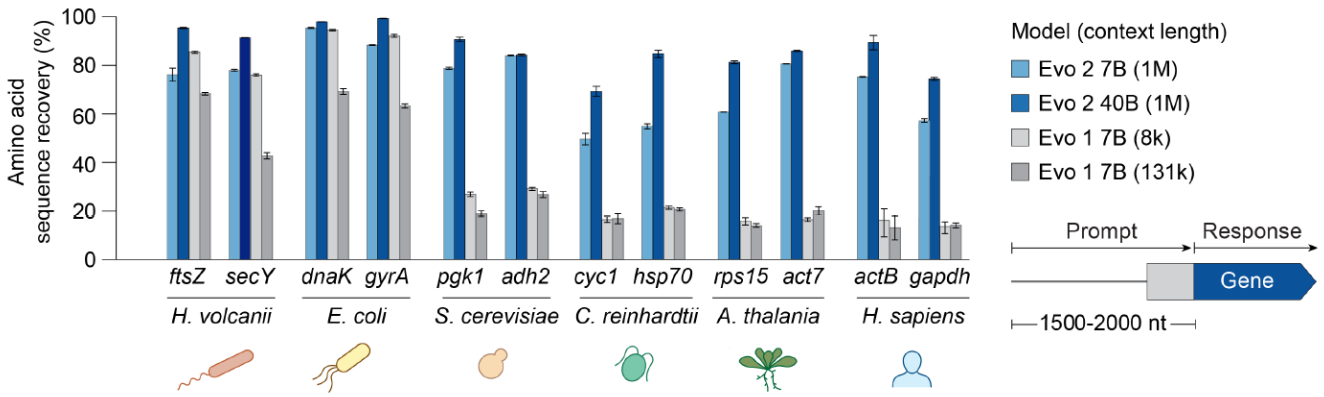
```
from evo2 import Evo2

evo2_model = Evo2('evo2_7b')

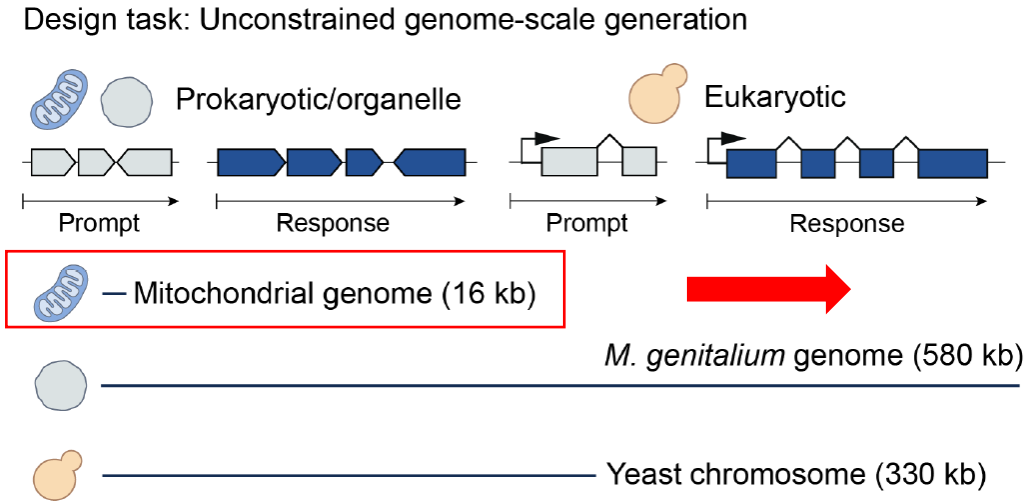
output = evo2_model.generate(prompt_seqs=["ACGT"], n_tokens=400, temperature=1.0, top_k=4)

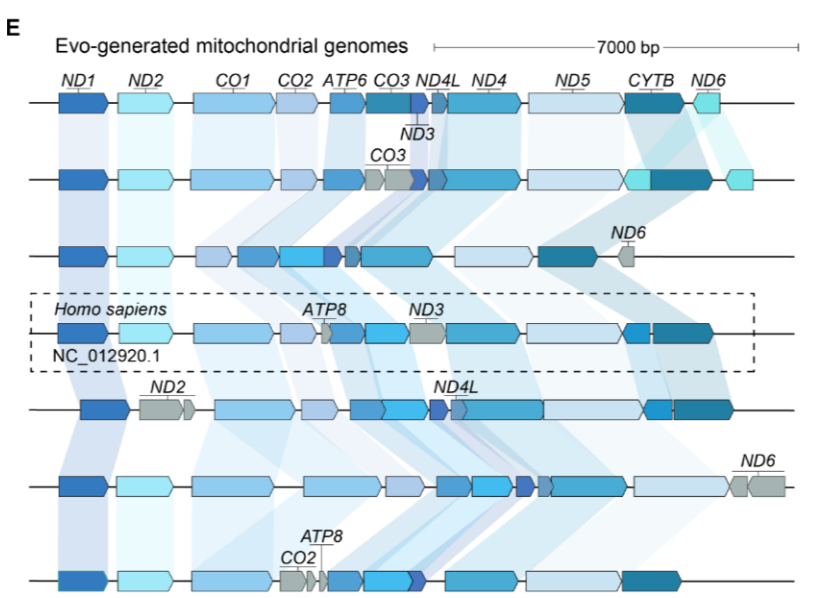
print(output.sequences[0])
```

- Evo 2 achieves high accuracy in gene sequence completion



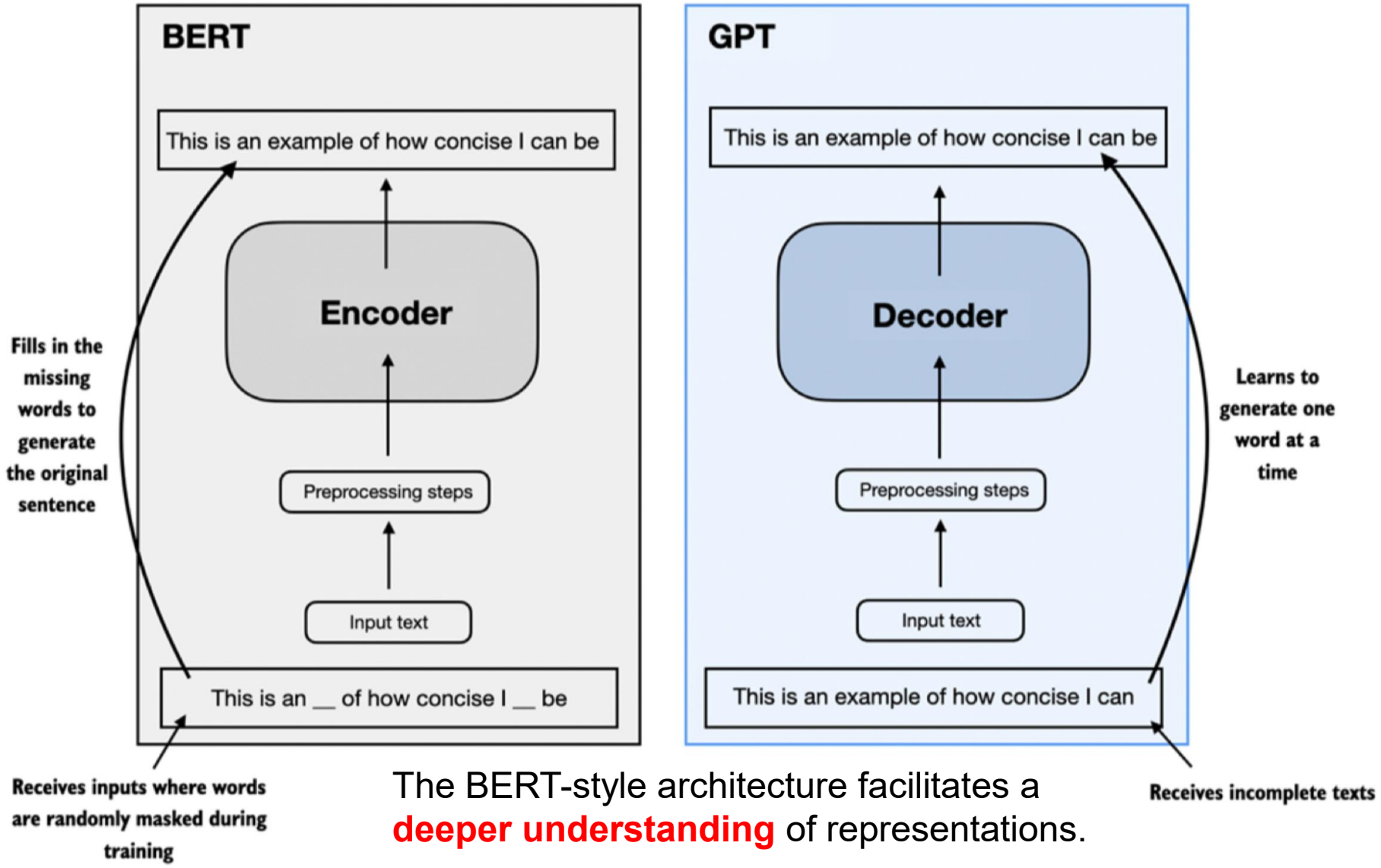
Evo 2 enables genome-scale generation





MitoZ annotated

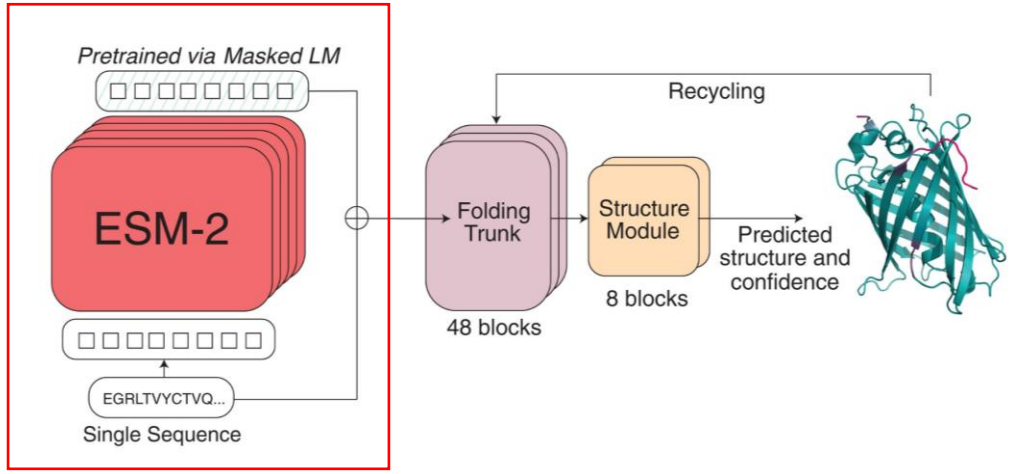
BERT-style architecture for sequence embedding

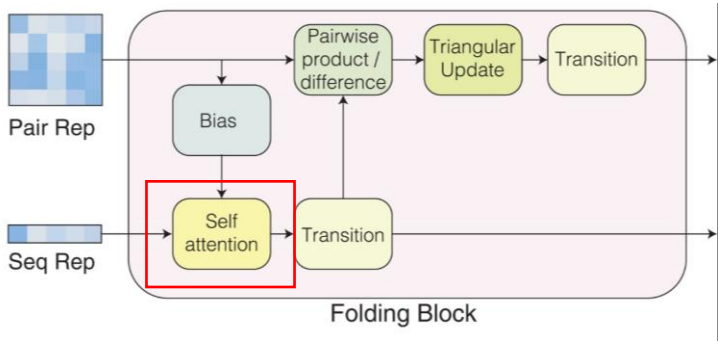


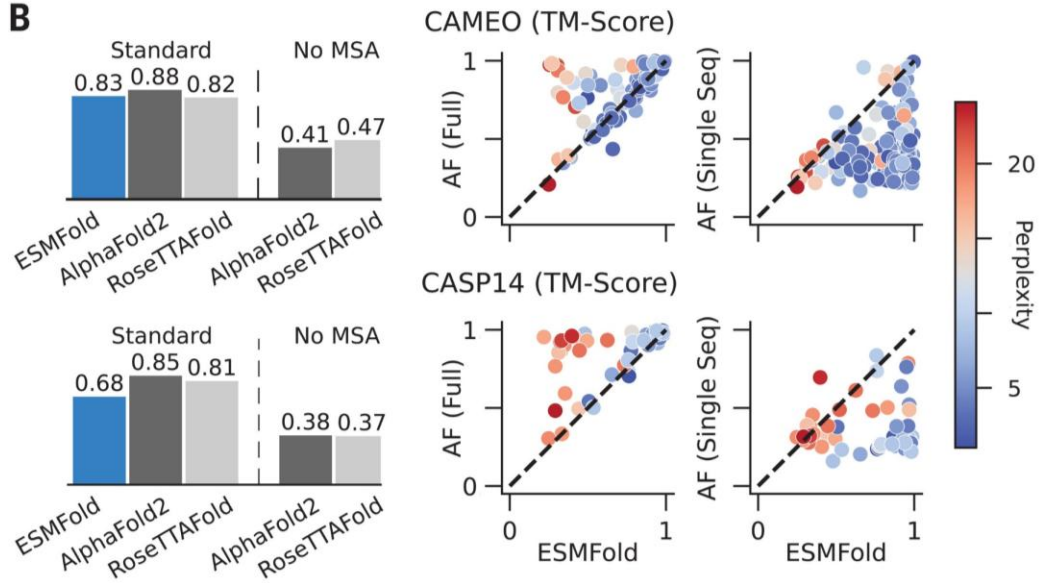
The BERT-style architecture facilitates a **deeper understanding** of representations.

ESMFold enables fast and accurate single sequence prediction with a pre-trained protein language model

Extract evolutionary information

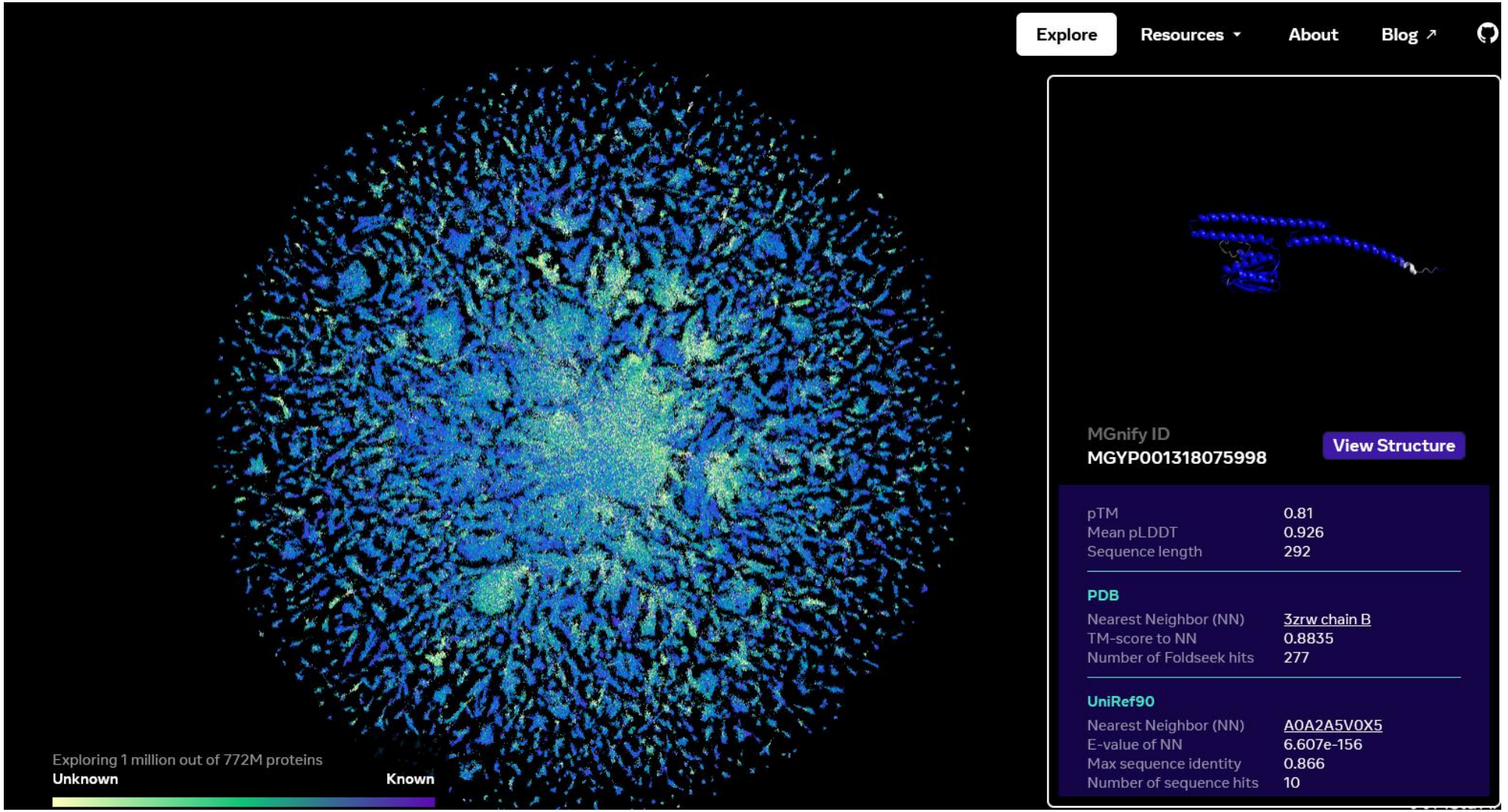




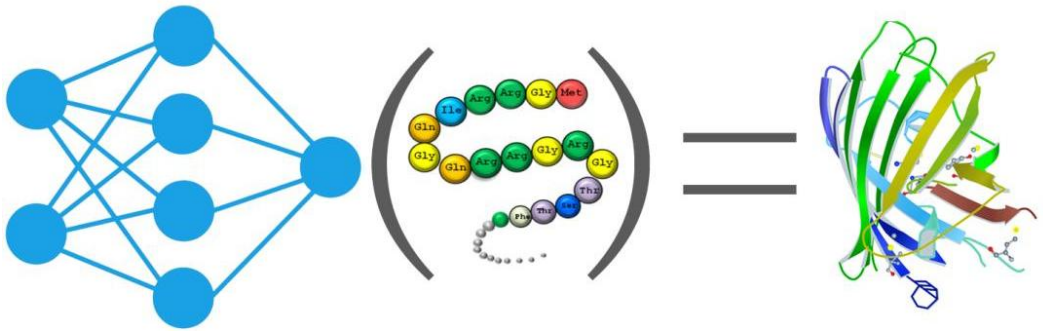


In AlphaFold2: row- and column-wise attention

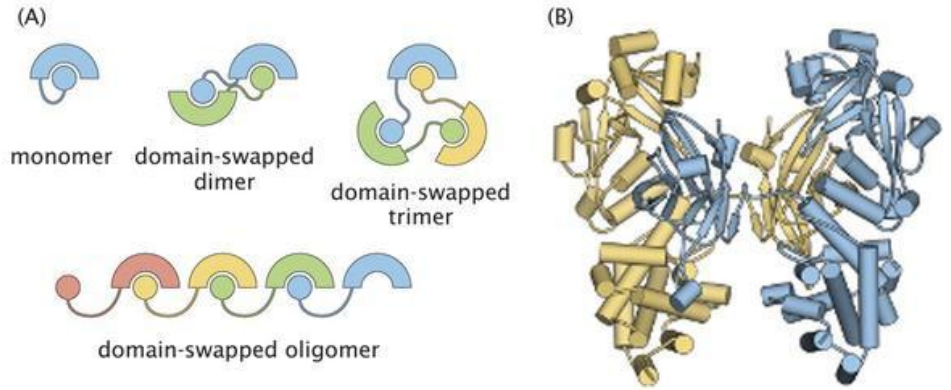
Protein universe is far greater than that of the proteins known in the biological world.



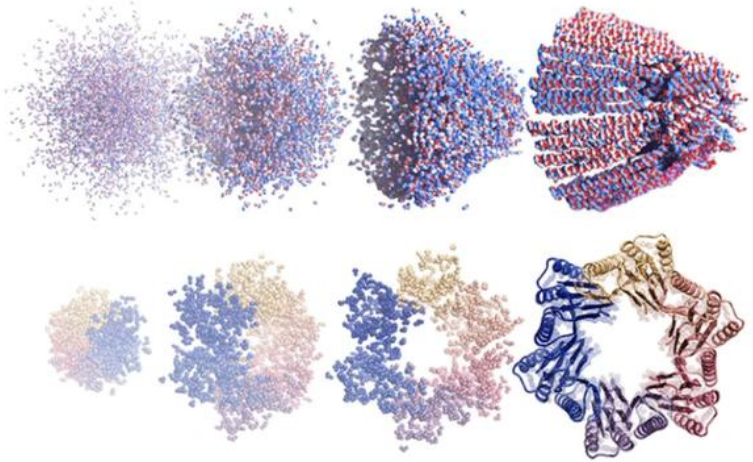
We need a protein foundation model to integrate the sequence, structure, function, and evolution information



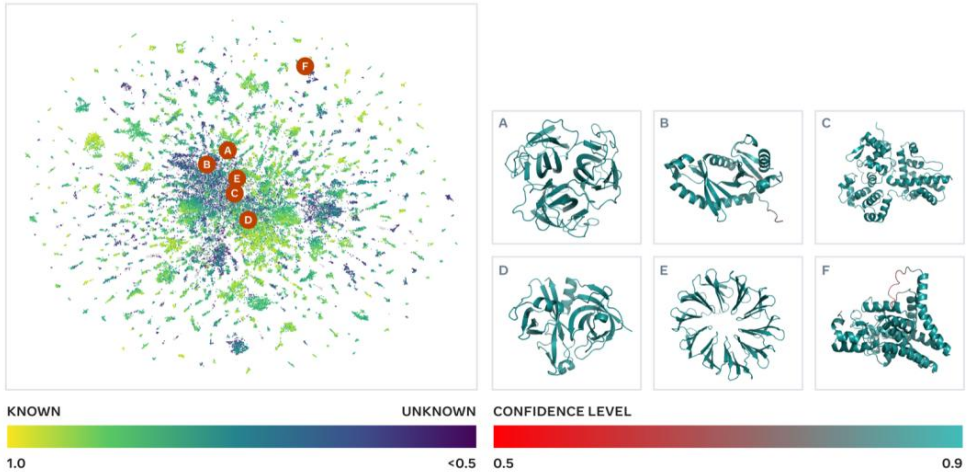
Structure prediction



Function definition

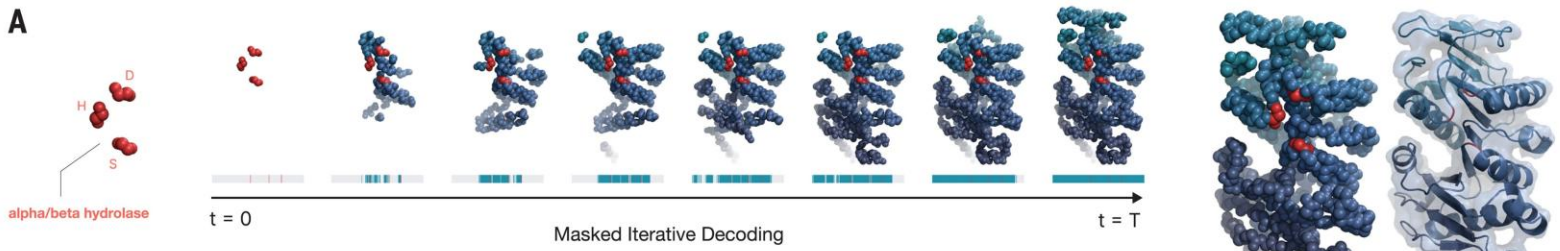


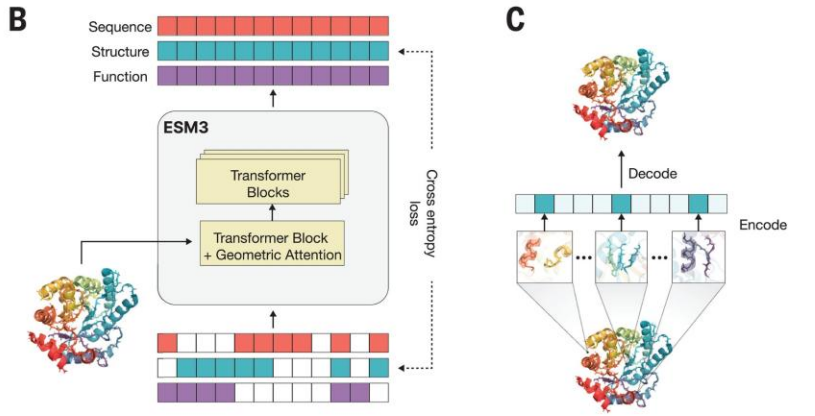
Protein design

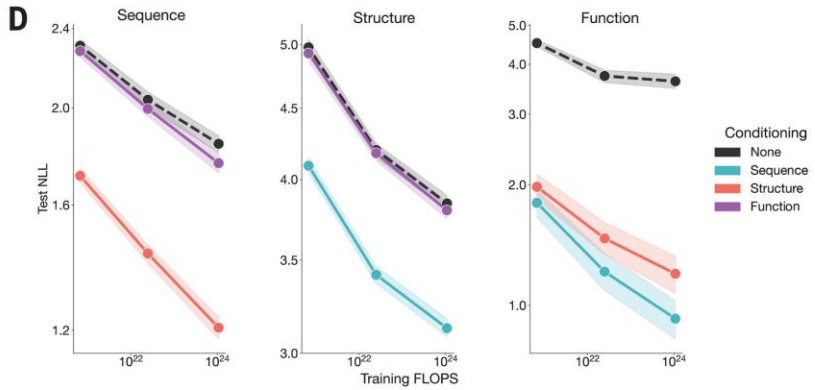


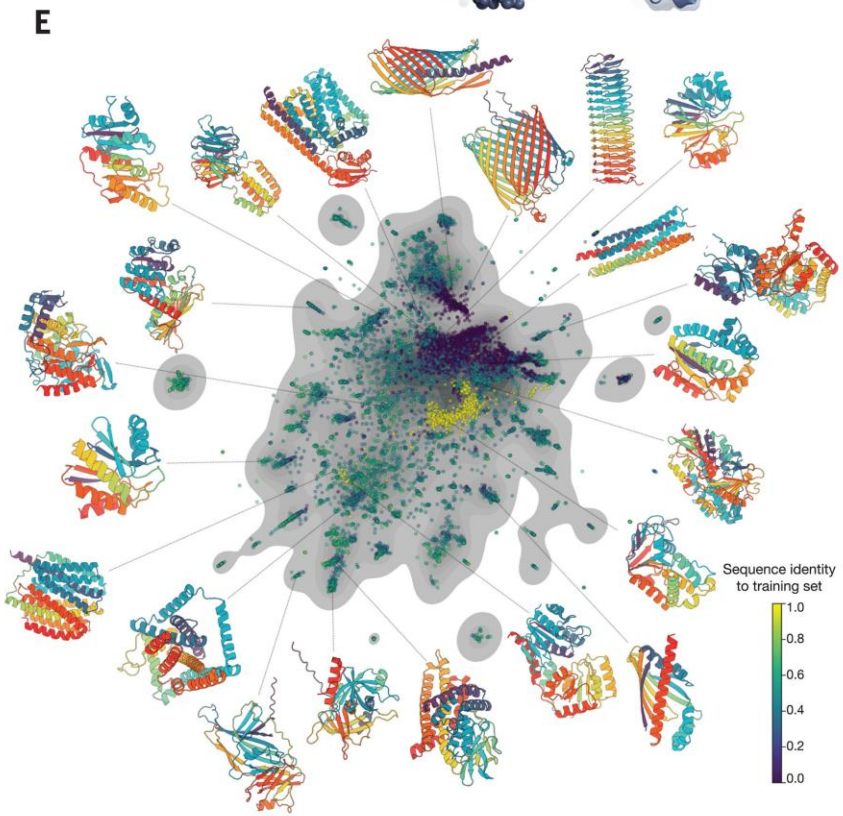
Protein Evolution

ESM3: simulating evolution with a protein LLM





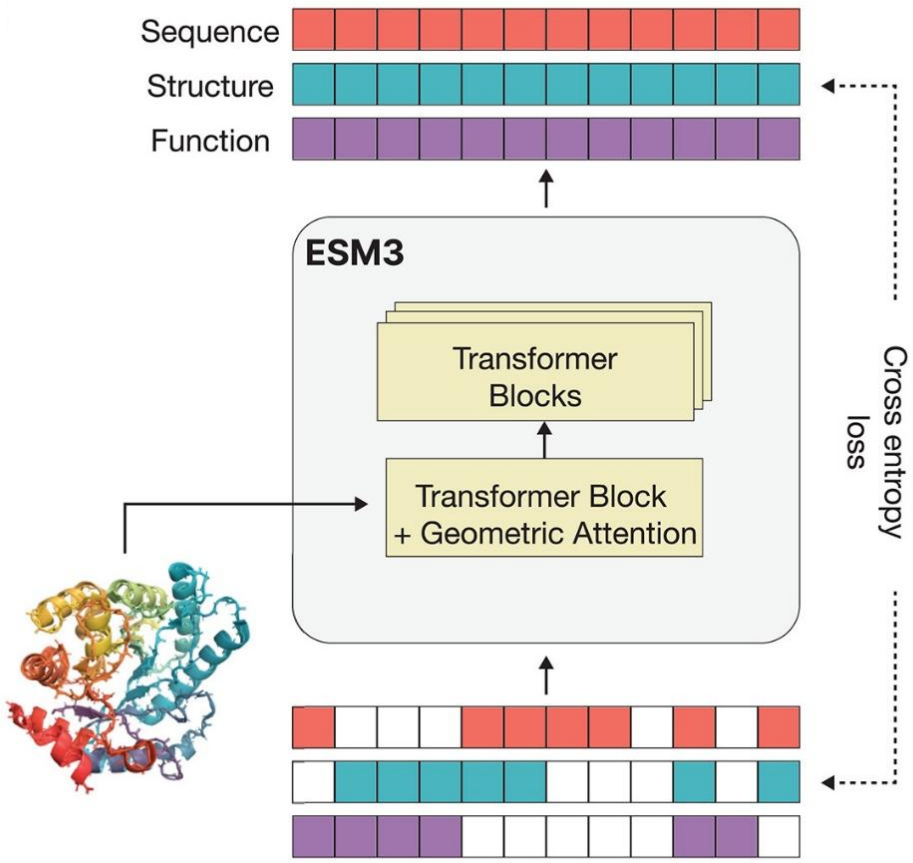


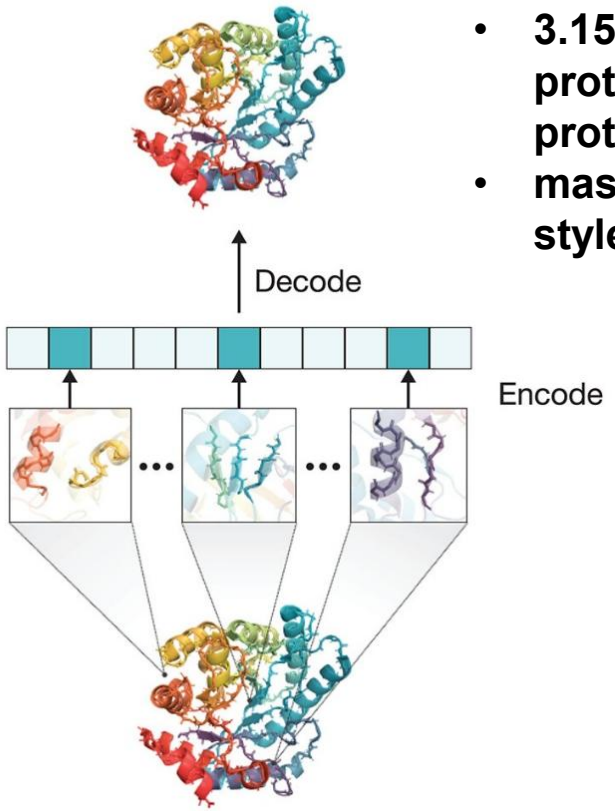


Unconditional generated sequences by ESM3 are diverse, high-quality, and cover the distribution of natural sequences.

(*Science*, 2025)

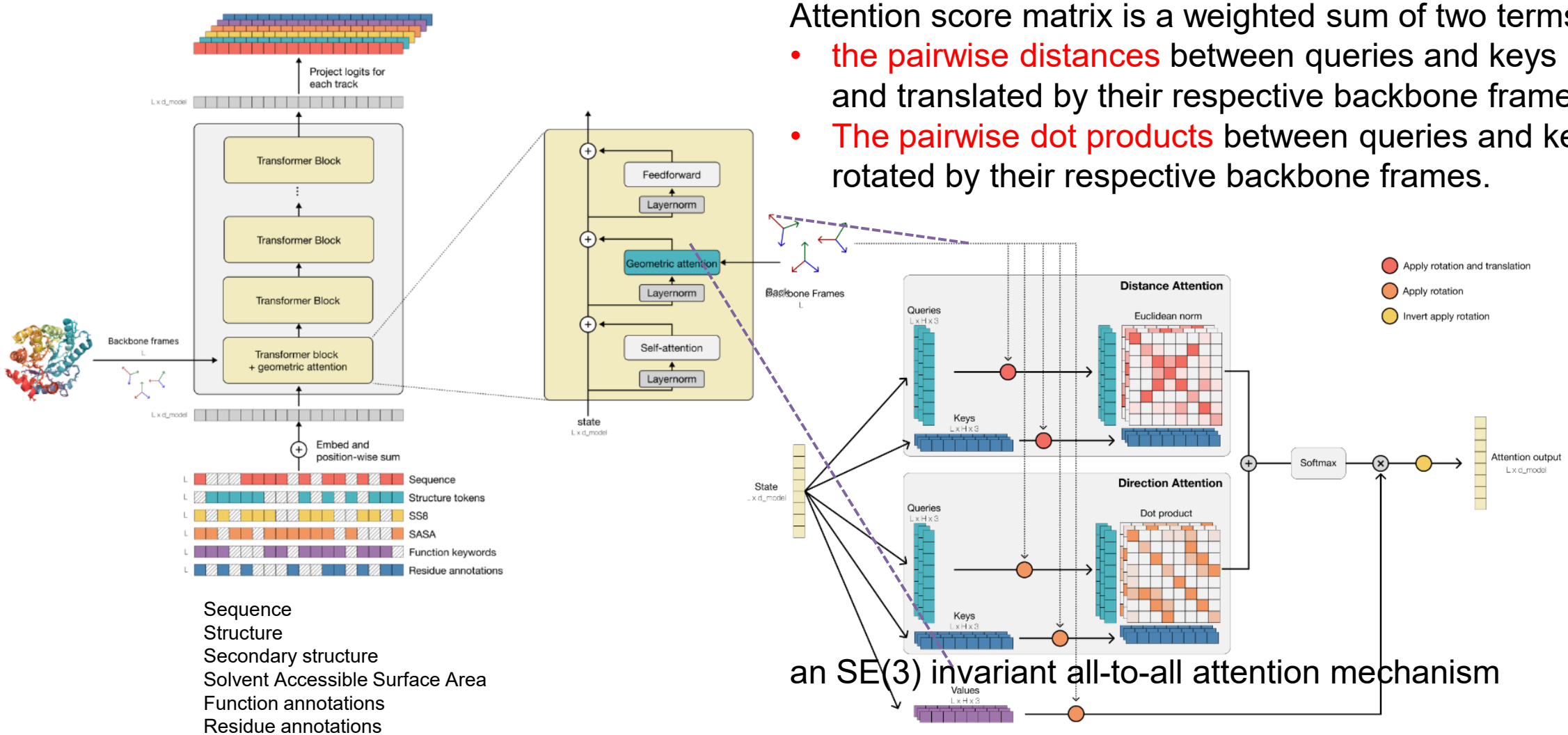
ESM3 architecture





- **98B parameters**
- **3.15B protein sequences, 236M protein structures, and 539M proteins with function**
- **masked language model (BERT-style)**

ESM3 employs a geometric attention mechanism to learn the relationships between protein residues



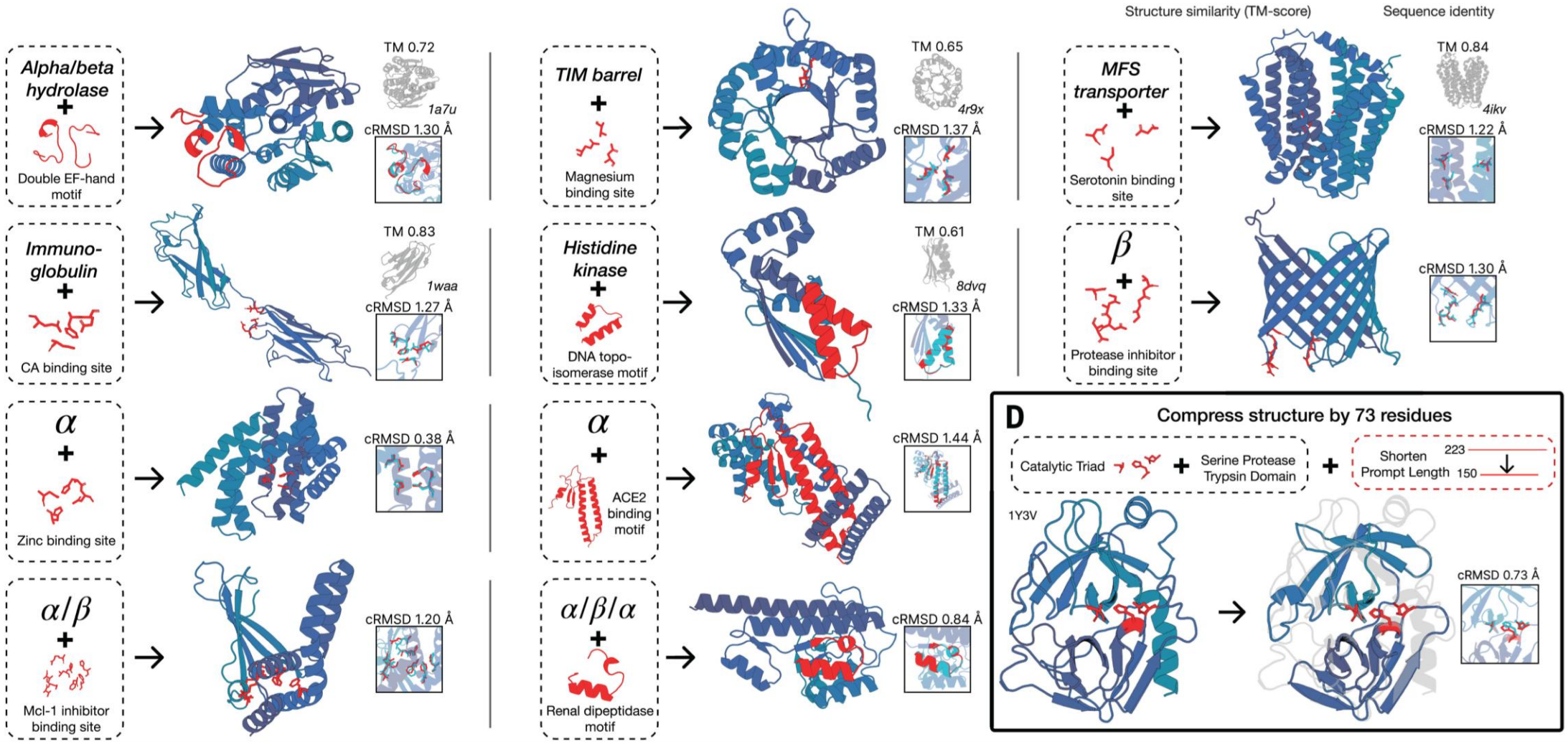
Attention score matrix is a weighted sum of two terms:

- the pairwise distances between queries and keys rotated and translated by their respective backbone frames
- The pairwise dot products between queries and keys rotated by their respective backbone frames.

an SE(3) invariant all-to-all attention mechanism

Sequence
Structure
Secondary structure
Solvent Accessible Surface Area
Function annotations
Residue annotations

ESM3 Designs whole proteins with prompt of physicochemical property or atomic-level motifs

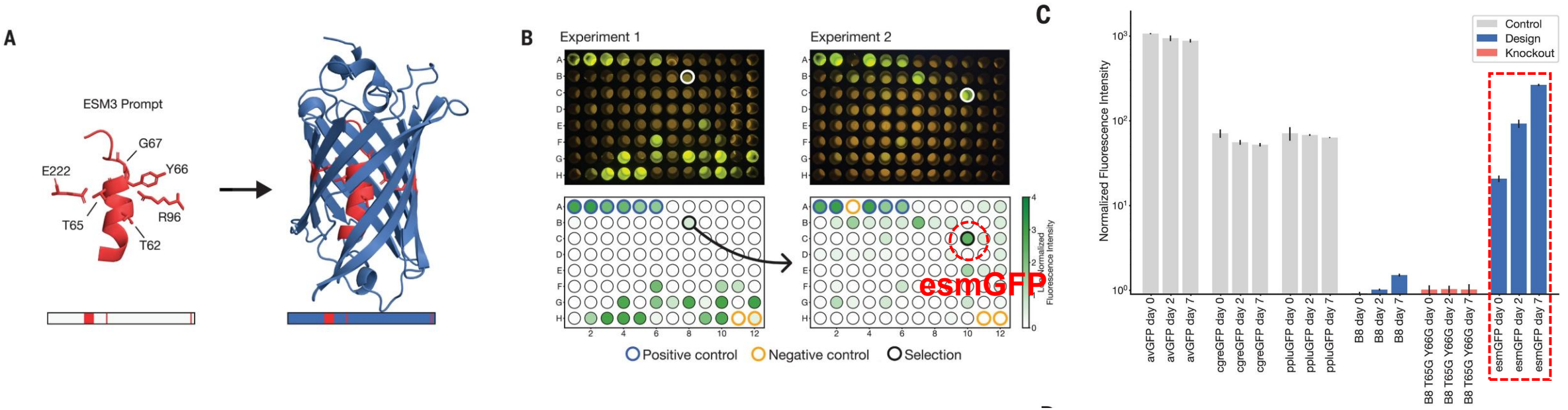


ESM3 Generates a functional fluorescent protein

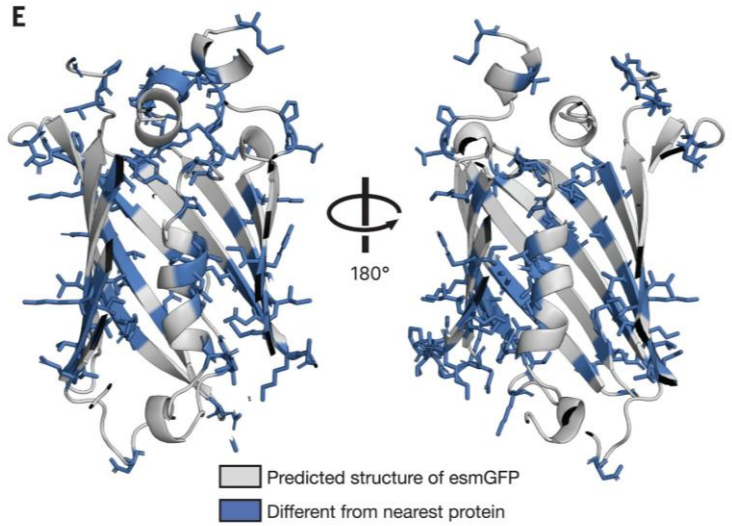
ESM3 generate a fluorescent protein condition on residues required for forming and catalyzing the chromophore reaction

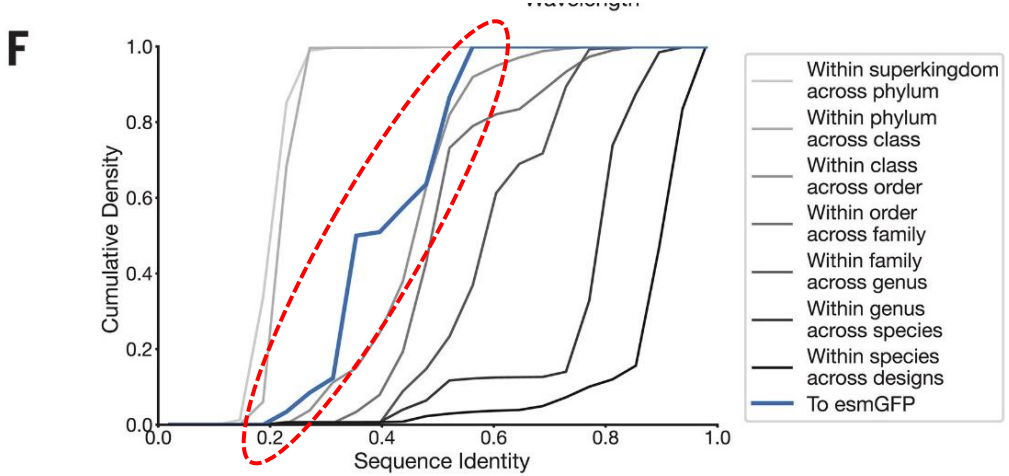
The fluorescent protein selected from 88 designed proteins is functional

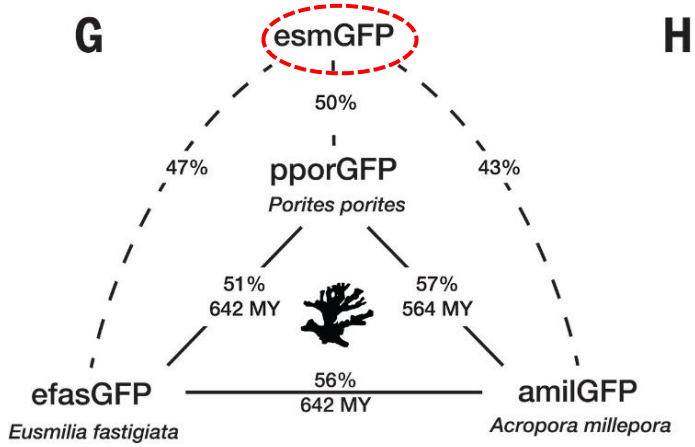
esmGFP exhibits fluorescence intensity similar to common GFPs.

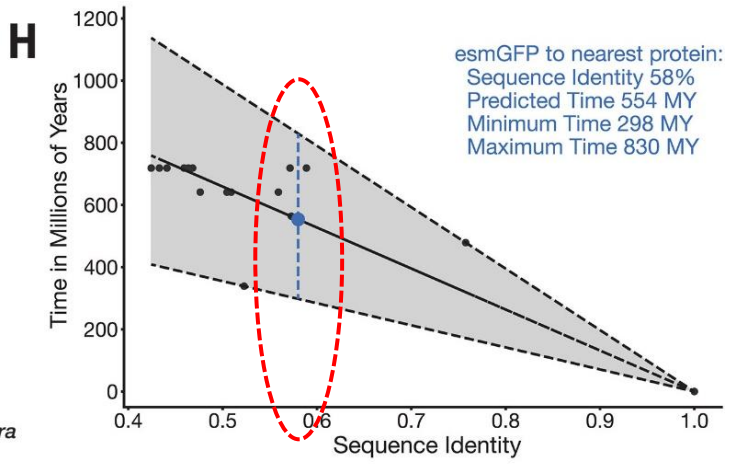


esmGFP has distant similarity with known proteins (57% sequence identity to the nearest existing fluorescent protein)

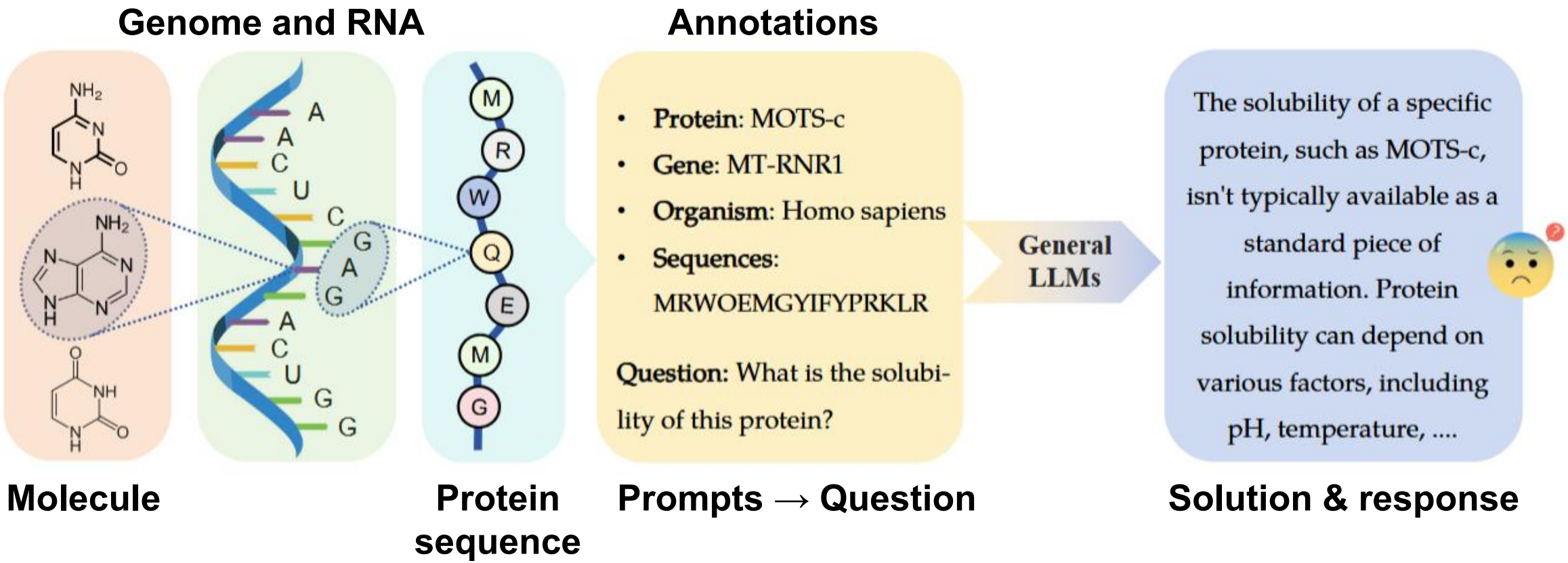




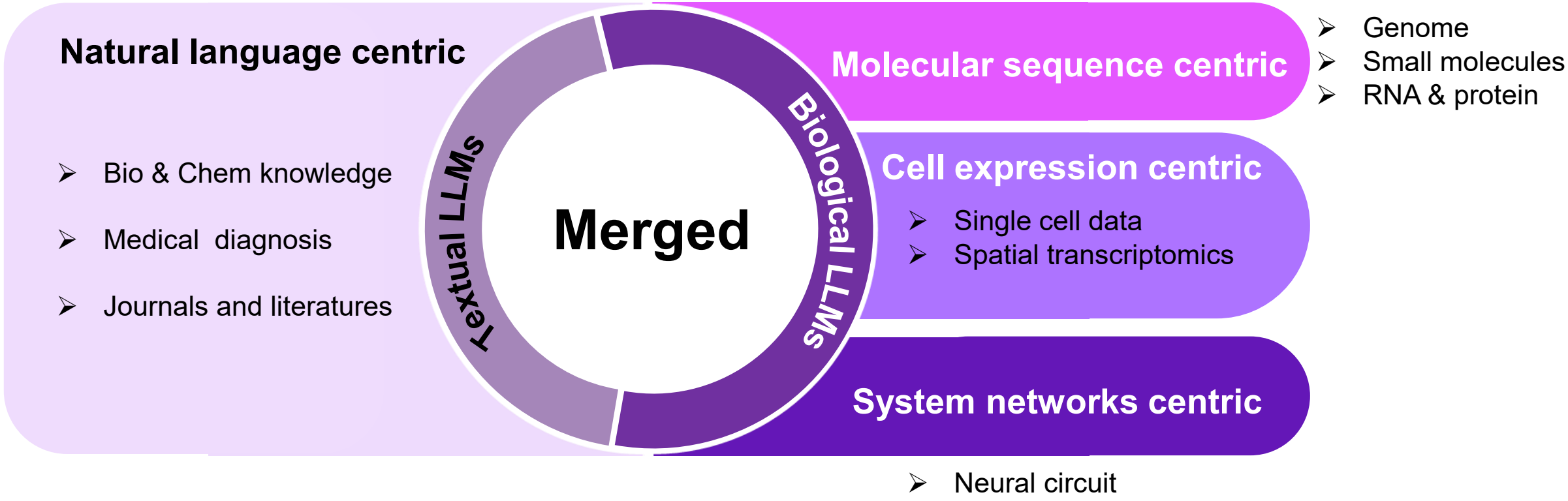




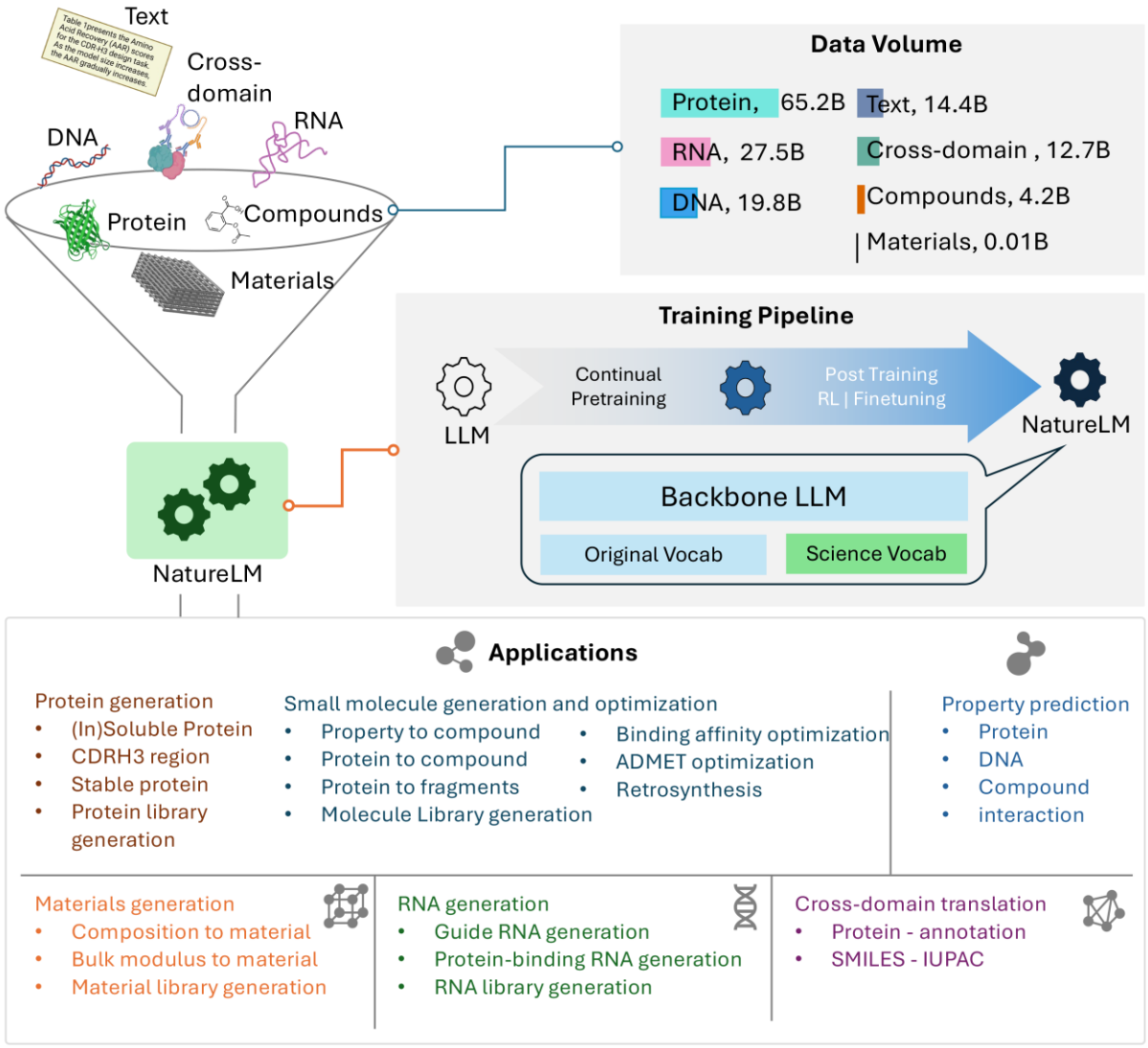
Biological chat foundation model



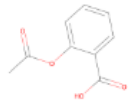
Biological chat foundation model



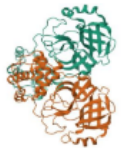
NatureLM: deciphering the Language of Nature for Scientific Discovery



NatureLM: deciphering the Language of Nature for Scientific Discovery



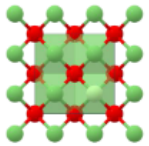
⟨mol⟩ C C (= O) O c 1 c c c c c 1 C (= O) O ⟨/mol⟩



⟨protein⟩ G S G F R K M A F P S G K ... V T F Q ⟨/protein⟩



⟨dna⟩ A T C A A T A T C C A C C T ... T G A T ⟨/dna⟩



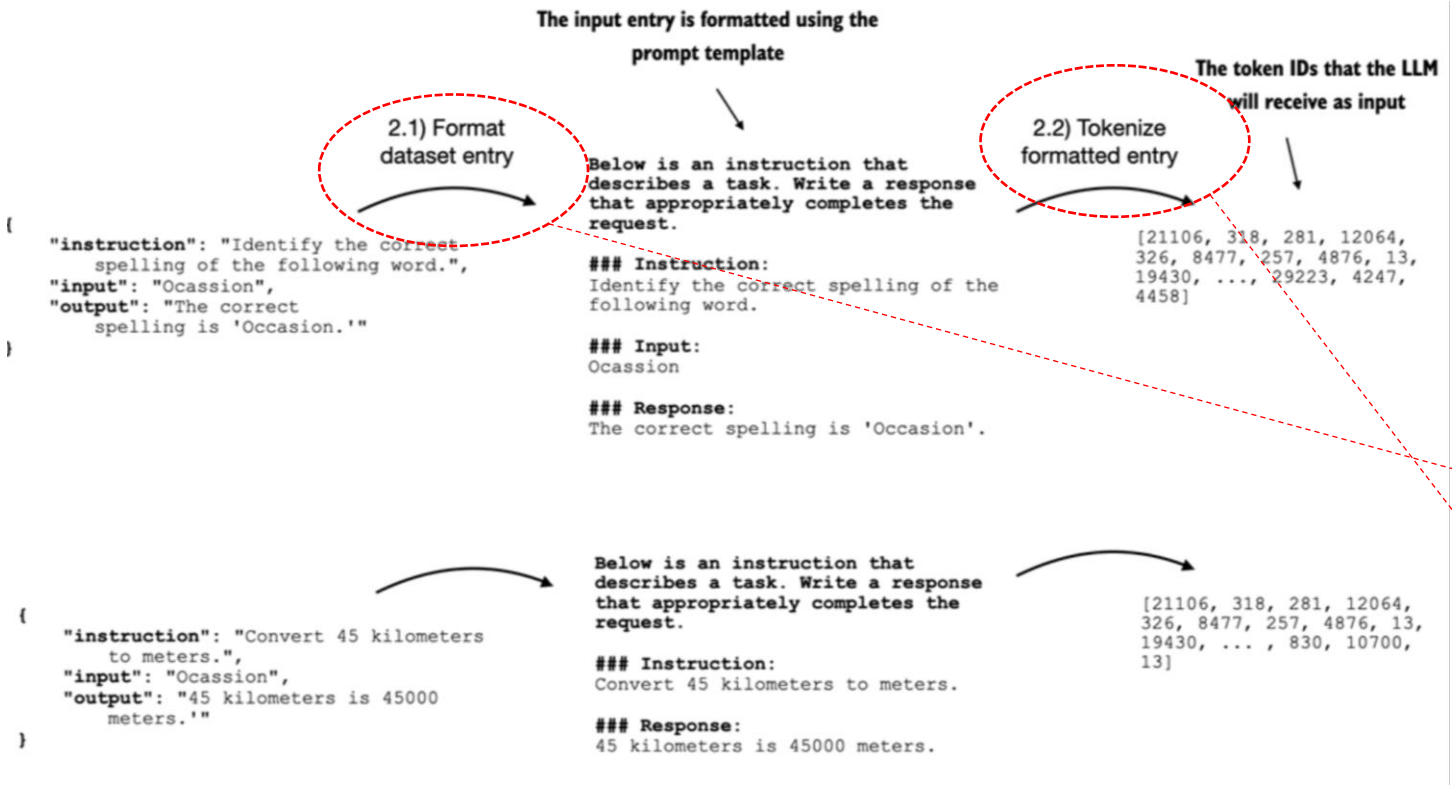
⟨material⟩ Li Li Li Li O O O O O O O O ⟨sg225⟩ ⟨/material⟩

Instruction:
Verify if there is a promoter region within ⟨dna⟩TGGACT···TGAGCTC⟨/dna⟩?
Response:
Yes.

Instruction:
Can the sequence ⟨dna⟩GCCTGCCAG···AAAAC⟨/dna⟩ be classified as a transcription factor binding site?
Response:
No.

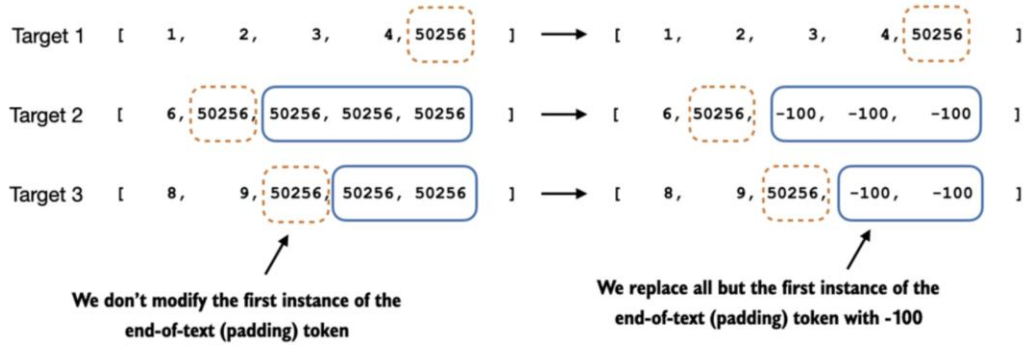
Instruction:
Enhance the effectiveness of the molecule ⟨mol⟩COc1cc2c(c(OC)c1OC)-c1ccc(OC)c(=O)cc1[C@@H](NC(C)=O)CC2⟨/mol⟩ in penetrating the blood-brain barrier.
Accepted Response:
⟨mol⟩COc1cc2c(c(OC)c1OC)-c1ccc(OC)c(=O)cc1C(NC(C)=O)CC2⟨/mol⟩
Reject Response:
⟨mol⟩COc1cc2c(c(OC)c1OC)-c1ccc(OC)c(=O)cc1[C@@H](NC(C)=O)CC2⟨/mol⟩

Instruction finetuning

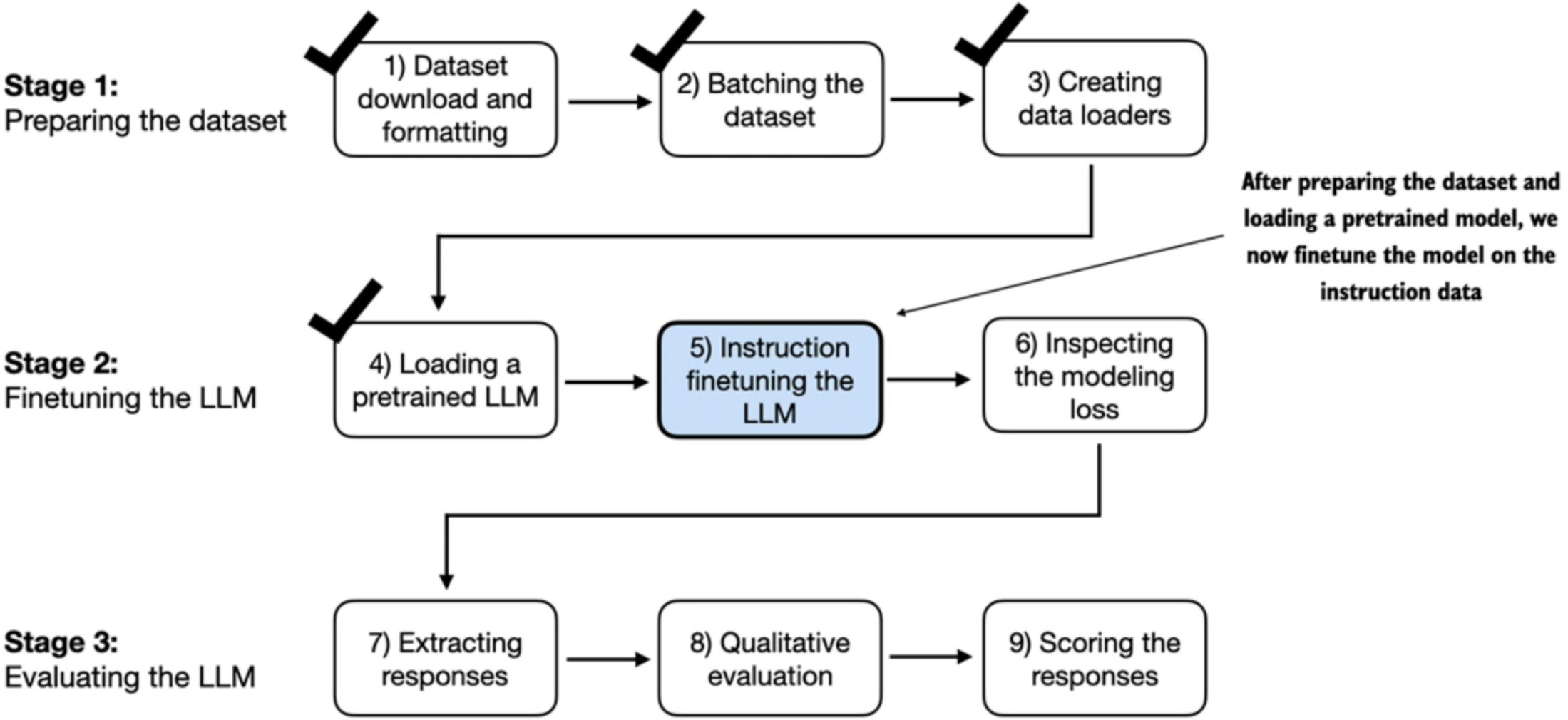



```python
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )
    input_text = f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    return instruction_text + input_text

instruction_plus_input = format_input(entry)
response_text = f"\n\n### Response:\n{entry['output']}"
full_text = instruction_plus_input + response_text
self.encoded_texts.append(
    tokenizer.encode(full_text)
)
```



Instruction finetuning



Q&A