block b4

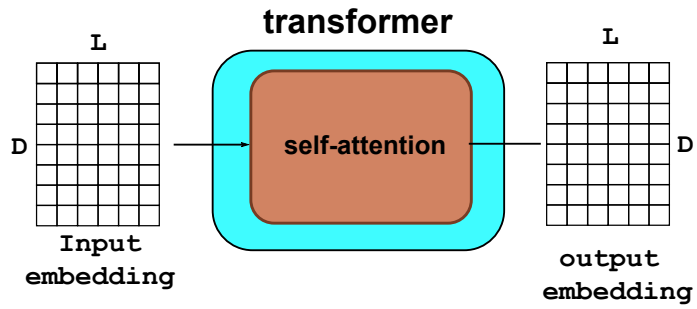
block b4
Large Language Models (LLMs)

block b4

Large Language Models (LLMs)

LLMs for genomics and proteins

self-attention



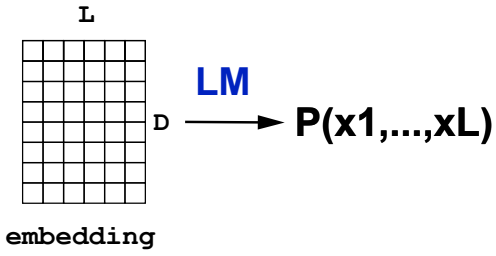
Transformers

Language Models (LLMs)

Probabilistic/Generative Models

[b4] Language Models (LLMs)

LM outputs are probability distributions over sequences P(x1,...,xL)



P(AUG,AUG) = 0.0001

P(AUG,stop-codon) = 0.0000001

P(AUG,$codon_1$,...,$codon_n$ stop-codon) = 0.1

LMs are generative models

can sample new examples (synthetic biology)

LMs usually autoregressive (which is exact, not an approximation)

x1 ~ P(x1)

x2 ~ P(x2 | x1)

x3 ~ P(x3 | x1,x2)

x4 ~ P(x4 | x1,x2,x3)

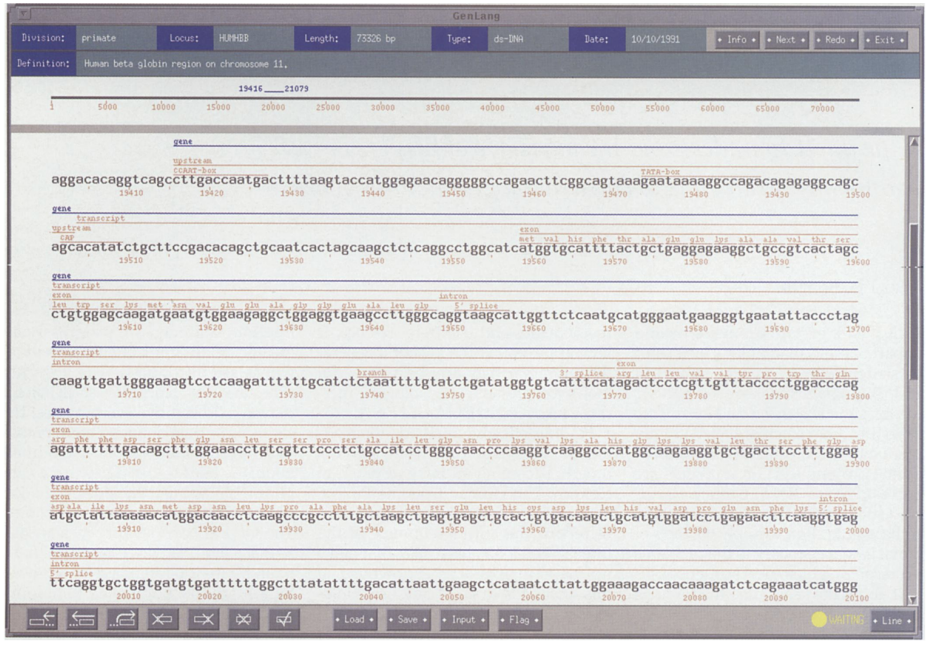
Language Models for biological sequences

The Linguistics of DNA

In the quest to understand the language of life,
a promising strategy is to construct a grammar of genes

David B. Searls (1999)

The structure of a gene revealed by linguistic analysis



Noam Chomsky hierarchy of Natural Languages

Grammar-based Languages

A language of 0's and 1's

This is the **Grammar**

$$\mathbf{S} \rightarrow 0\mathbf{S} \mid 1\mathbf{S} \mid \epsilon$$

These are **"sentences"** the grammar can produce

$$\mathbf{S} \Rightarrow 1\mathbf{S} \Rightarrow 11\mathbf{S} \Rightarrow 110\mathbf{S} \Rightarrow 110\epsilon$$

$$\mathbf{S} \Rightarrow 1\mathbf{S} \Rightarrow 10\mathbf{S} \Rightarrow 100\mathbf{S} \Rightarrow 1000\mathbf{S} \Rightarrow \ldots$$

Grammer-based Biological Languages

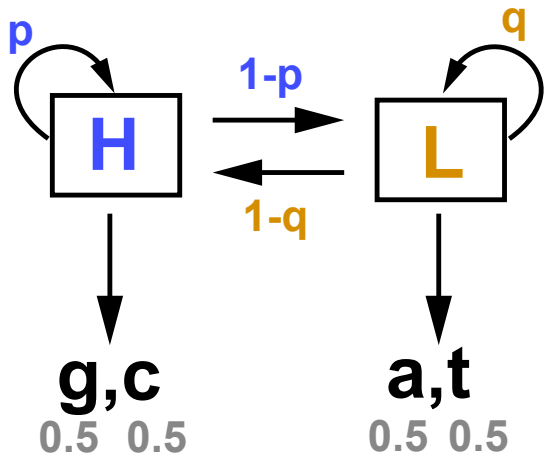
Hidden Markov Models (regular grammars)

To identify GC righ regions in a genome

S -> H | L

H -> {g,c} H | {g,c} L | ϵ

L -> {a,t} L | {a,t} H | ϵ



S => gH=> ggH=> ggcH=> ggcgH

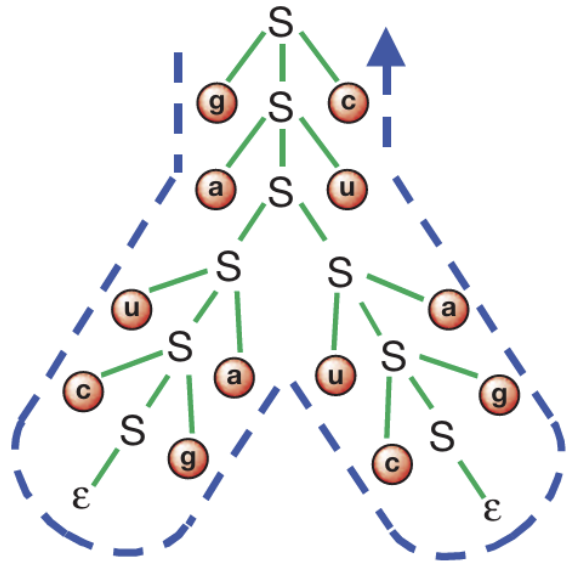
=> ggcgaL=>ggcgaaL=>ggcgaagH=>ggcgaagϵ

Grammer-based Biological Languages

SCFGs for RNA structure

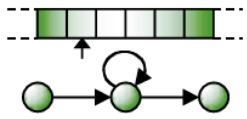
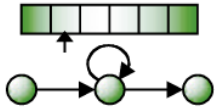
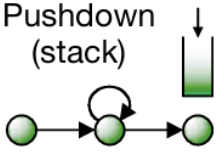
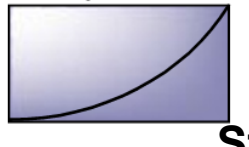
S -> a S b | a S | S a | SS | ϵ

S => g S c
=> g a S u c
=> g a S S u c
=> g a u S a S u c
=> g a u c S g a S u c
=> g a u c ϵ g a S u c
=> g a u c ϵ g a u S a u c
=> g a u c ϵ g a u c S ga u c
=> g a u c ϵ g a u c ϵ ga u c



Grammer-based Biological Languages

Hierarchy of Grammar-based Languages

Language	Automaton	Grammar	Recognition
Recursively enumerable languages	Turing machine 	Unrestricted $Baa \rightarrow A$	Undecidable 
Context-sensitive languages	Linear bounded 	Context sensitive $At \rightarrow aA$	Exponential? 
Context-free languages	Pushdown (stack) 	Context free $S \rightarrow gSc$	Polynomial 
Regular languages	Finite-state automaton 	Regular $A \rightarrow cA$	Linear 

extensively used for RNA structure

Stochastic Context-Free Grammars

extensively used for protein/DNA homolgy

Hidden Markov Models

Language Models = Probabilistic Models

P_{data} that the LM tries to approximate
by a probability distribution P_θ

The LM gives a parametric representation of the data

Generative models

The language or messenger RNA sequences that code for proteins

data = { many mRNA sequences }

start-codon codon1 codon2 .... codonL stop-codon
AUG NNN NNN ... NNN UAA

The LM produces a probability distribution
P_θ(RNA sequences)
such that you may expect...

P_θ(AUG NNN NNN ... NNN UAA) $= 0.01$

P_θ(UG NNN NNN ... NNN UAA) $= 0.0000001$ a deletion in the start codon AU

P_θ(AUG NNN NNN ... NNN UUA) $= 0.00001$ a mutation in the stop codon UAA-

P_θ(AUG NN NNN ... N UAA) $= 0.001$ a frameshift

Generative models

Learning

find $\theta^\star$

so that

$$P_{\theta^\star} \approx P_{data}.$$

Generative models

Learning Objectives

$$P_{\theta^\star} \approx P_{data}$$

- What data to use as representative for the language we are trying to model?
  need large dataset, overfitting
- What objective function to use to compare the data and the parametric probability distributions?
  The loss

Generative models

Inference

Given

$$P_{\theta^\star} \approx P_{data}$$

Now for a new data example x

$$P_{\theta^\star}(x)$$

gives us the probability that x belongs to the language

Generative models

Sampling

Given

$$P_{\theta^\star} \approx P_{data}$$

We can use

$$P_{\theta^\star}$$

to generate **novel data** by sampling from the model distribution

$$x_{new} \sim P_{\theta^\star}$$

Generative models

Unsupervised learning

Because we are estimating the whole probability of the totality of the data x

$$P_{\theta^\star}(x)$$

May be able to learn features of the data
(not used in training)

For instance,
monocistronic/polycistronic mRNA
the genetic code and variants

Generative vs Discriminative Models

Discriminative models: you give me one data point x,
I will give you info about its labels.
$P(label|x)$

Generative models: information about all data points and their labels
$P(x, label)$

sometimes just the marginal
$P(x) = \sum_{label} P(x, label)$

Generative (probabilistic) Models

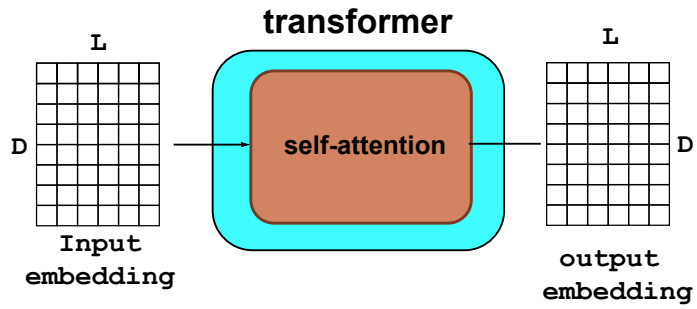
Grammar-based

1. Hidden Markov Model
2. Stochastic Context-Free Grammars

Deep Learning

1. Transformer-based autoregressive language models (block 4)
2. Variational Autoencoders (block 5)
3. Denoising Diffusion Models (block 6)

self-attention

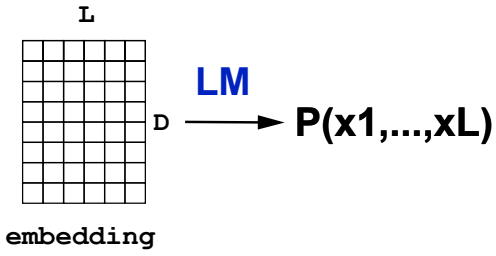


Transformers

Language Models (LLMs)

[b4] Language Models (LLMs)

LM outputs are probability distributions over sequences P(x1,...,xL)



P(AUG,AUG) = 0.0001

P(AUG,stop-codon) = 0.0000001

P(AUG,$codon_1$,...,$codon_n$ stop-codon) = 0.1

LMs are generative models

can sample new examples (synthetic biology)

LMs usually autoregressive (which is exact, not an approximation)

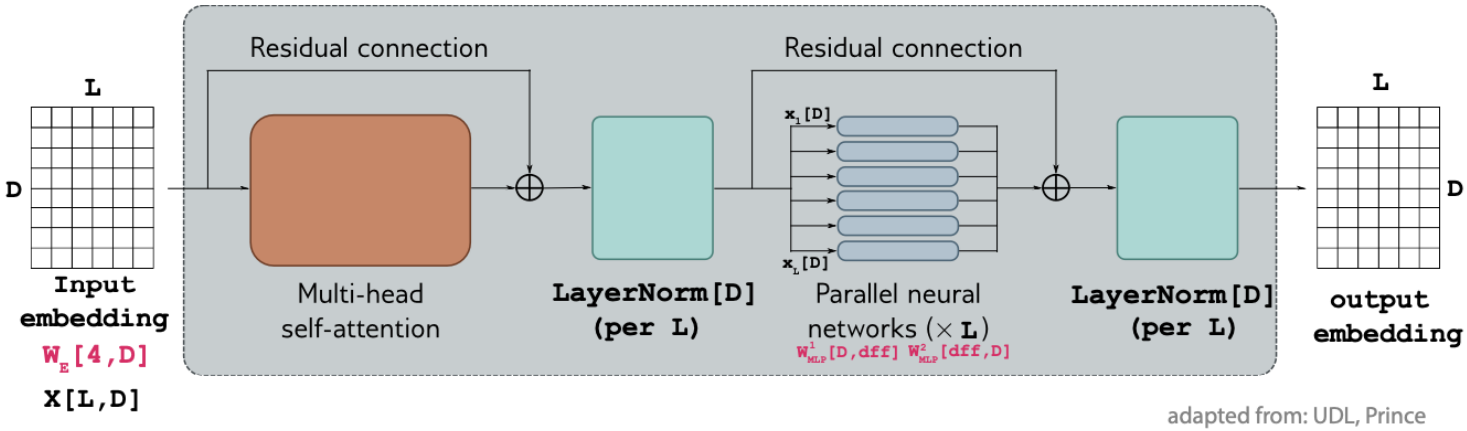
x1 ~ P(x1)

x2 ~ P(x2 | x1)

x3 ~ P(x3 | x1,x2)

x4 ~ P(x4 | x1,x2,x3)

Transformer Block



adapted from: UDL, Prince

```python
# simple Transformer block = attention -> MLP
class SimpleTransformerBlock(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()

        self.attn = SelfAttention(d_model)

        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model),
        )

        self.norm_att = nn.LayerNorm(d_model)
        self.norm_ff  = nn.LayerNorm(d_model)

    def forward(self, x):
        x = x + self.attn(x)
        x = x + self.mlp(self.norm_att(x))
        x = self.norm_ff(x)
        return x
```

Three types of Transformer-based LLMs

- **Decoders**
  predict next token in a sentence.
  GPT3
- **Encoders**
  transform an input into an output.
  BERT
- **Encoder-Decoders**
  sequence-to-sequence tasks.
  “attention is all you need”

Three types of Transformer-based LLMs

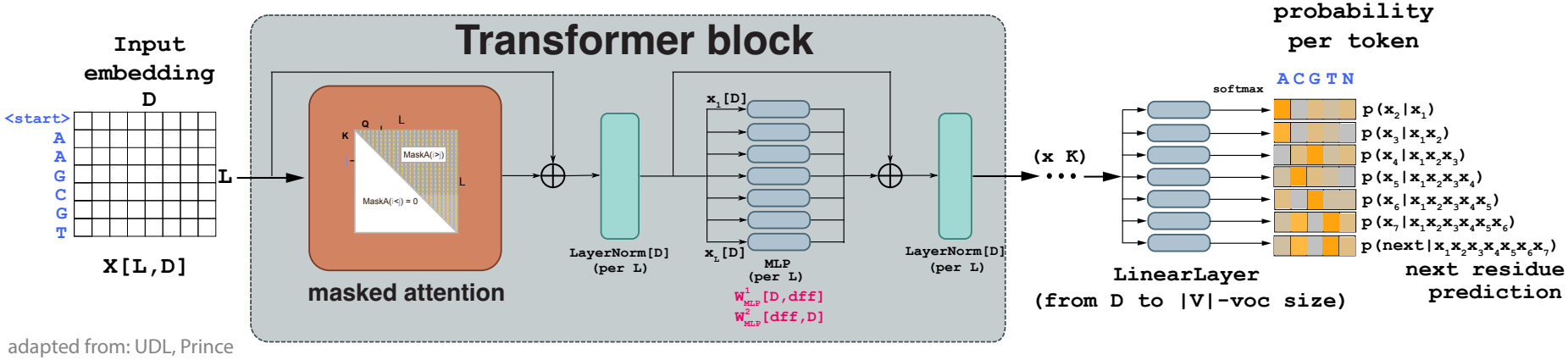
- **Decoders**
  predict next token in a sentence.
  are sentence generators.
  GPT3
- **Encoders**
  transform an input into an output.
  are text classifiers (labeled tasks).
  BERT
- **Encoder-Decoders**
  sequence-to-sequence tasks.
  text translators.
  "attention is all you need"

Decoders **are autoregressive**

Instead of calculating the joint $p(x_1 ... x_L)$
it calculates the next token
conditional on the previous one

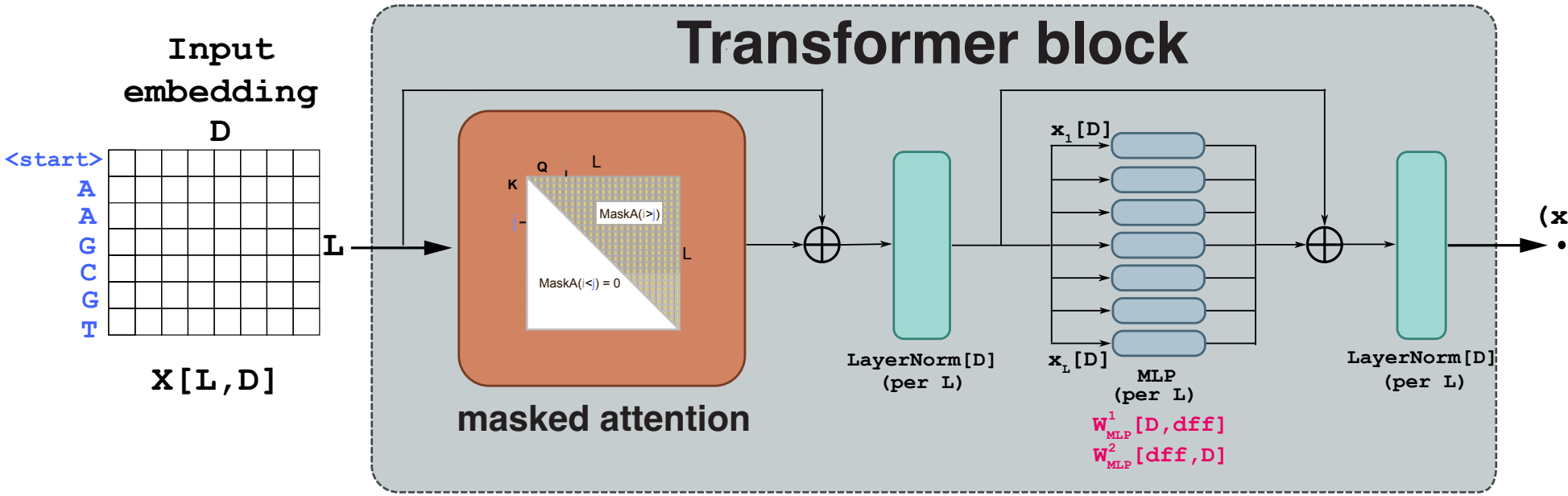
$$p(x_2 \mid x_1)$$
$$p(x_3 \mid x_1 x_2)$$
$$p(x_4 \mid x_1 x_2 x_3)$$
$$\ldots$$
$$p(x_i \mid x_1 x_2 x_3 ... x_{i-1})$$

Decoders are autoregressive



adapted from: UDL, Prince

Decoder use self-masked attention



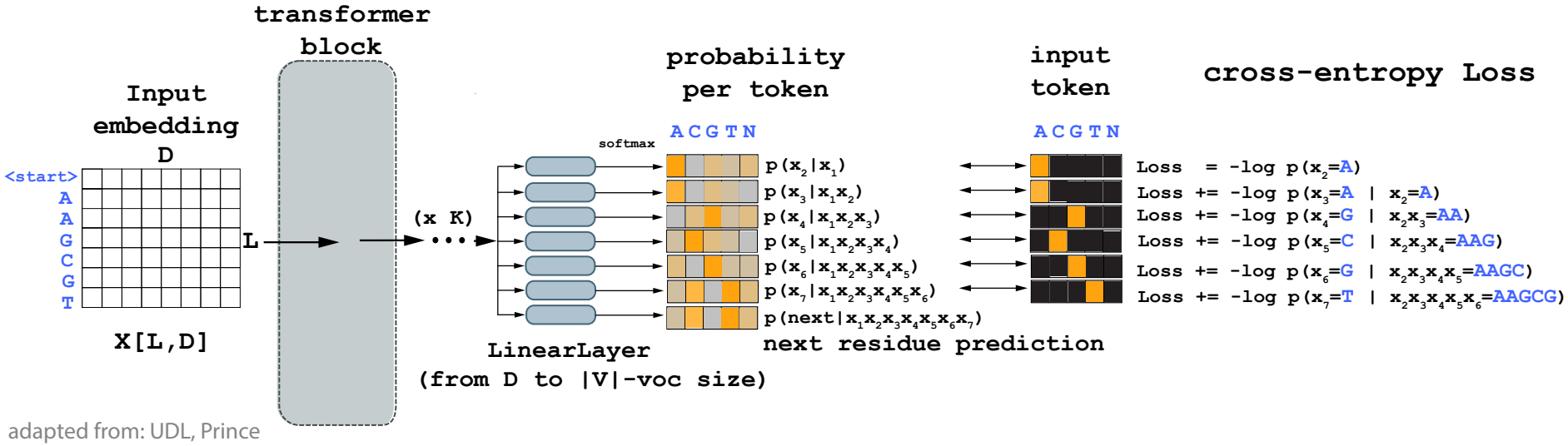
MaskAtt (i<j) = 0

MaskAtt (i>j) = softmax(score($q_i \cdot k_j$))

adapted from: UDL, Prince

decoder loss

Decoder uses cross-entropy loss



adapted from: UDL, Prince

Find parameters that maximize the probability of the data

$$max \prod_i P_\theta(x_i \mid x_1 ... x_{i-1})$$

$$= max \left(\sum_i \log P_\theta(x_i \mid x_1 ... x_{i-1}) \right)$$

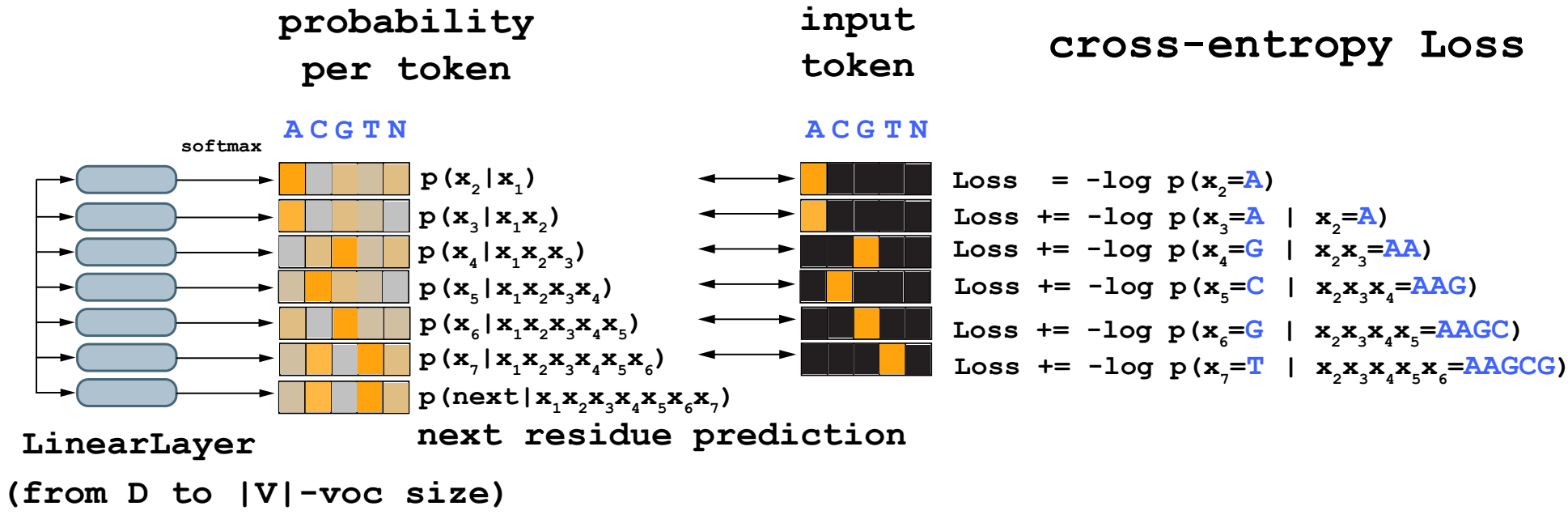
$$= min \left(- \sum_i \log P_\theta(x_i \mid x_1 ... x_{i-1}) \right)$$

Loss to minimize is then

$$Loss = - \sum_i \log P_\theta(x_i \mid x_1 ... x_{i-1})$$

decoder loss

Decoder uses cross-entropy loss



adapted from: UDL, Prince

Decoder loss

Find parameters that maximize the probability of the data

$$Loss = -\sum_i \log P_\theta(x_i \mid x_1 ... x_{i-1})$$

Is a reformulation of cross-entropy

For the one-hot distribution of the true tokens

$$q(a_i \mid a_1 ... a_{i-1}) = \delta(a_1 = x_1) \dots \delta(a_i = x_i)$$

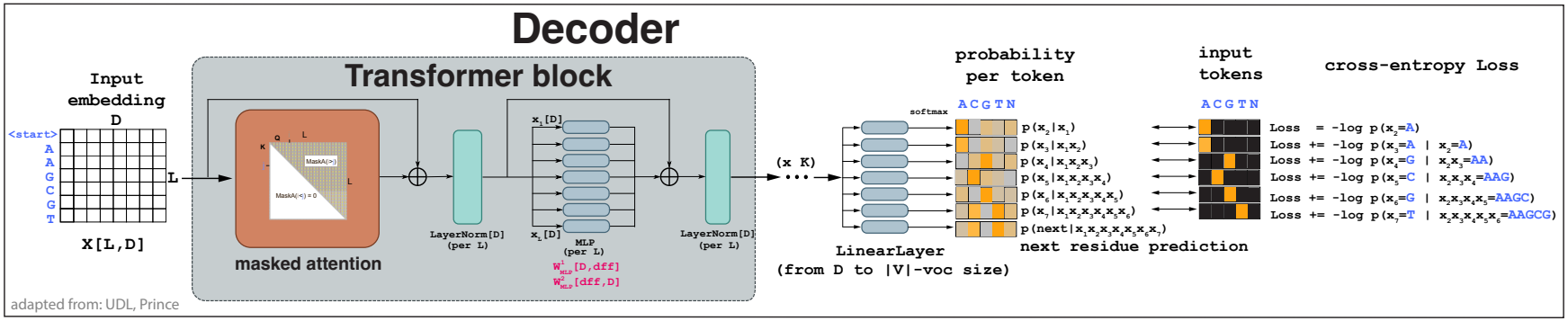
for $a_i \in V$.

Cross-entropy loss

$$\begin{aligned} CELoss &= -\sum_i \sum_{a_1 : a_I} q(a_i \mid a_1 ... a_{i-1}) \log P_\theta(a_i \mid a_1 ... a_{i-1}) \\ &= -\sum_i \log P_\theta(x_i \mid x_1 ... x_{i-1}) \end{aligned}$$

GPT3 is a LLM that apply the decoder mechanism on a massive scale.

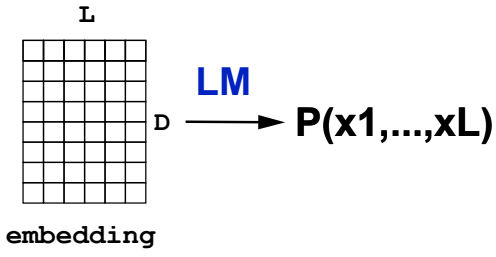
vocabulary	V	50,257 tokens
embedding	D	12,288
Transformer layers	K	96
Transformer heads per layer	h	96
query, key, value dimension per head	D_h	128
FF hidden dimension	$d_f f$	49,152
max length of input	Lmax	2,048 tokens
training	words	300 billion tokens



1. The autoregressive task uses all the data
2. The autoregressive task uses only context to the left
3. Many techniques to improve samples: don't just take the token with highest prob.

Language Models (LLMs)

LM outputs are probability distributions over sequences P(x1,...,xL)



P(AUG,AUG) = 0.0001

P(AUG,stop-codon) = 0.0000001

P(AUG,codon$_1$,...,codon$_n$ stop-codon) = 0.1

LMs are generative models

can sample new examples (synthetic biology)

LMs usually autoregressive (which is exact, not an approximation)

x1 ~ P(x1)

x2 ~ P(x2 | x1)

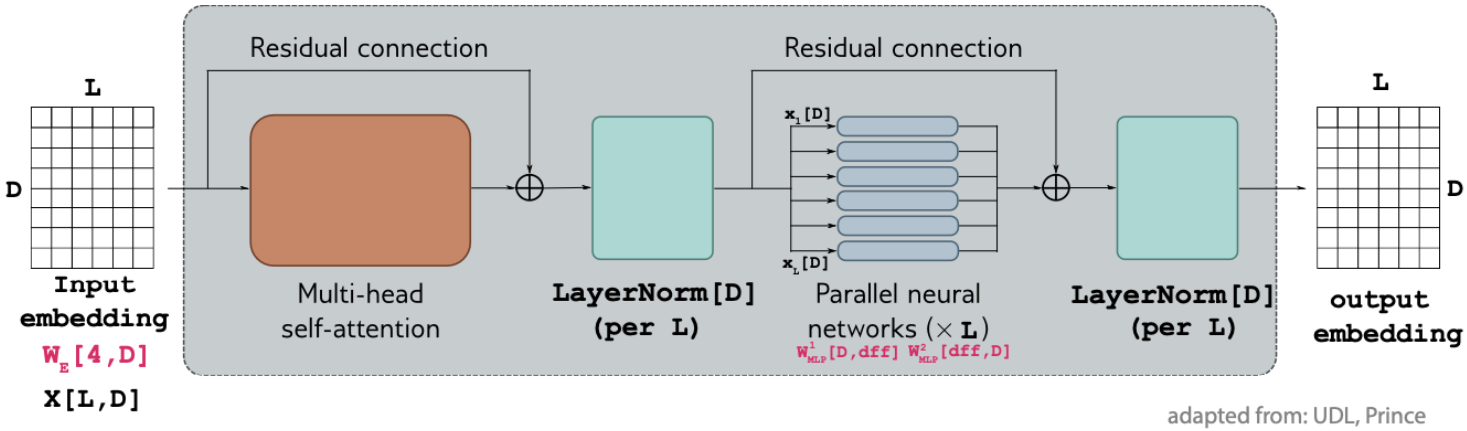
x3 ~ P(x3 | x1,x2)

x4 ~ P(x4 | x1,x2,x3)

P(x1,x2,x3,x4) = P(x1) P(x2 | x1) P(x3 | x1,x2) P(x4 | x1,x2,x3)

LMs use unsupervised learning (no labels)

Transformer Block



adapted from: UDL, Prince

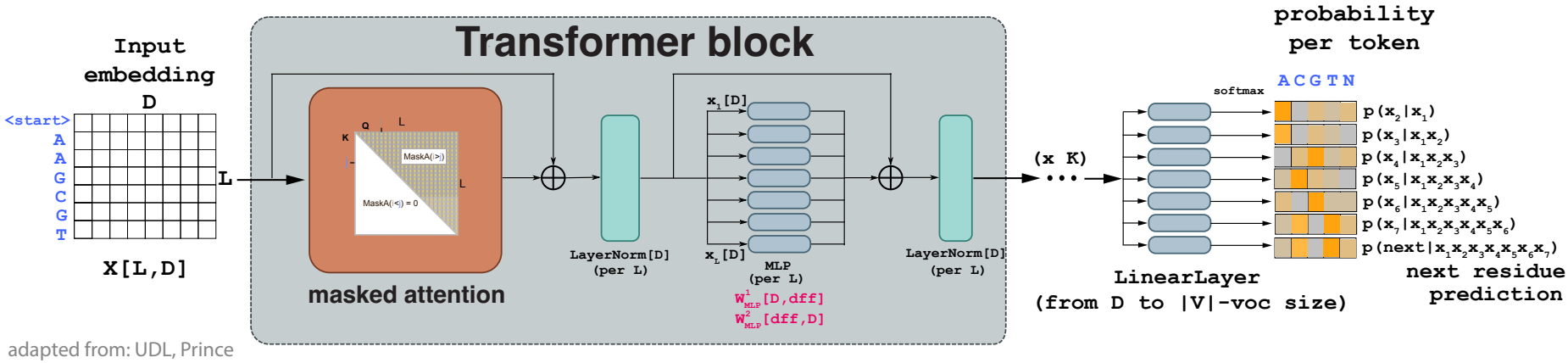
```python
# simple Transformer block = attention -> MLP
class SimpleTransformerBlock(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()

        self.attn = SelfAttention(d_model)

        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model),
        )

        self.norm_att = nn.LayerNorm(d_model)
        self.norm_ff  = nn.LayerNorm(d_model)

    def forward(self, x):
        x = x + self.attn(x)
        x = x + self.mlp(self.norm_att(x))
        x = self.norm_ff(x)
        return x
```

Three types of Transformer-based LLMs

- **Decoders**
  predict next token in a sentence
  sentence generators
  GPT3
- **Encoders**
  transform an input embedding into an output embedding
  classifiers for downstream labeled tasks
  BERT
- **Encoder-Decoders**
  sequence-to-sequence tasks
  text translators
  "attention is all you need"

Transformer Decoder

Decoders are autoregressive

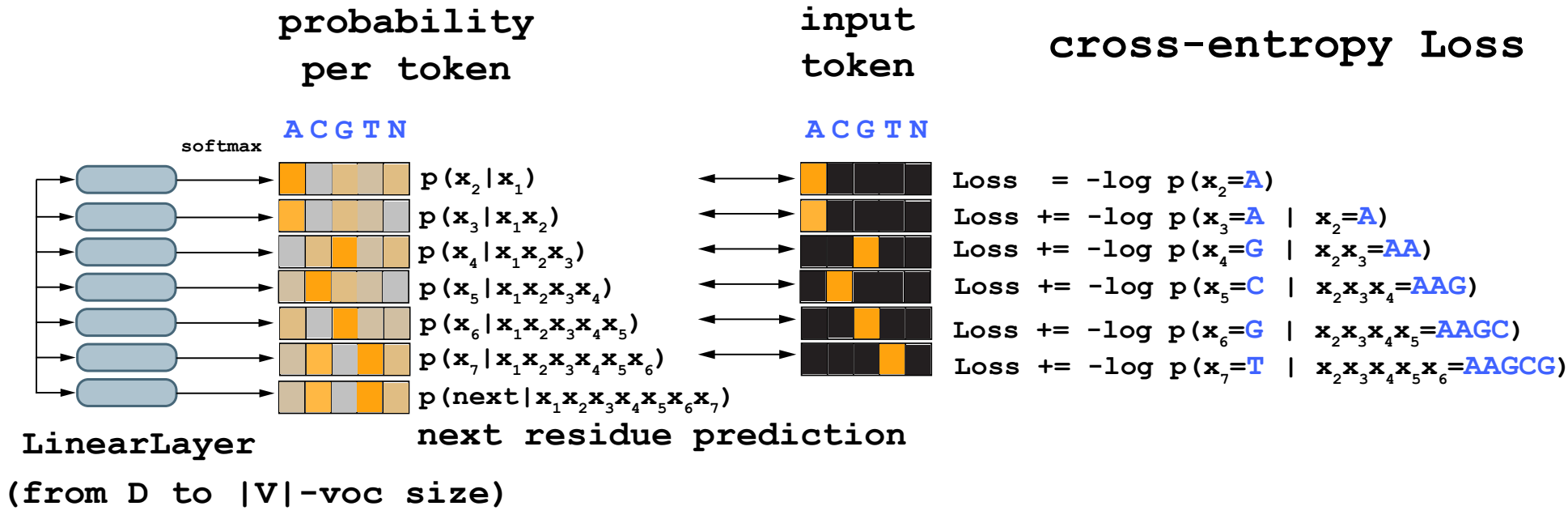


adapted from: UDL, Prince

Code

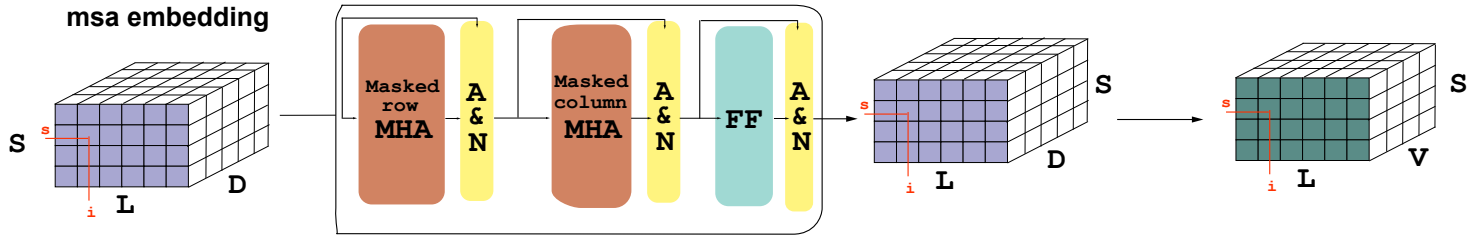
decoder loss

Decoder uses cross-entropy loss

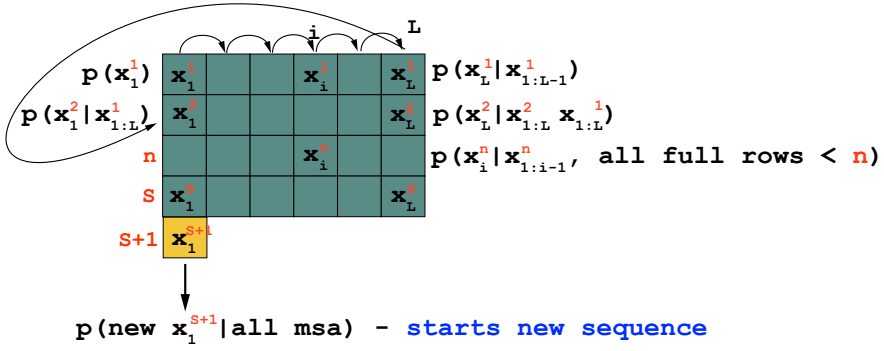


adapted from: UDL, Prince

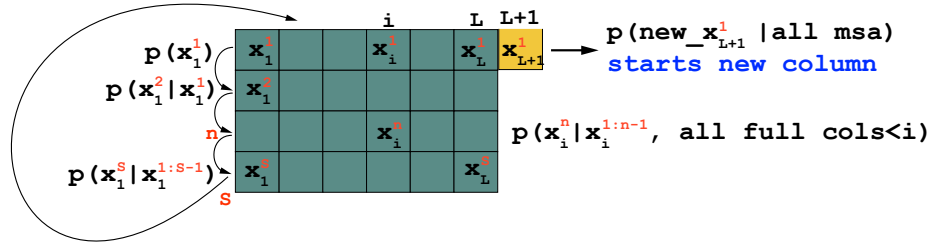
MSA Decoder



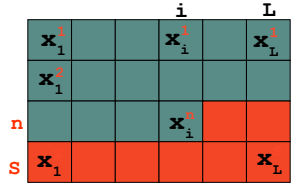
Autoregression by Rows



Autoregression by Columns



Autoregression by Rows masking

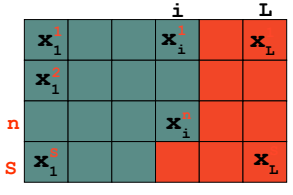


mask: (n, >i) & (>n)

$A_{row}[L,L,S]$ $A_{row}[i, j>i, n] = 0$

$A_{col}[L,S,S]$ $A_{col}[i, n, m>n] = 0$

Autoregression by Columns masking



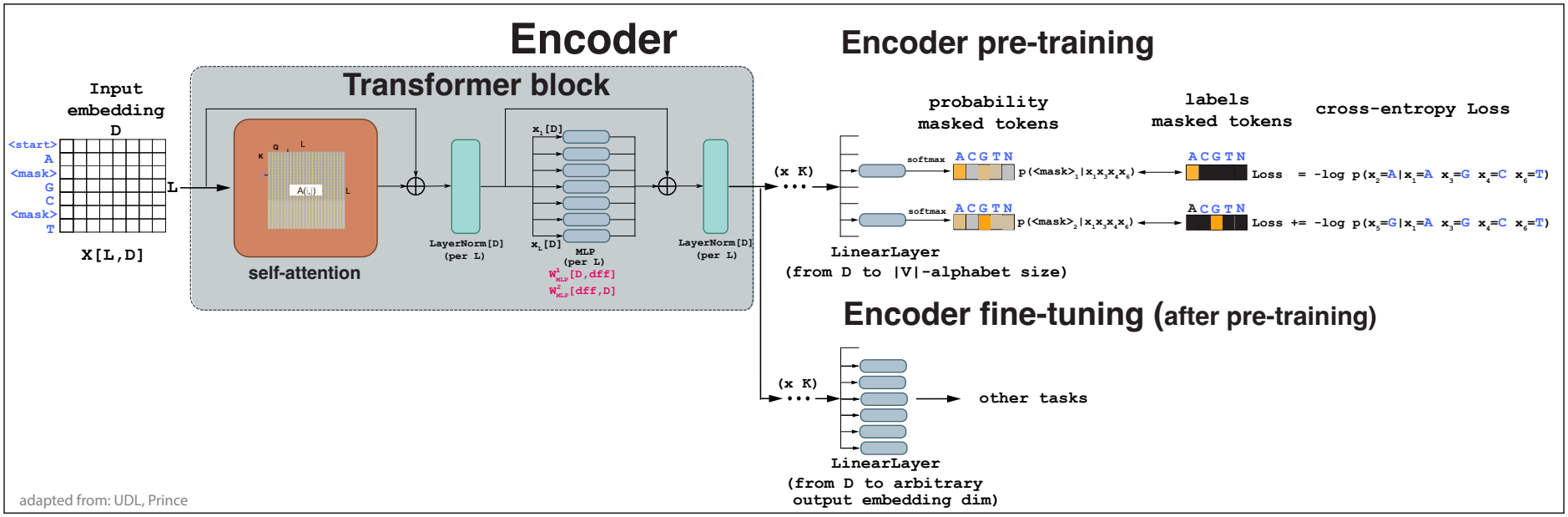
mask: (i, >n) & (>i)

$A_{row}[L,L,S]$ $A_{row}[i, j>i, n] = 0$

$A_{col}[L,S,S]$ $A_{col}[i, n, m>n] = 0$

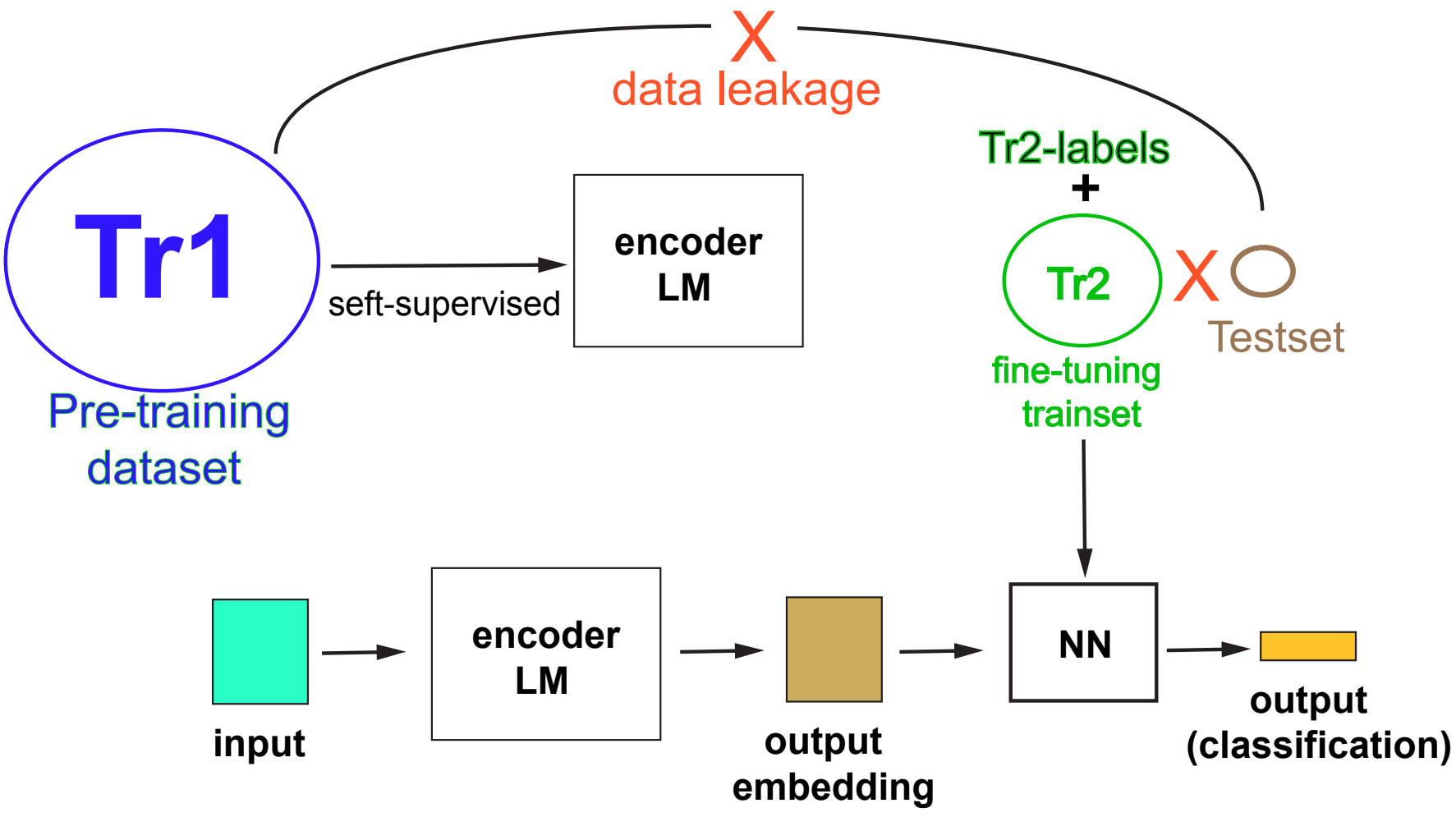
Encoders - BERT

DNABERT
ProteinBERT



adapted from: UDL, Prince

Encoder fine-tuning (after pre-training)



Attention is all you need

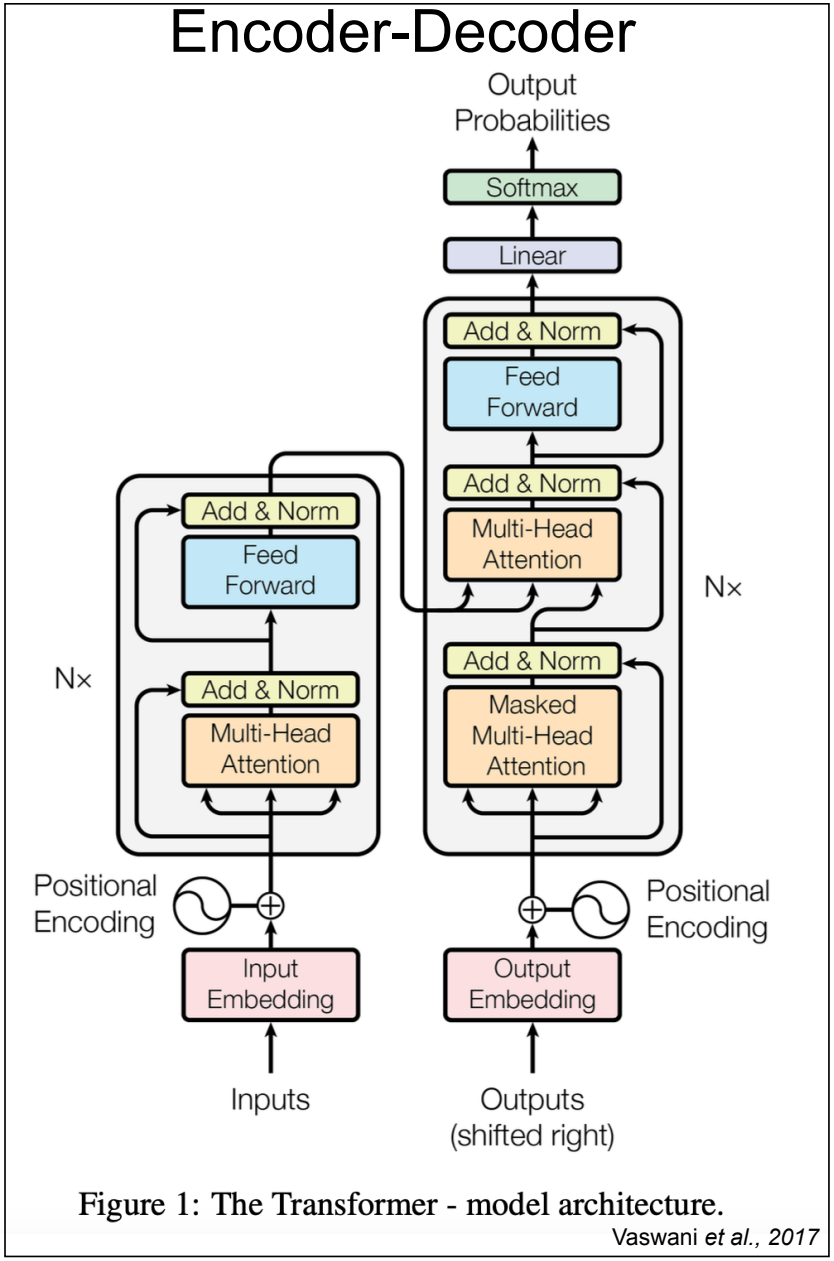


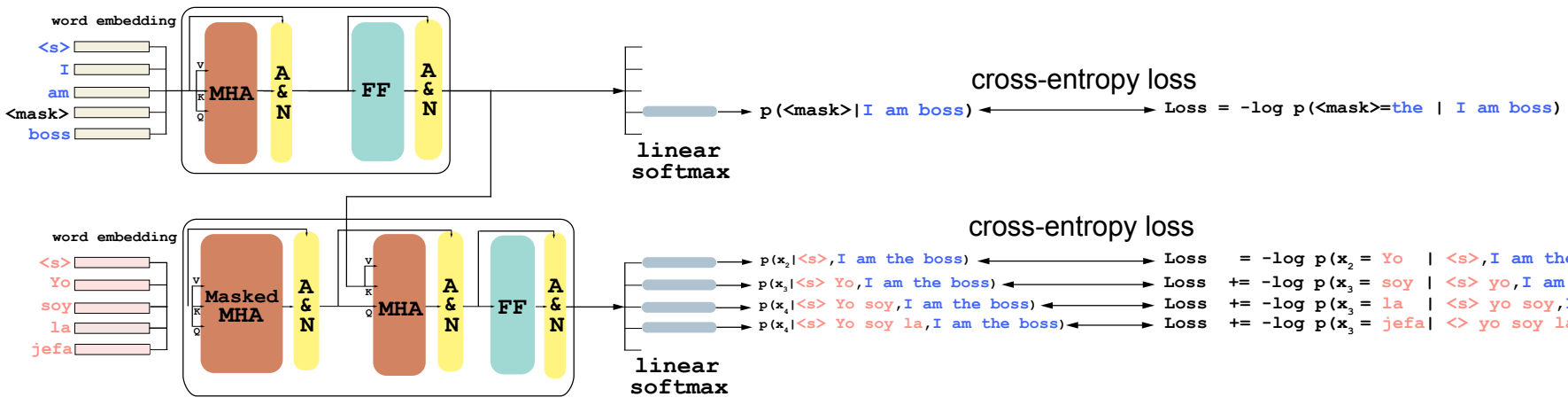
Figure 1: The Transformer - model architecture.

Vaswani *et al., 2017*

Encoder-Decoder training

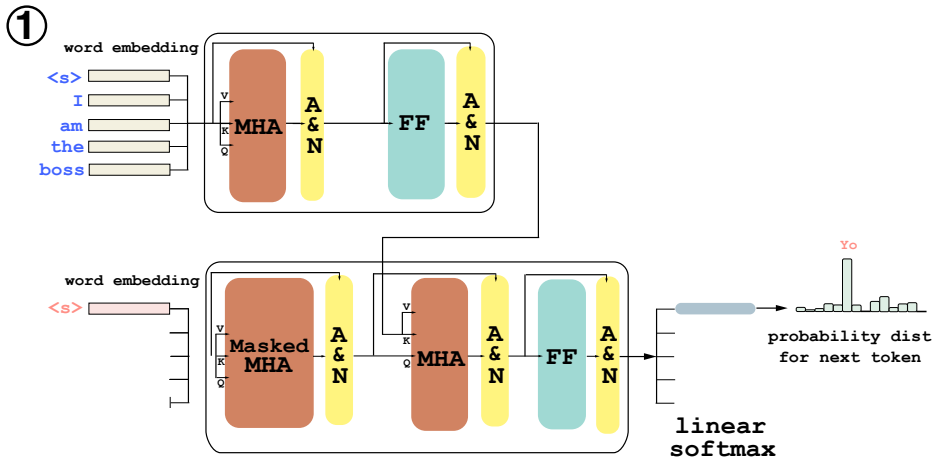
[I am the boss → Yo soy la jefa]

source sequence — target sequence



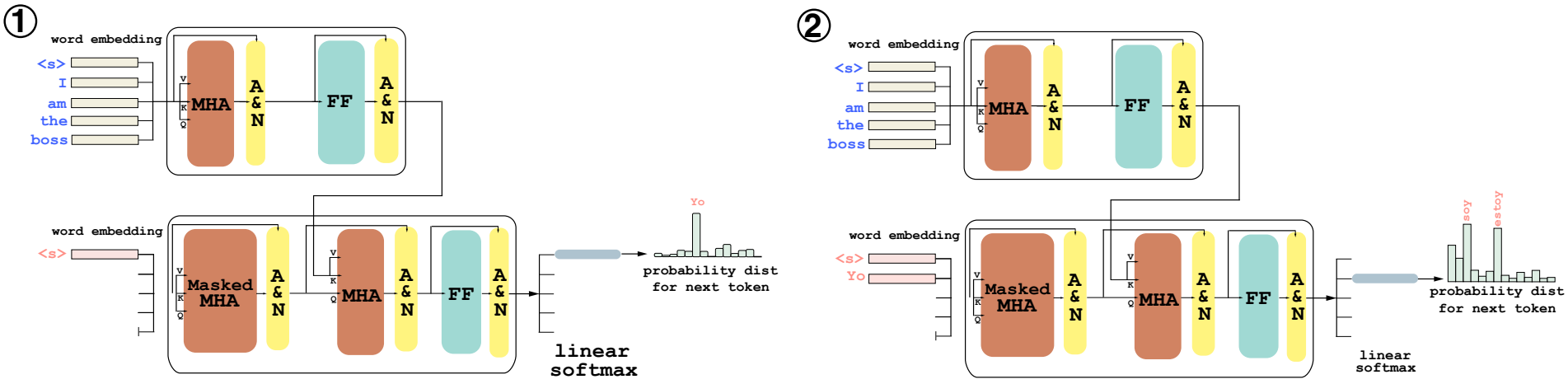
Encoder-Decoder inference (translation)

[I am the boss → ??]



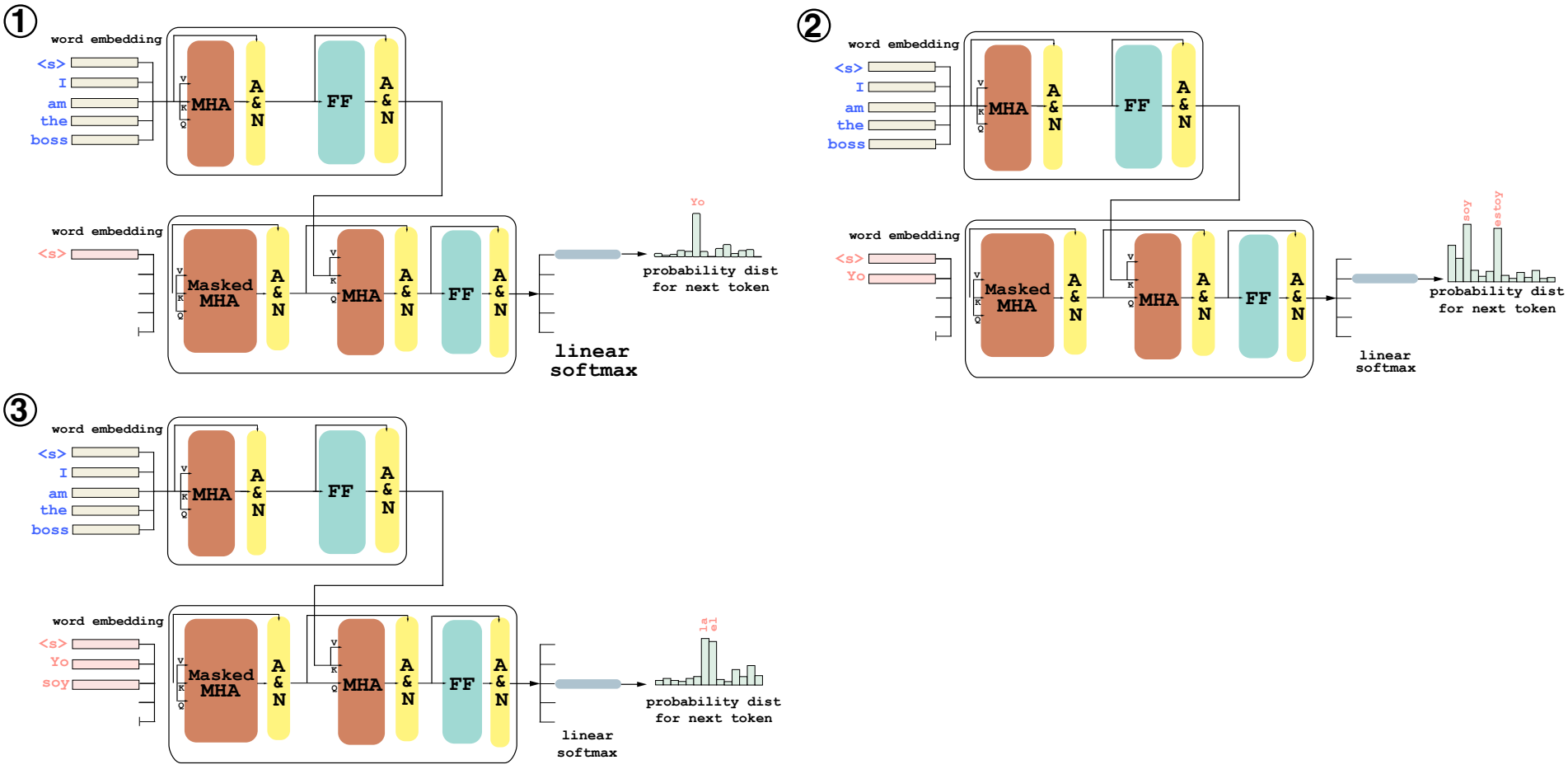
Encoder-Decoder inference (translation)

[I am the boss → ??]



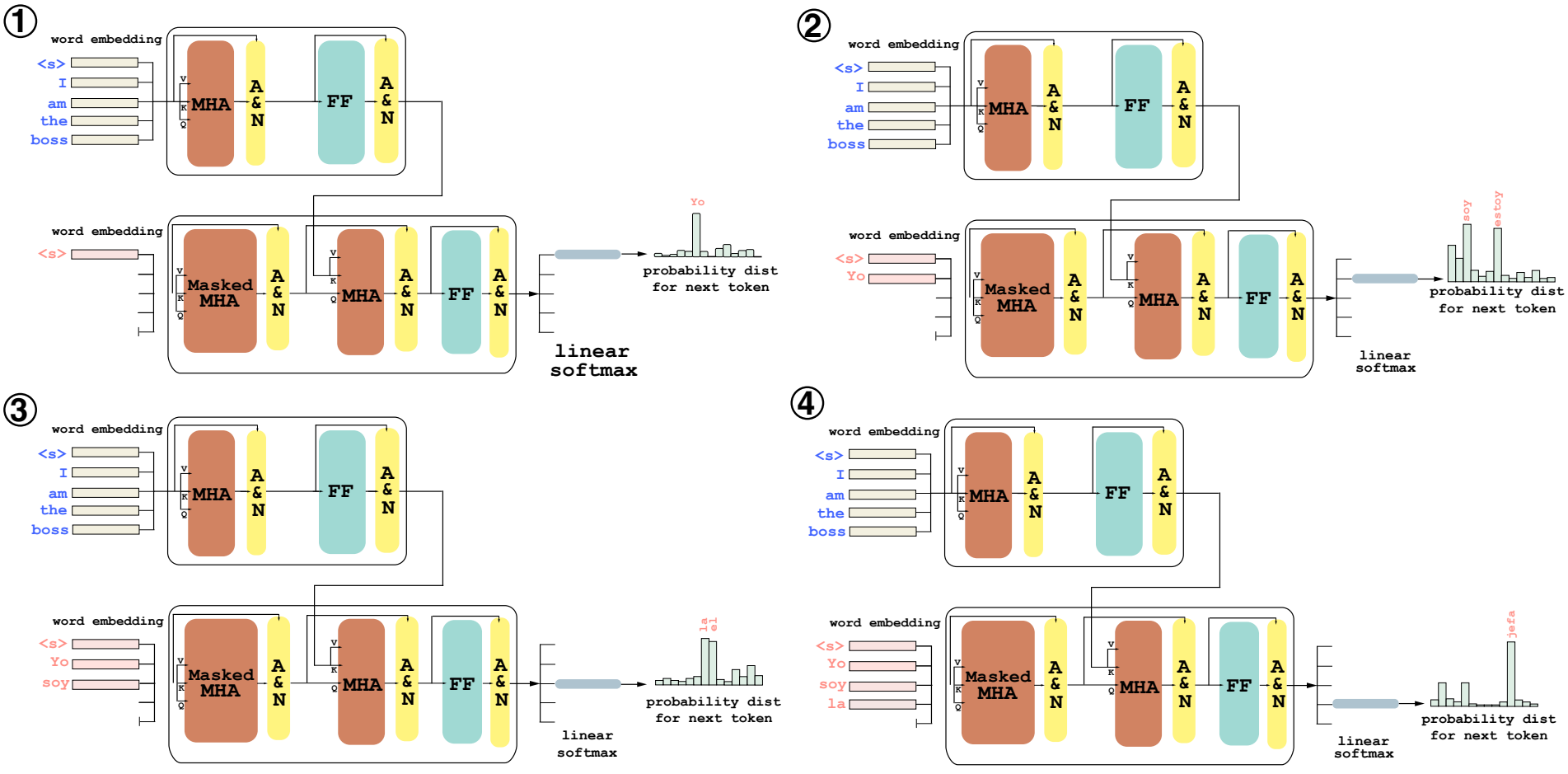
Encoder-Decoder inference (translation)

[I am the boss → ??]



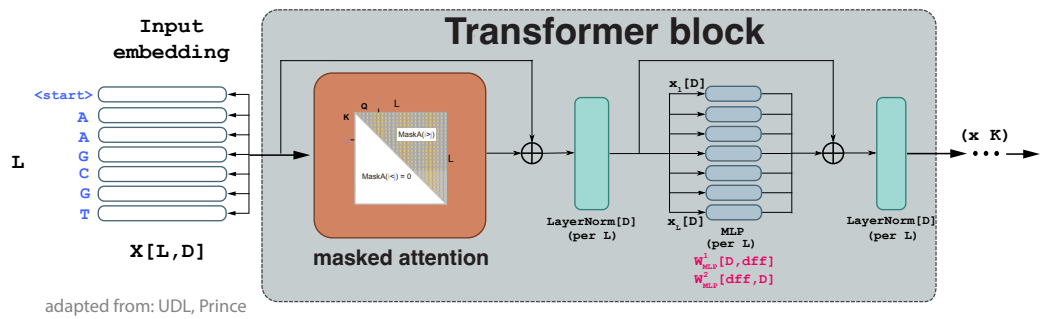
Encoder-Decoder inference (translation)

[I am the boss → ??]



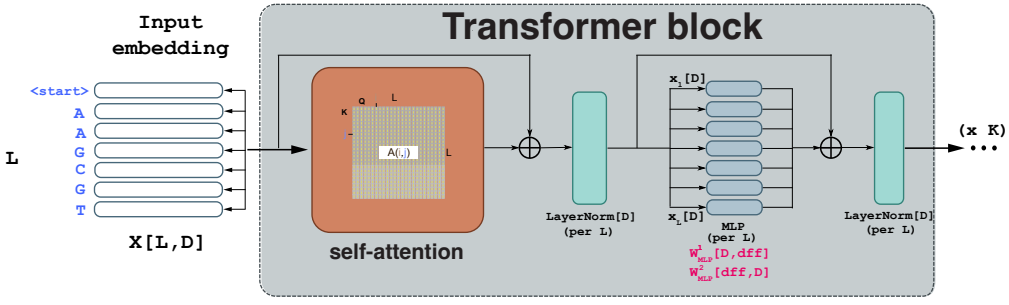
Transformer-based Language Models: Decoders vs Encoders

Decoder



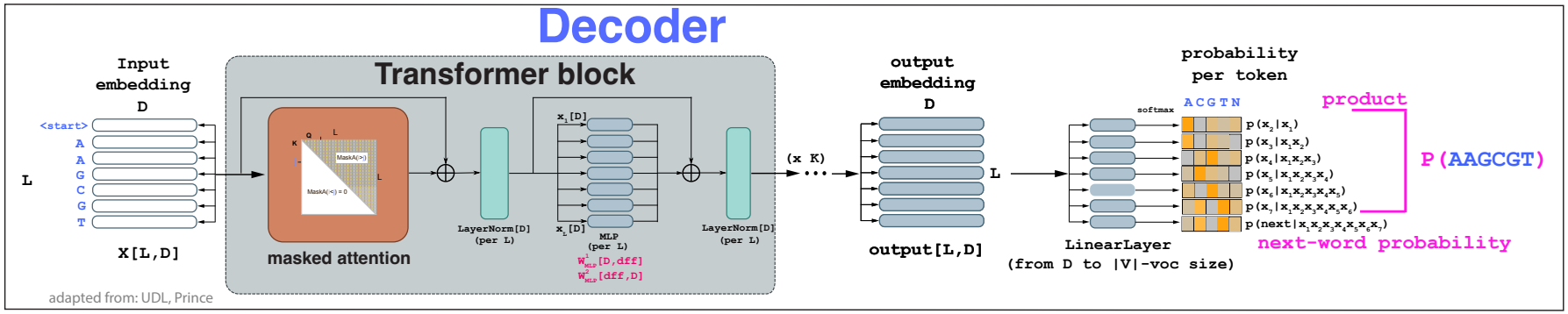
adapted from: UDL, Prince

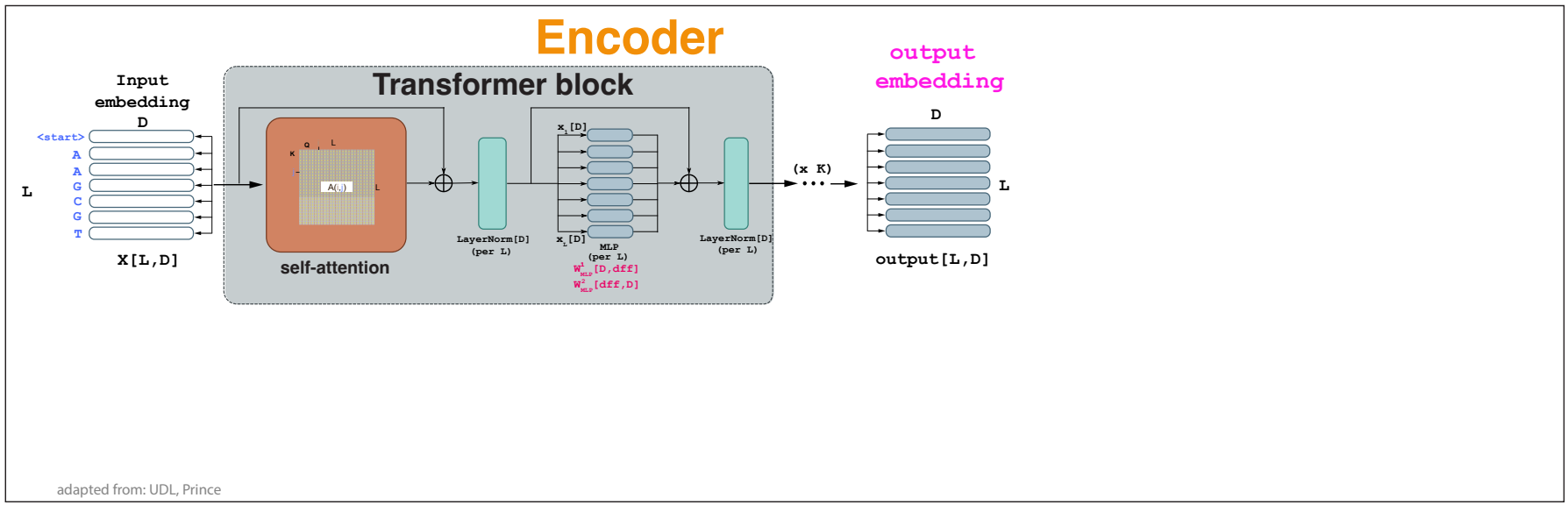
Encoder



adapted from: UDL, Prince

Decoders vs Encoders



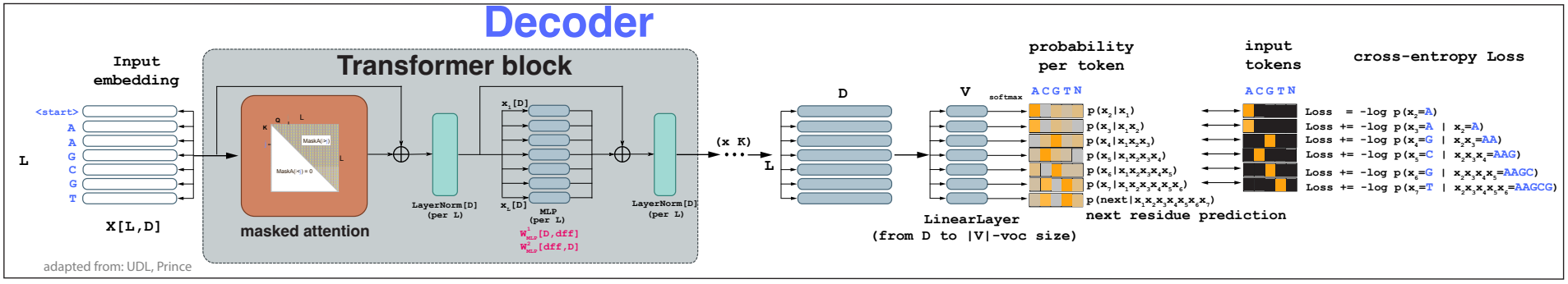


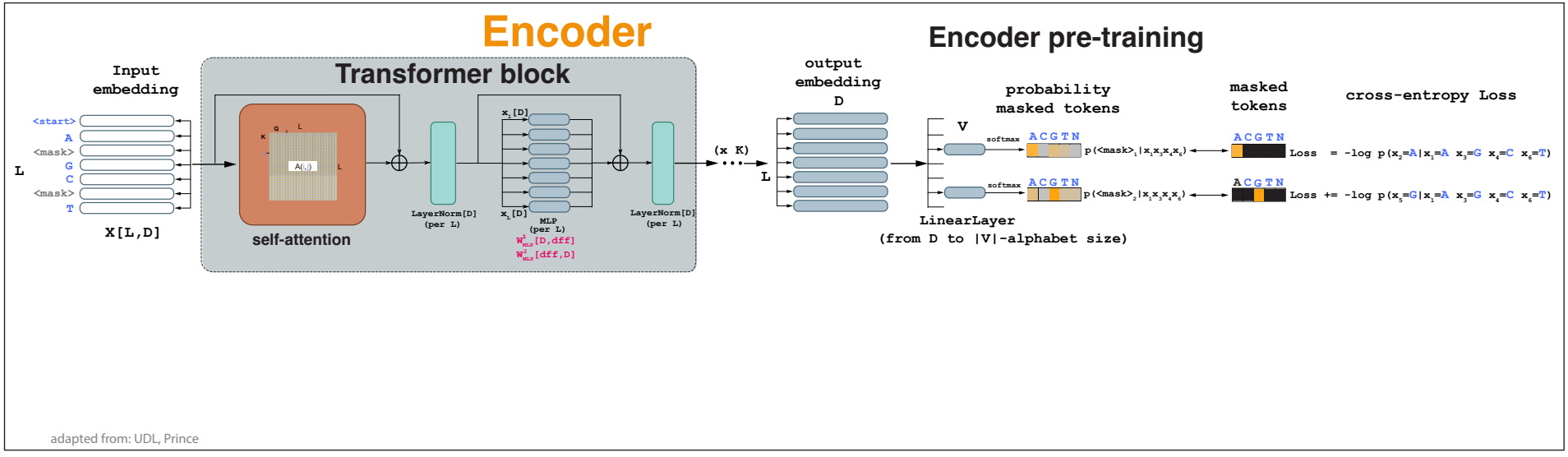
Outputs

Decoder: Generates the ***joint prob*** of input by autoregression
Generates ***next token***

Encoder: Produces an ***informative embedding*** of whole input

Decoders vs Encoders





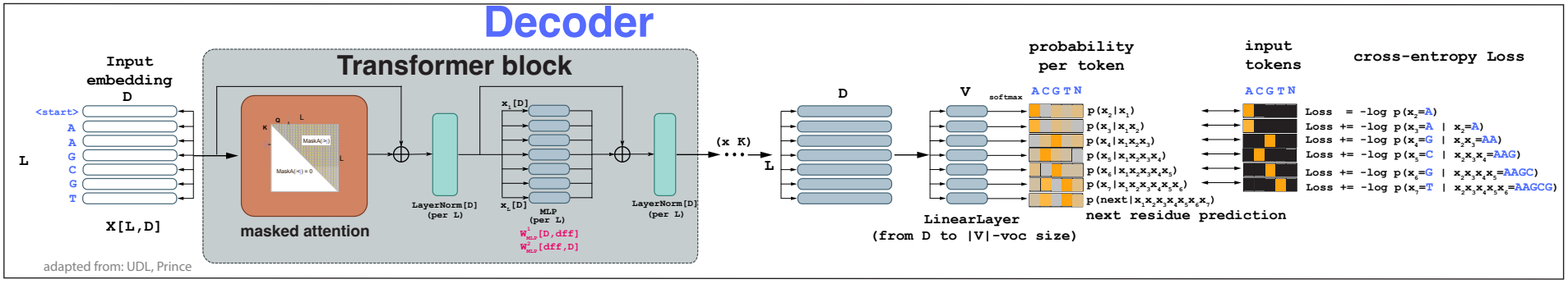
Training

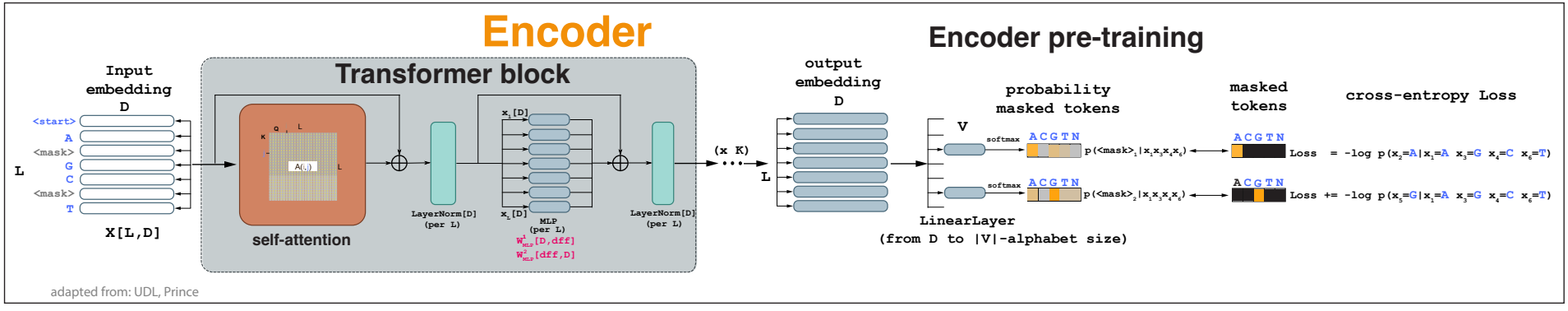
self-supervised

Decoder: ***predicts next*** token in training seqs

Encoder: ***predicts masked*** tokens in training seqs

Decoders vs Encoders





Training

self-supervised

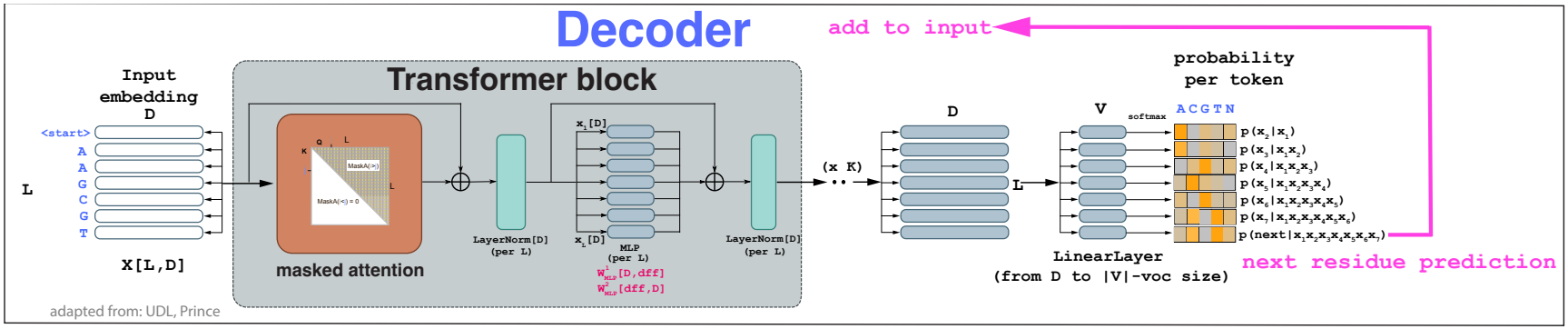
Decoder: ***predicts next*** token in training seqs

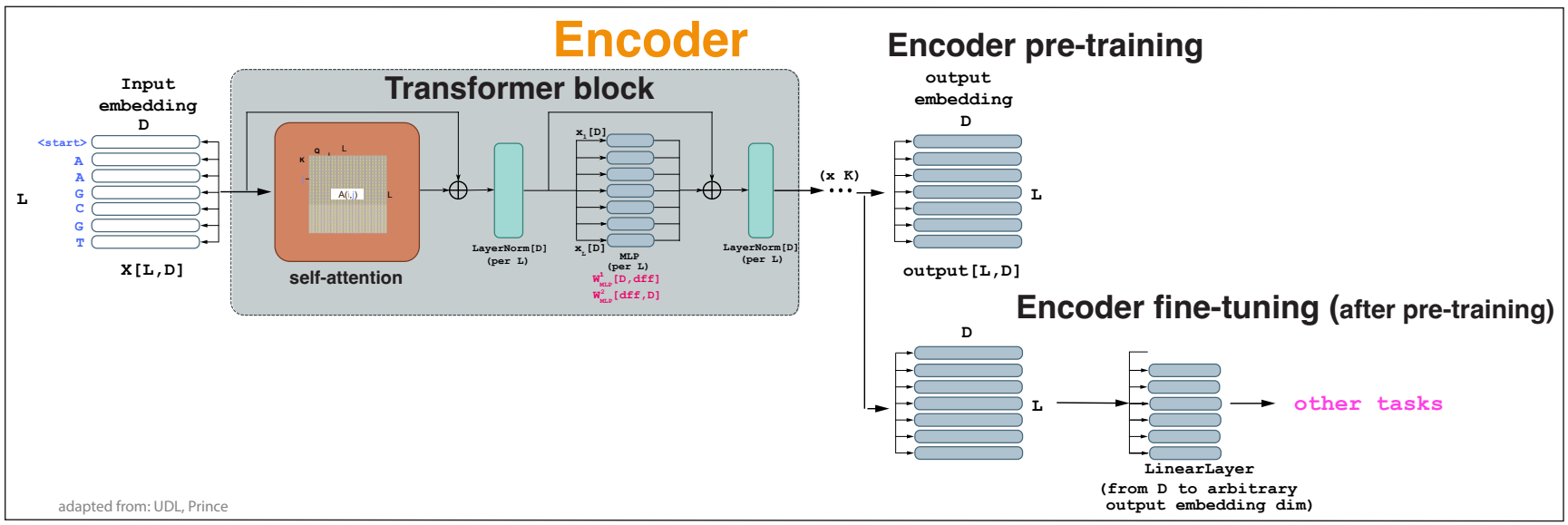
$$\text{Loss}(x_{1:L}) = -\left(\sum_i \log P(x_i \mid x_{1:i-1})\right)$$

Encoder: ***predicts masked*** tokens in training seqs

$$\text{Loss}(x_{\text{masked}} = -\left(\sum_{i \in \text{mask}} \log P(x_i \mid x_{\notin \text{mask}})\right)$$

Decoders vs Encoders



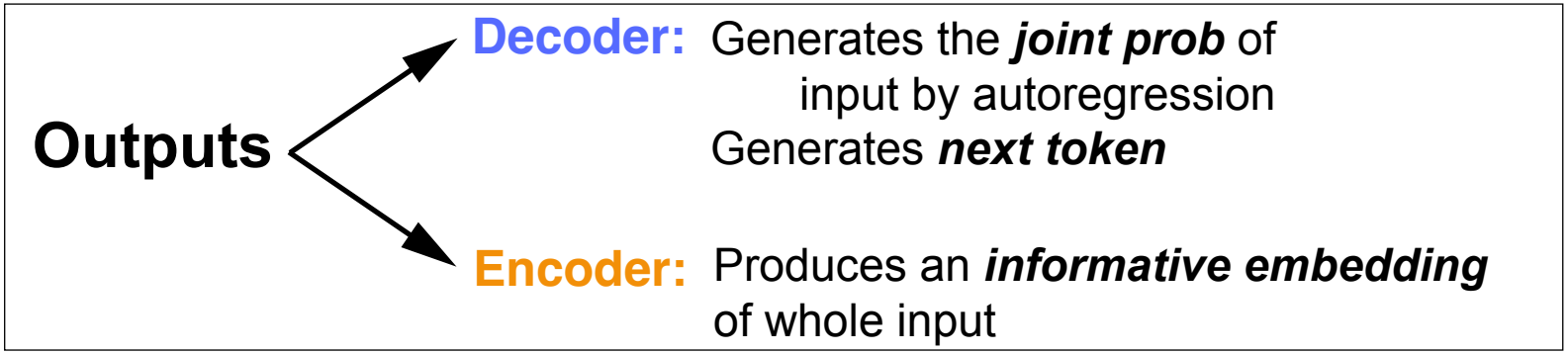


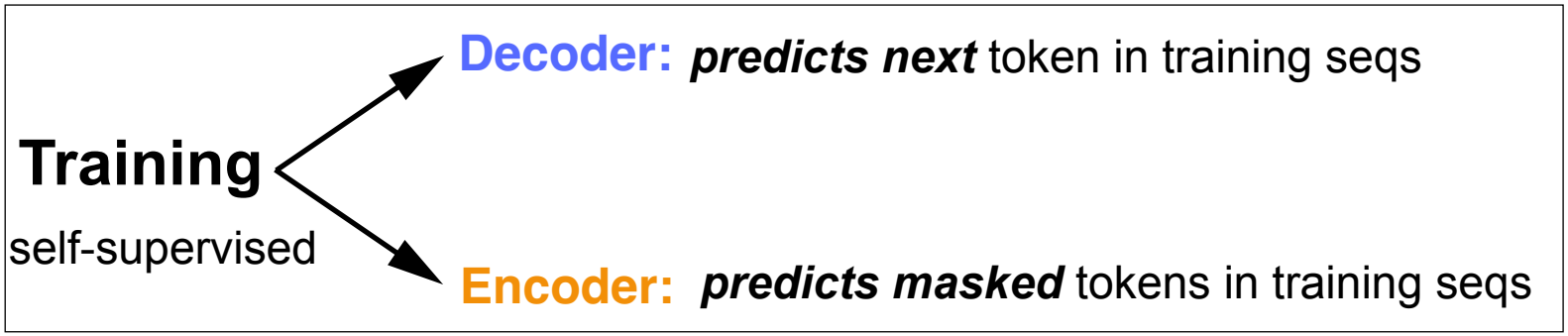
Usage

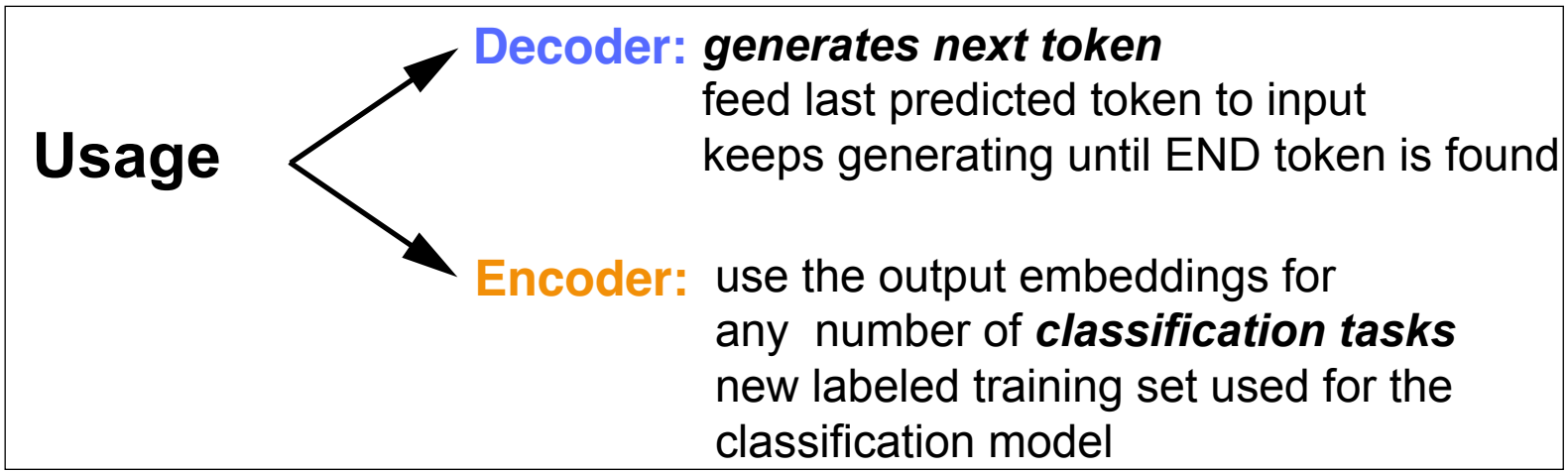
Decoder: feeds last predicted token to input
generates next token
keeps going until END token reached

Encoder: use the output embeddings for
any number of ***classification tasks***

Decoders vs Encoders







Is an ENCODER (BERT) a Language Model?

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Is an ENCODER (BERT)
a Language Model?

not really

Decoders vs Encoders

Outputs

Decoder: Generates the ***joint prob*** of input by autoregression
Generates ***next token***

Encoder: Produces an ***informative embedding*** of whole input

Training

self-supervised

Decoder: ***predicts next*** token in training seqs

both tasks are probabilistic and Language Model's "objectives"

Encoder: ***predicts masked*** tokens in training seqs

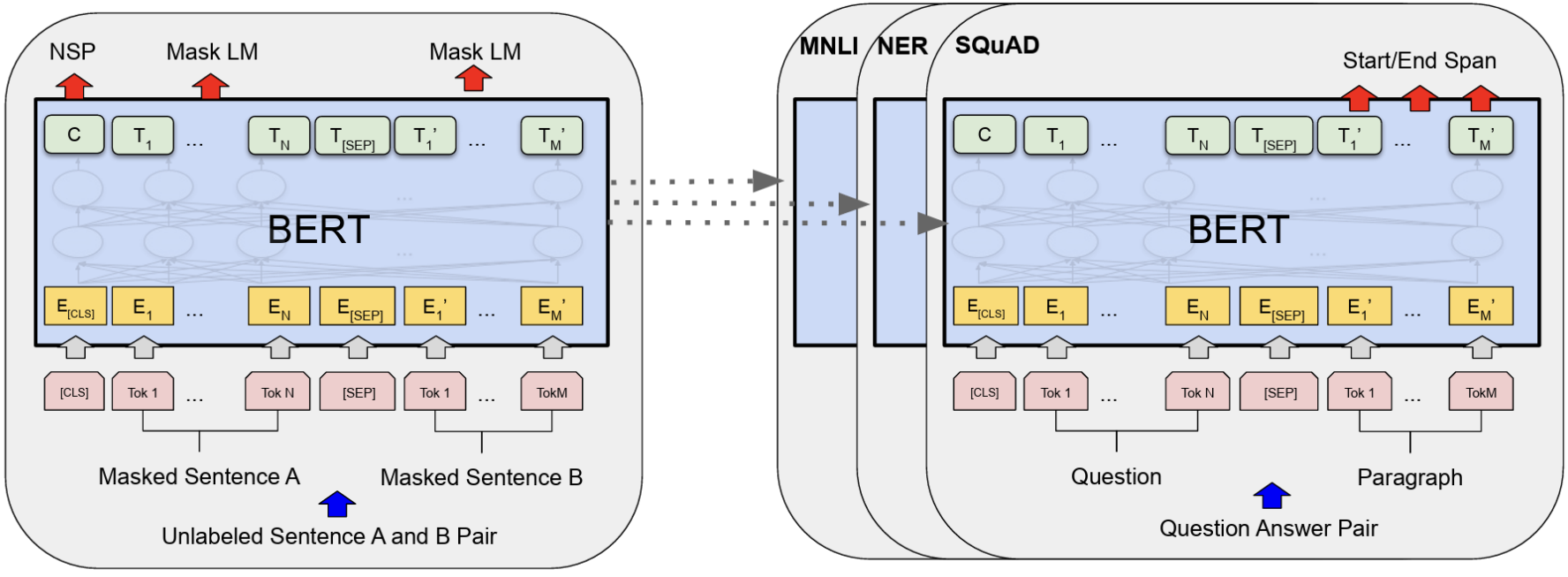
Usage

Decoder: ***generates next token***
feed last predicted token to input
keeps generating until END token is found

Encoder: use the output embeddings for any number of ***classification tasks***
new labeled training set used for the classification model

DNABERT = a DNA ENCODER

BERT: Bidirectional Encoder Representations from Transformers



Pre-training

Fine-Tuning

Pre-training Tasks:

Mask LM

Next Sentence Prediction (NSP)

[this taks requires labels]

Fine tuning tasks:

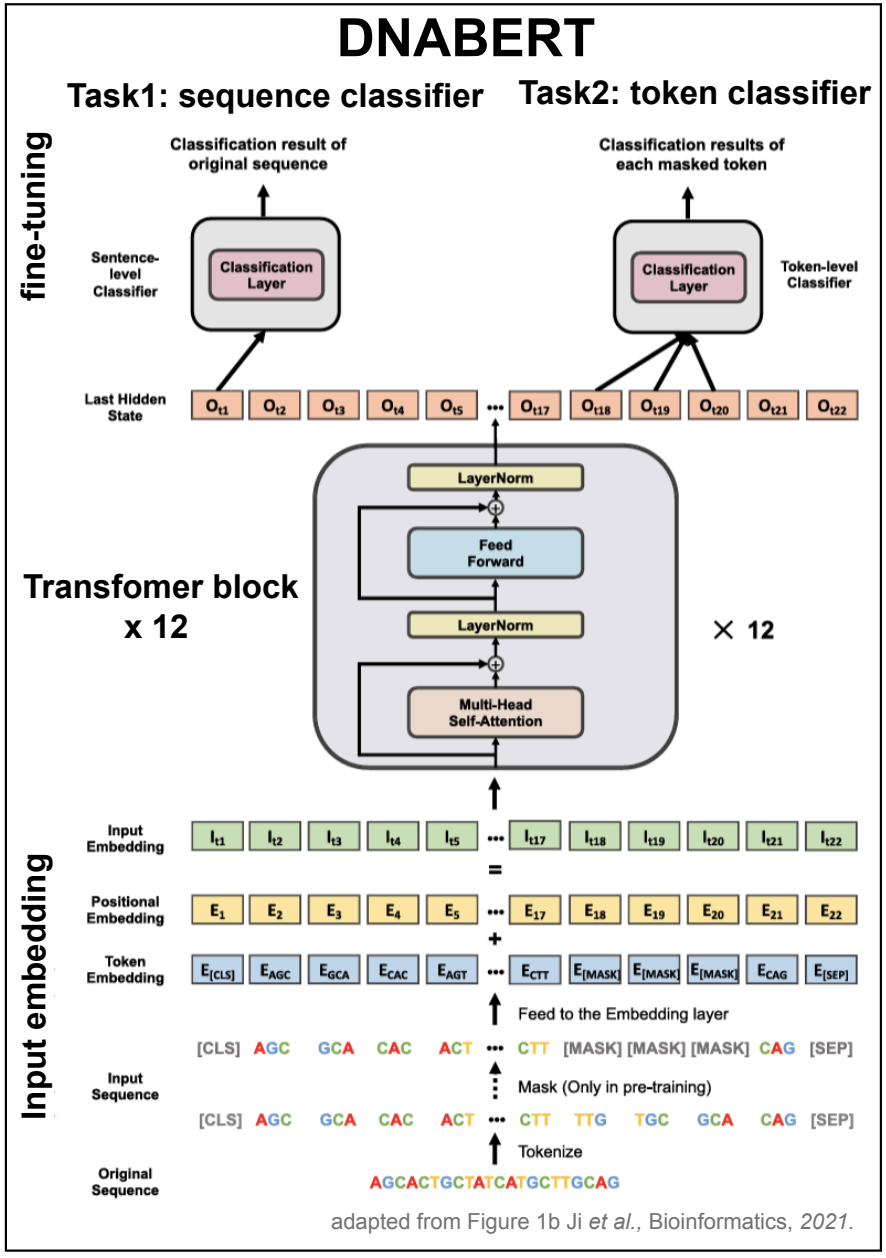
sentence pairs in paraphrasing

hypothesis-premise pairs

question/answer pairs

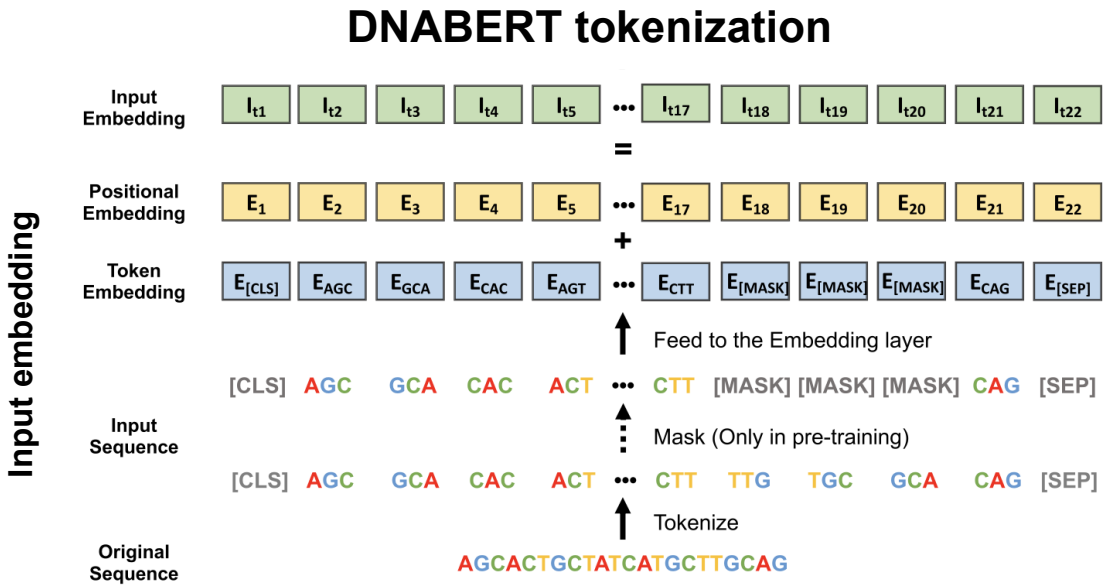
all parameters are re-trained

DNABERT = a DNA ENCODER



adapted from Figure 1b Ji *et al.,* Bioinformatics, *2021.*

DNABERT Tokenization



K-mers

alphabet size: $V = 4^k + 5$ extra tokens

>CLS> classification token
>PAD> padding token
>UNK> unknown token
>SEP> separation token
>MASK> masked token

DNABERT Tokenization

DNABERT tokenization

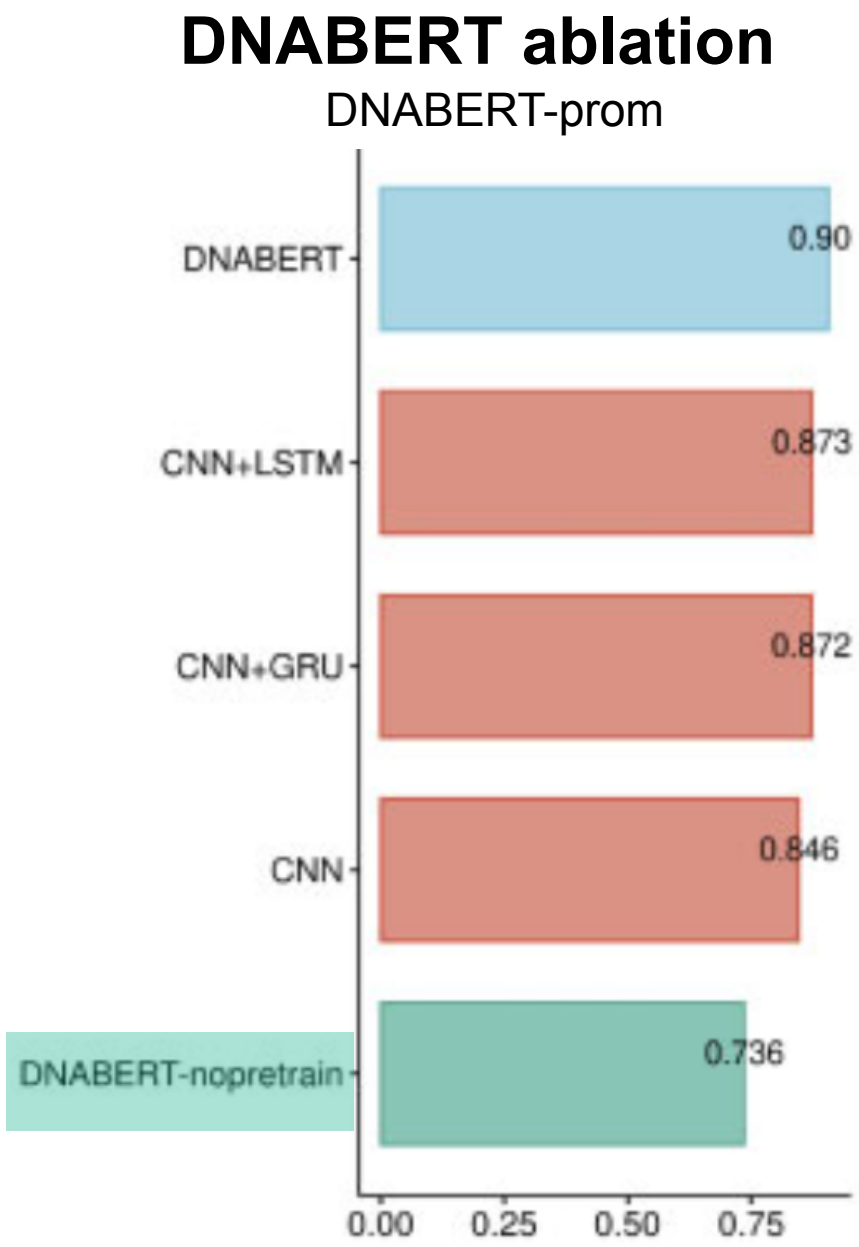
```python
# DNABERT-like tokenization
#
#
import torch
from itertools import product

# for k, the number of words in the alphabet is
# 4^k + 5
def build_kmer_abc(k=6):
    alphabet = ['A', 'C', 'G', 'T']
    kmers = [''.join(p) for p in product(alphabet, repeat=k)]
    print("Kmers", kmers)
    abc = {kmer: i+5 for i, kmer in enumerate(kmers)}  # reserve 0-4 for special
    abc['[CLS]']  = 0
    abc['[PAD]']  = 1
    abc['[UNK]']  = 2
    abc['[SEP]']  = 3
    abc['[MASK]'] = 4
    return abc

def dna_to_abc_idx(seq, abc, k=6):
    tokens = []
    tokens.append(abc['[CLS]'])
    for i in range(len(seq) - k + 1):
        kmer = seq[i:i+k]
        tokens.append(abc.get(kmer, abc['[UNK]']))
    tokens.append(abc['[SEP]'])
    return torch.tensor(tokens, dtype=torch.long)


abc3 = build_kmer_abc(k=3)
print("abc k=3. Alphabet size =", len(abc3))
seq = "ACGTACGT"
tokens_ids = dna_to_abc_idx(seq, abc3, k=3)
print("3-kmer tokenization of seq", seq, "\n", tokens_ids)
```

```
Kmers ['AAA', 'AAC', 'AAG', 'AAT', 'ACA', 'ACC', 'ACG', 'ACT', 'AGA', 'AGC', 'AGG', 'AGT', 'ATA', 'ATC', 'ATG',
'ATT', 'CAA', 'CAC', 'CAG', 'CAT', 'CCA', 'CCC', 'CCG', 'CCT', 'CGA', 'CGC', 'CGG', 'CGT', 'CTA', 'CTC', 'CTG',
'CTT', 'GAA', 'GAC', 'GAG', 'GAT', 'GCA', 'GCC', 'GCG', 'GCT', 'GGA', 'GGC', 'GGG', 'GGT', 'GTA', 'GTC', 'GTG',
'GTT', 'TAA', 'TAC', 'TAG', 'TAT', 'TCA', 'TCC', 'TCG', 'TCT', 'TGA', 'TGC', 'TGG', 'TGT', 'TTA', 'TTC', 'TTG',
'TTT']
abc k=3. Alphabet size = 69
3-kmer tokenization of seq ACGTACGT
 tensor([ 0, 11, 32, 49, 54, 11, 32,  3])
```

DNABERT fine-tuning classification tasks

1. The prediction of transcription factors binding sites.
   DNABERT-TF
2. The prediction of proximal and core promoter regions.
   DNABERT-Prom-300 (trained on TATA box human promoters)
   DNABERT-Prom-scan (trained on non-TATA box human promoters)
3. Recognition of canonical (GT-AG) and non-canonical splice sites.
   DNABERT-Splice
4. DNABERT-viz of attention maps

DNABERT importance of pre-training



from Figure 6f from Ji et al, 2021

BERT / DNABERT comparison

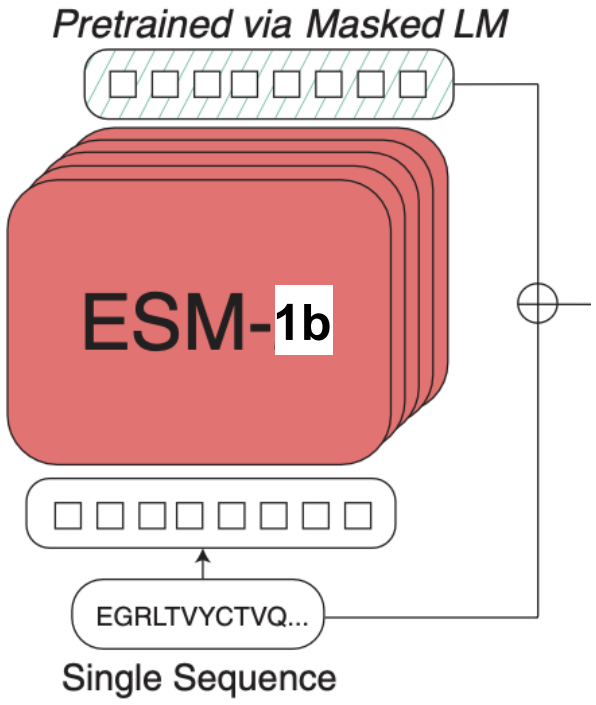
model comparison		**BERT**	**DNABERT-k**
vocabulary	V	30,000 tokens	4^k + 5
token size	k		k-mers: 3,4,5,6
input embedding	D	1,024	768
Transformer layers	K	24	12
Transformer heads per layer	h	16	12
query, key, value dimension per head	D_h	64	64
FF hidden dimension	d_ff	4,096	3,072
max length of input	Lmax	512 tokens	512 tokens
pre-training	steps	1,000,000	
pre-training	epochs	50	
pre-training	words	3.3 billions	

ESM-1b (2020)

Transformer for proteins trained by sequence masking

"Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences"

ESM-1b Transformer



Transformer

250 million sequences

650 million parameters

downstream tasks

biological variations
(aromatic, polar, hydrophobic)

remote homology

alignments of protein families

prediction of 2D and 3D structure

ESM-1b (2020)

ESM-1b Transformer

downstream tasks

biological variations (aromatic, polar, hydrophobic)

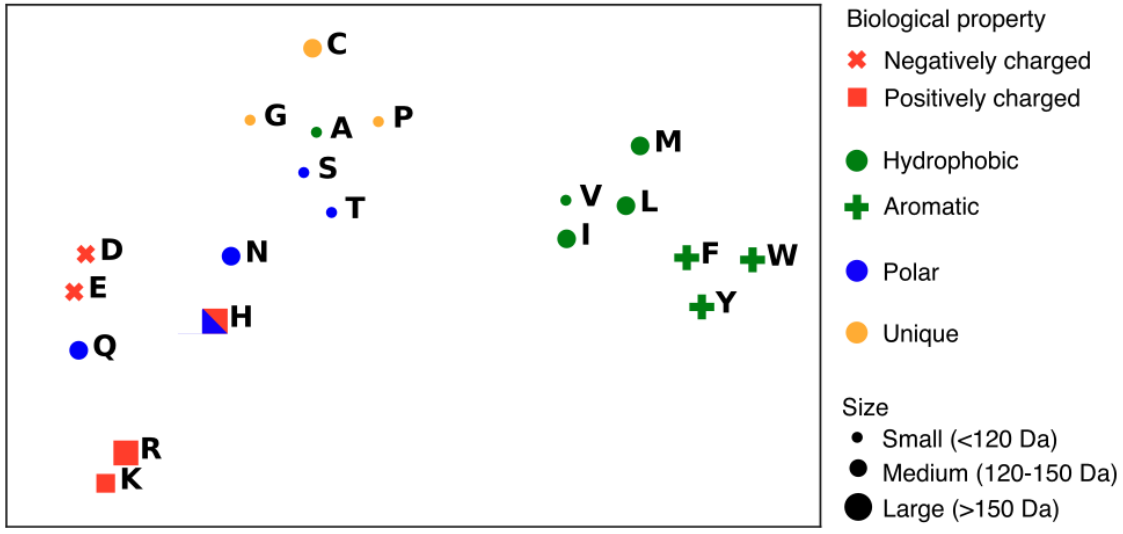


Fig. 1. Biochemical properties of amino acids are represented in the Transformer model's output embeddings, visualized here with t-SNE. Through unsupervised learning, residues are clustered into hydrophobic, polar, and aromatic groups and reflect overall organization by molecular weight and charge. Visualization of 36-layer Transformer trained on UniParc.

ESM-1b (2020)

ESM-1b Transformer

downstream task

Table 2. Remote homology detection

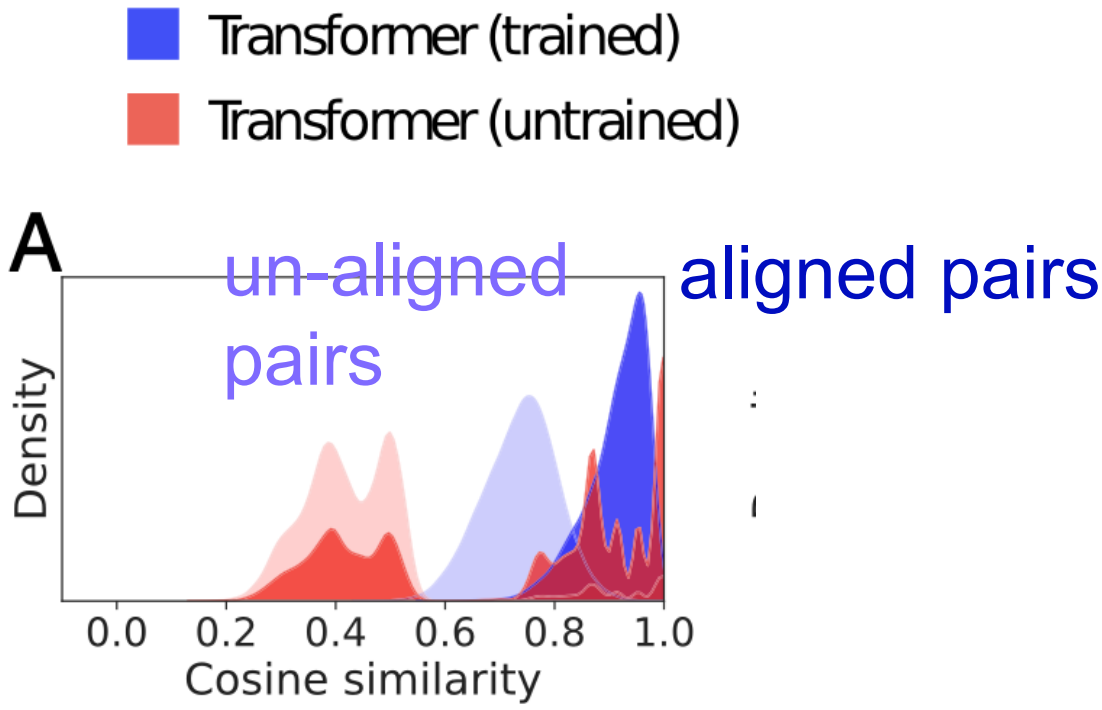
		Hit-10		AUC	
	Pretraining	Fold	SF	Fold	SF
HHblits*		0.584	0.965	0.831	0.951
LSTM (S)	UR50/S	0.558	0.760	0.801	0.863
LSTM (L)	UR50/S	0.574	0.813	0.805	0.880
Transformer-6	UR50/S	0.653	0.878	0.768	0.901
Transformer-12	UR50/S	0.639	0.915	0.778	0.942
Transformer-34	(None)	0.481	0.527	0.755	0.807
Transformer-34	UR100	0.599	0.841	0.753	0.876
Transformer-34	UR50/D	0.617	0.932	0.822	0.932
Transformer-34	UR50/S	0.639	0.931	0.825	0.933
ESM-1b	UR50/S	0.532	0.913	0.770	0.880

ESM-1b (2020)

ESM-1b Transformer

downstream task

alignment information

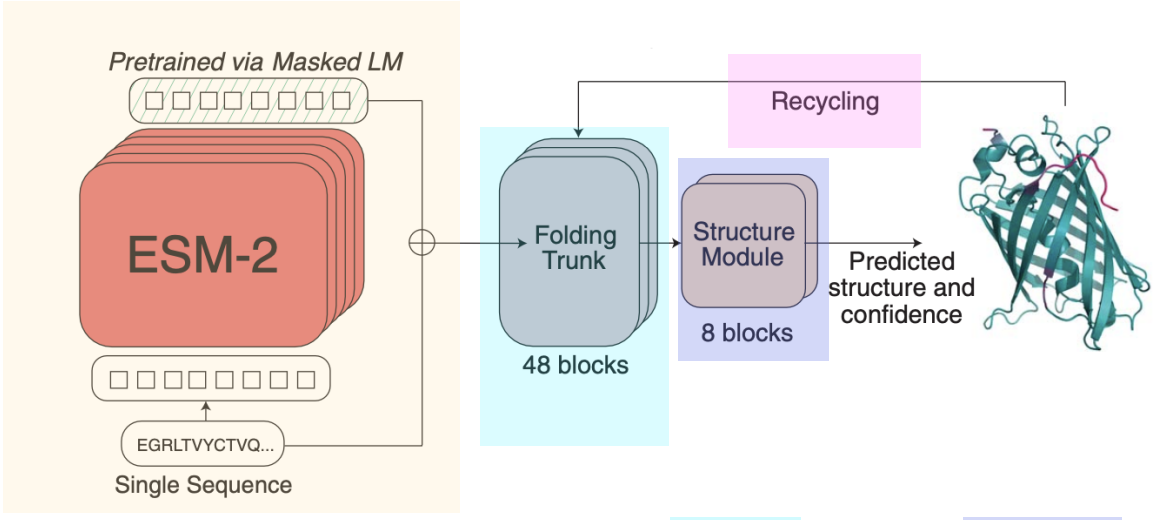


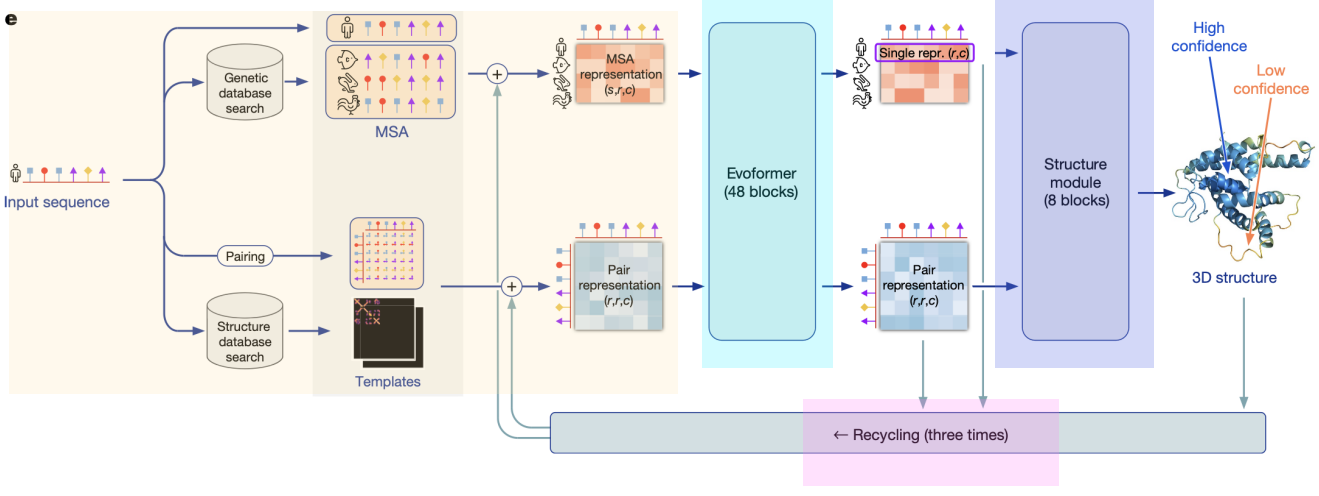
Overall

ESM2 (2023)

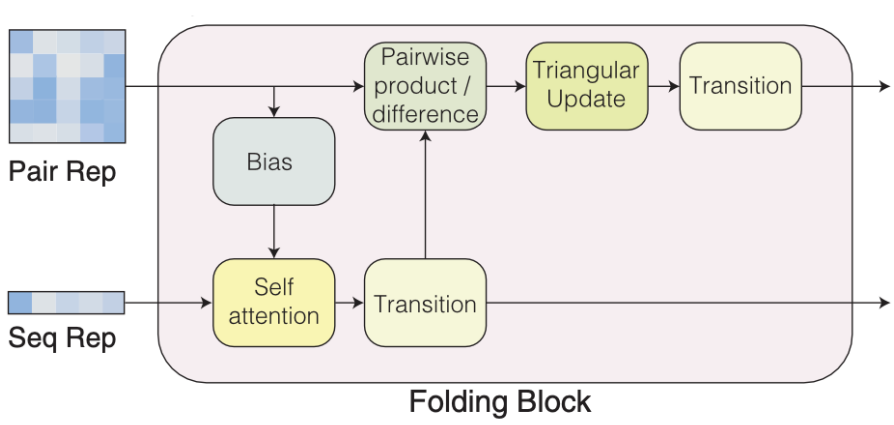
"Evolutionary-scale prediction of atomic-level protein structure with a language model"

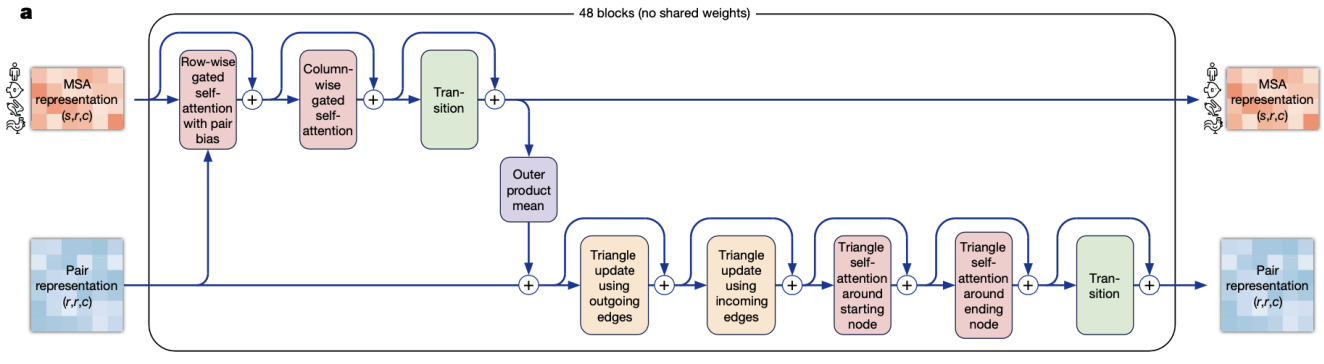
AF2 vs ESM-2





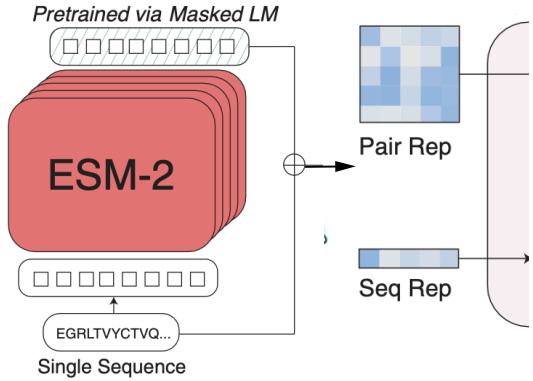
ESM-2 vs AF2


ESM-2 vs AF2
Pairwise product / difference
Triangular Update
Transition
Bias
Pair Rep
Self attention
Transition
Seq Rep
Folding Block

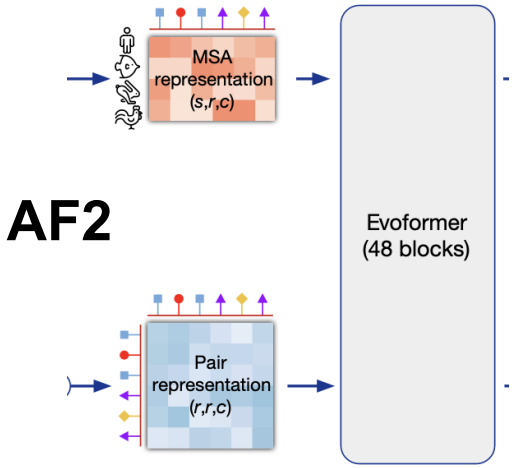

a
48 blocks (no shared weights)
MSA representation (s,r,c)
Row-wise gated self-attention with pair bias
Column-wise gated self-attention
Tran-sition
Outer product mean
Pair representation (r,r,c)
Triangle update using outgoing edges
Triangle update using incoming edges
Triangle self-attention around starting node
Triangle self-attention around ending node
Tran-sition
MSA representation (s,r,c)
Pair representation (r,r,c)

Atomic-resolution structure emerges in language models trained on protein sequences

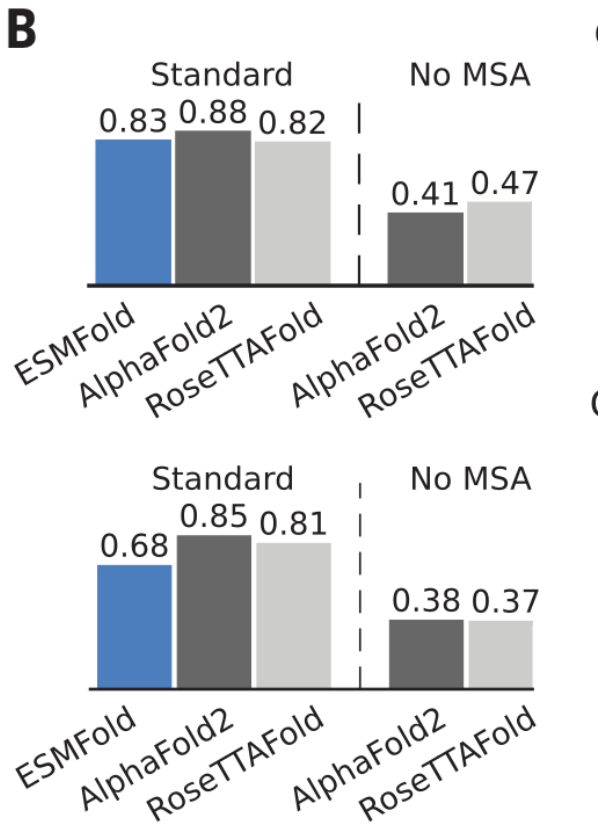
Encoder language models trained on sequences only
vs
models using alignments



150 million parameters
65 million unique sequences.



90–100 million parameters



Decoders

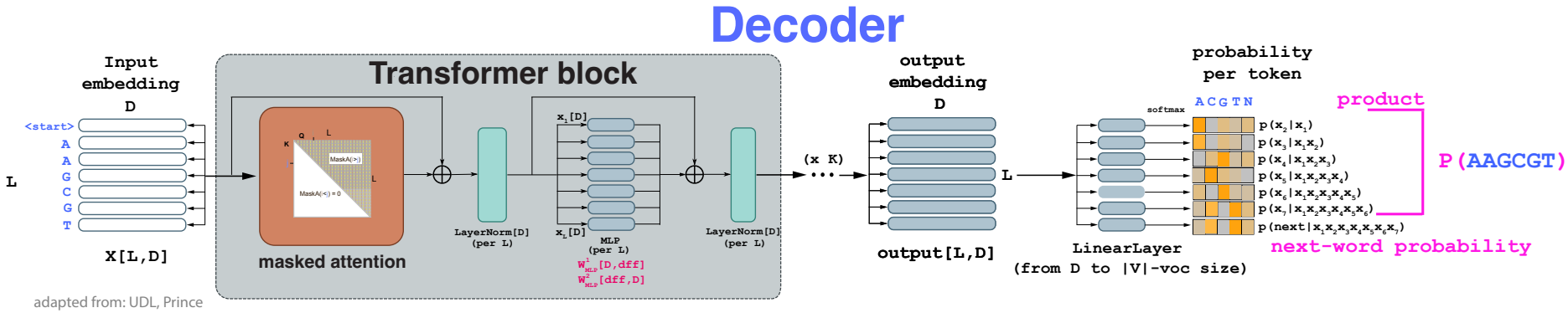
ProGen2 - to sample protein sequences

EVO - to sample genomic sequences

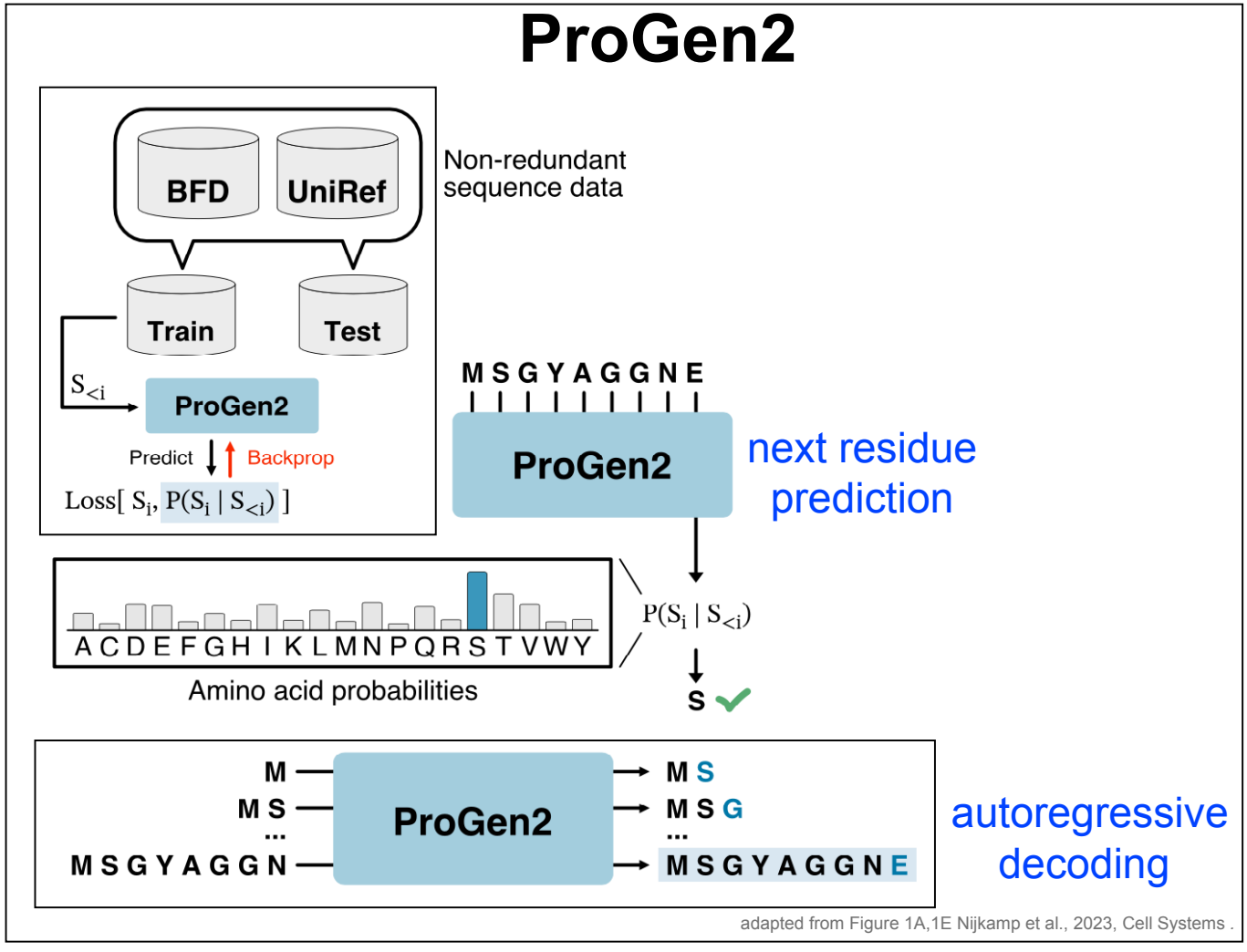
ProGen2

Exploring the boundaries of protein language models

ProGen2 = "Causal LLMs in the form of autoregressive decoders for the modeling of proteins"

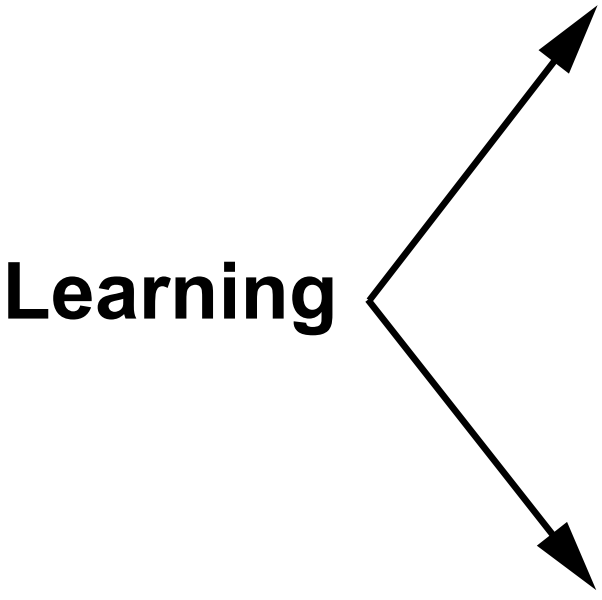


ProGen2



adapted from Figure 1A,1E Nijkamp et al., 2023, Cell Systems .

Types of Learning



encoders = masked LMs

Fine-tuning: (training)

uses pre-trained weights
needs task-specific data
weights are updated

decoders = autoregressive LMs

Few-shot learning: (no-training)

give a few examples at inference time

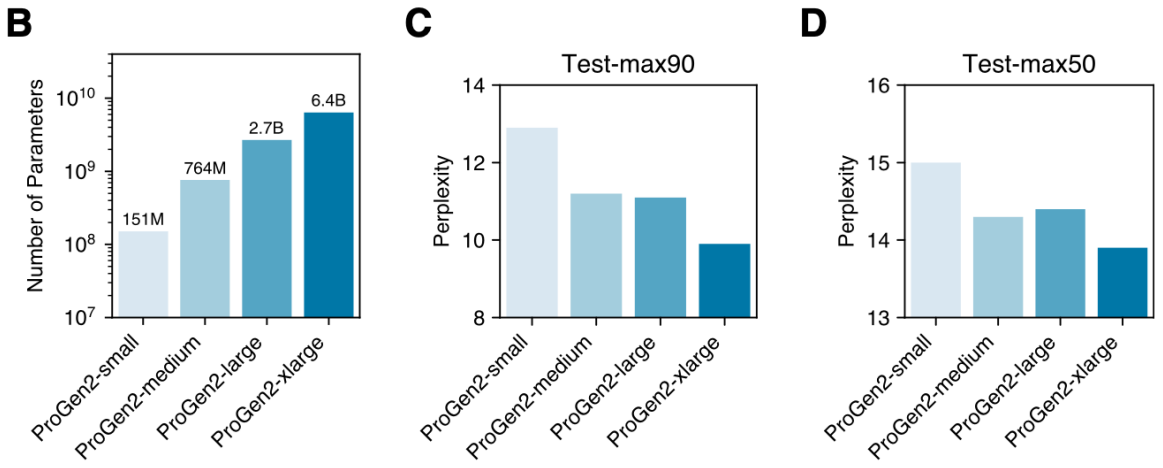
no weights are updated

Zero-shot learning:

give some specs, but no examples

Perplexity

Capturing the distribution of observed proteins



i)

perplexity

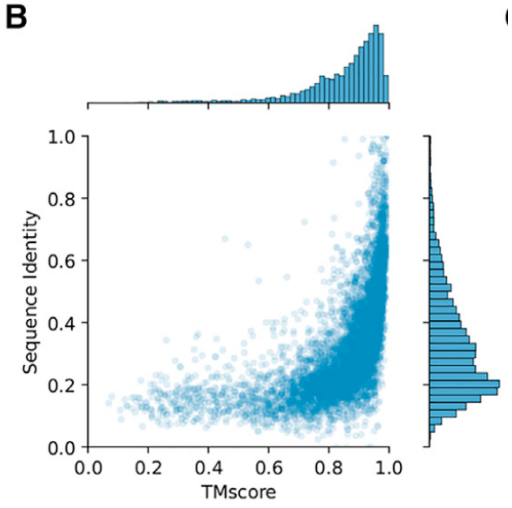
$$\mathrm{PPL}(x_{1:L}) = 2^{\frac{1}{L}\mathrm{Loss}(x_{1:L})} = 2^{-\frac{1}{L}\sum_i \log P(x_i|x_{1:i-1})} = \left(\prod_i P(x_i \mid x_{1:i-1})\right)^{-1/L},$$

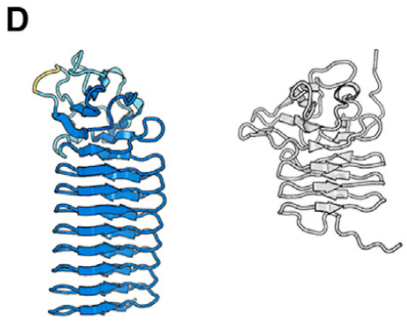
- Perplexity ranges in values from 1 (Loss = 0) to the vocabulary size |V| **why** ?
- Perplexity, by taking the average over token of the loss, is independent of length of the sequence.
- Perplexity does not depend on the log base used.

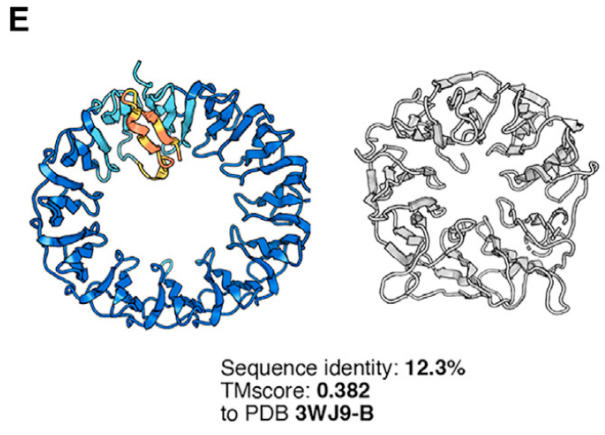
The lower the perplexity the better. Perplexity intuitively measures the number of tokens that your are hesitating betweer

Sequences adopt natural folds with sequence divergence

ProGen2 sequences: known folds with high seq diversity

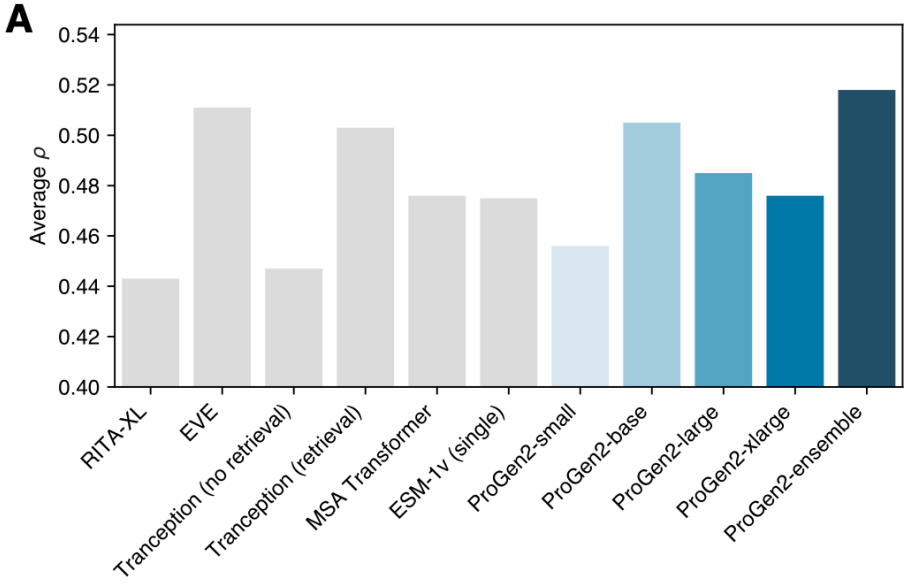






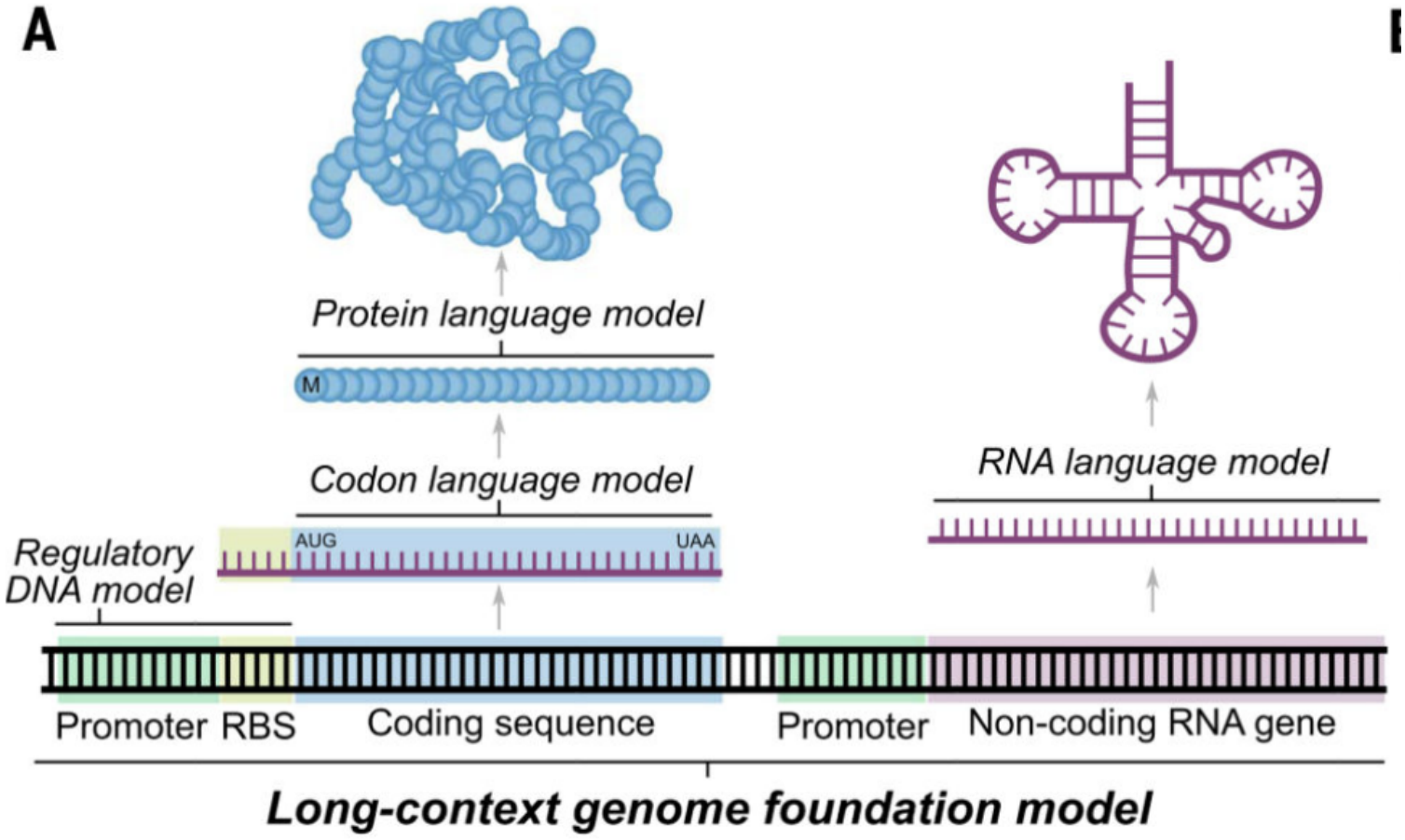
Progen2 zero-shot fitness predictions

ProGen2 zero-shot fitness



“To improve model performance we must carefully consider the alignment of the pretraining dataset and the downstream task”

EVO: a decoder for genomic sequences



EVO

EVO:
Sequence modeling and design from molecular to genome scale

7 billion paramers.

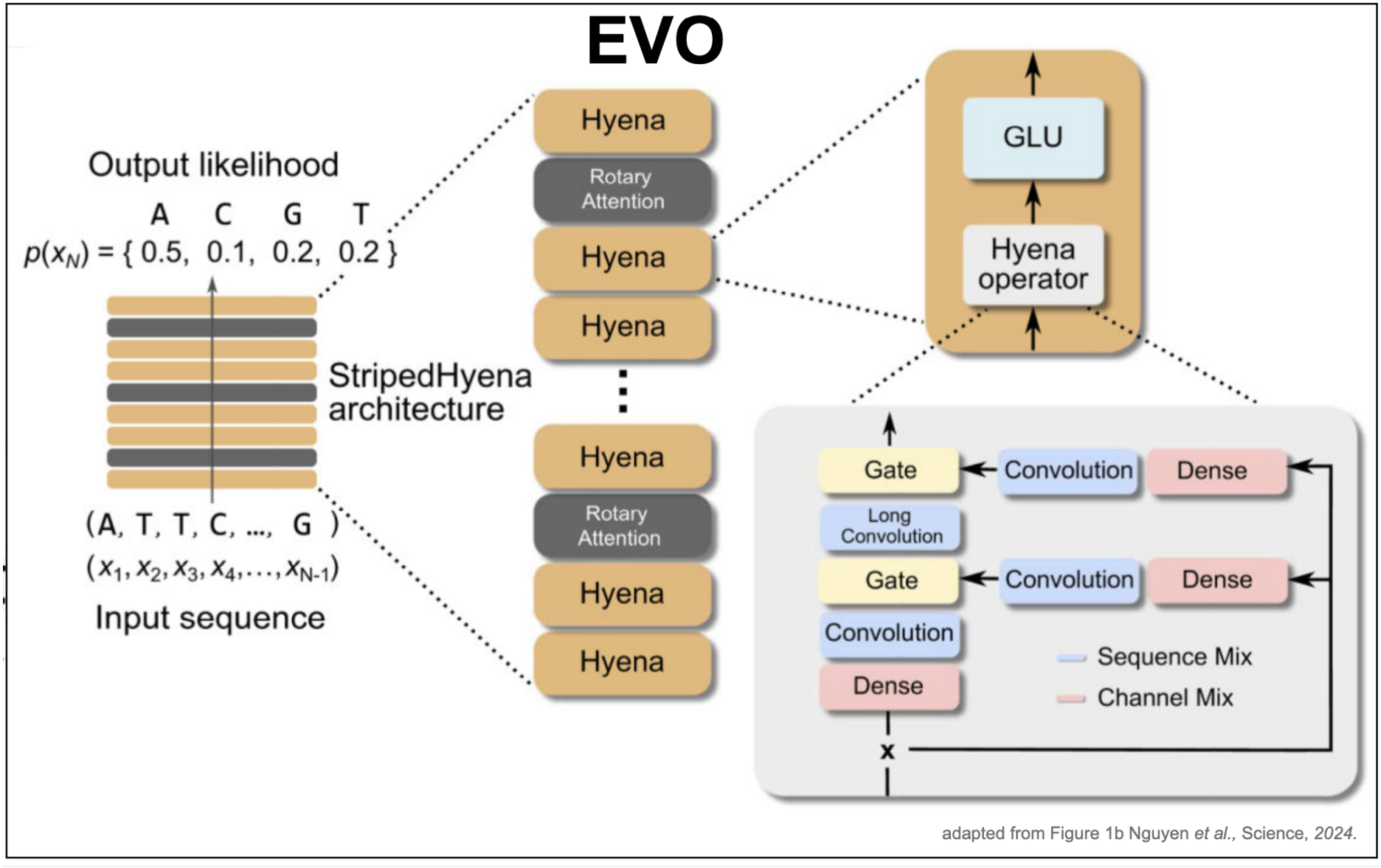
a context length of 131 kilobases at single-nucleotide resolution.

Trained on 2.7 million prokaryotic and phage genomes

zero-shot function prediction across DNA, RNA, and protein modalities

StripedHyena architecture

EVO

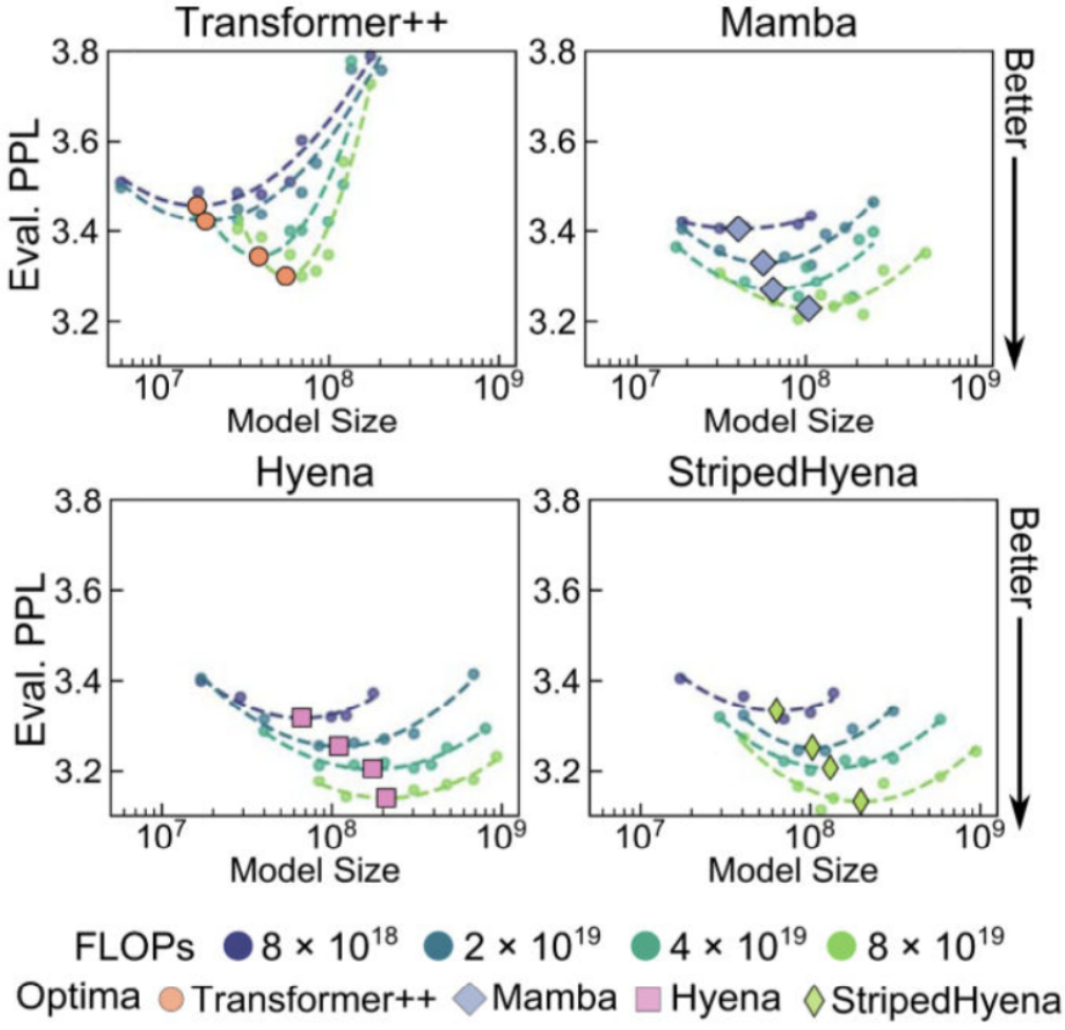


adapted from Figure 1b Nguyen *et al.*, Science, *2024.*

The stripedHyena architecture

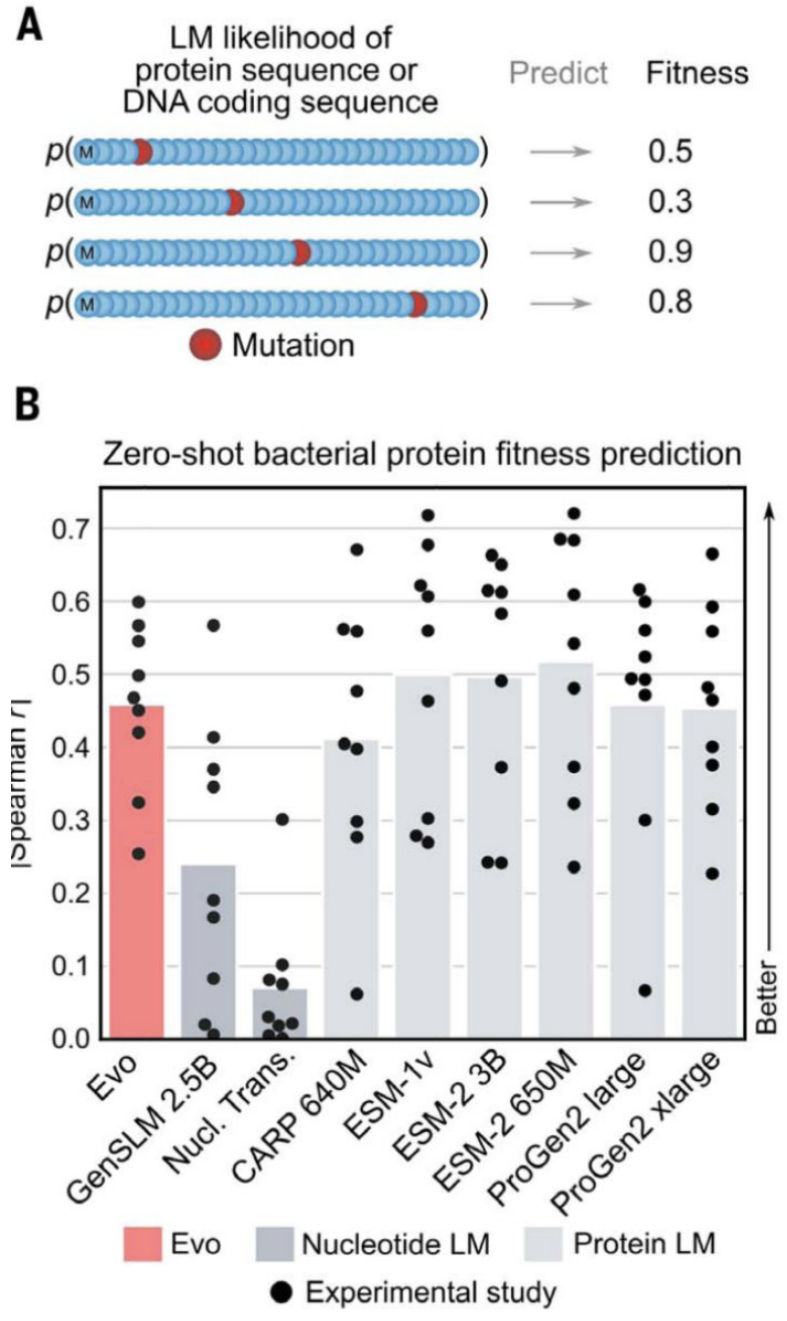
```
* LayerNorm

* Striped Self-attention

* Hyena gated block, which includes:

  * LayerNorm
      input x           # (B, L, d_model)
      residual = x
      x = LayerNorm(x)

  * Two x projections: d_model = 2*hidden_dim
      u, v # (B, L, hidden_dim)

  * One long convolution on u:
      u_cov = cov1d(u)   #(B, L, hidden_dim)

It is called long convolution because the filter spans most of the length of the sequence.
The hope is that such long convolution will help learn long-distance interaction in sequence

  * gating with v:
       gate = torch.sigmoid(v)            # (B, L, hidden_dim)
       h = gate * u_conv                  # (B, L, hidden_dim)

  * non-linearity (RELU or GELU) and projection
        h = F.gelu(h)                     # (B, L, hidden_dim)
        h = self.out_proj(h)              # (B, L, d_model)

  * residual connection
         return residual + h
```

EVO perplexity

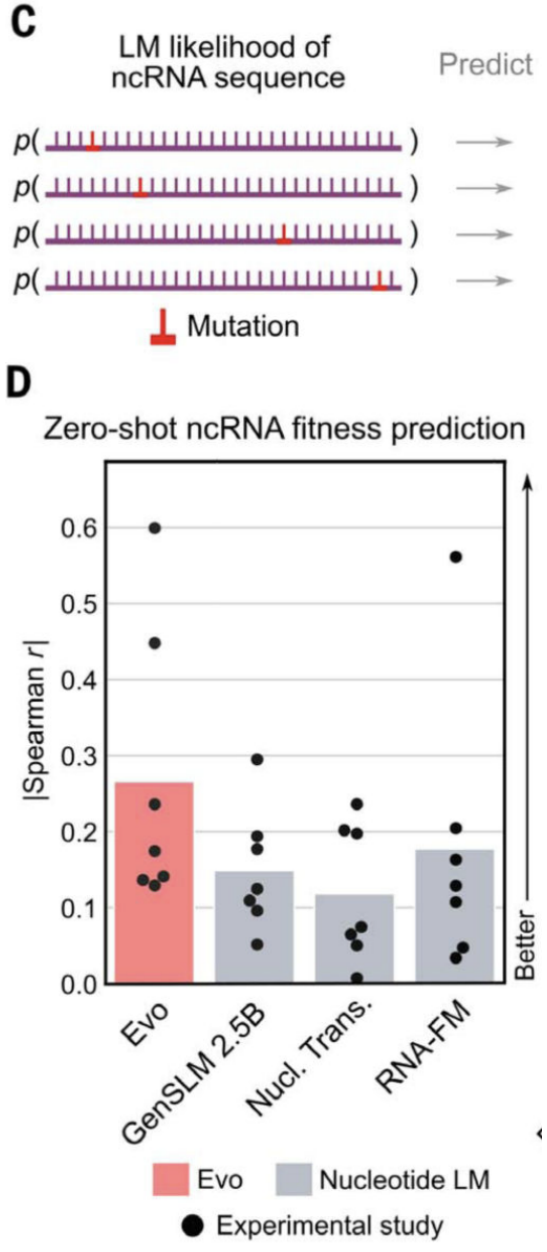


adapted from Figure 1F Nguyen *et al.,* Science, *2024.*

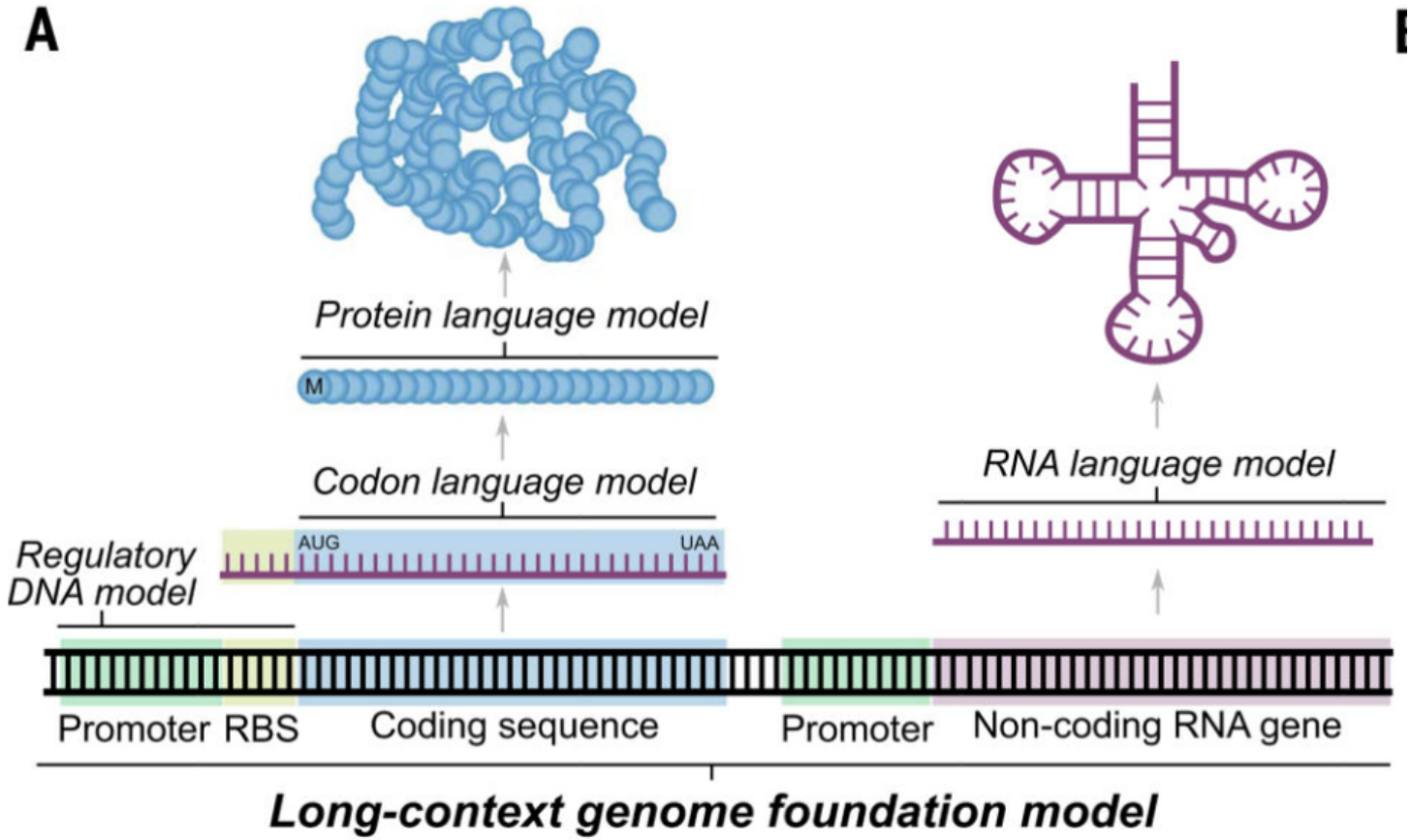
EVO: zero-shot protein fitness prediction



EVO: zero-shot RNA fitness prediction



EVO: a decoder for genomic sequences



how to find the specific:
RNA language,
mRNA language,
regulatory DNA language,...
b4 homework!

nature

Explore content | About the journal | Publish with us

nature > articles > article

Article | Open access | Published: 04 March 2026

Genome modelling and design across all domains of life with Evo 2

Garyk Brixi, Matthew G. Durrant, Jerome Ku, Mohsen Naghipourfar, Michael Poli, Gwanggyu Sun, Greg Brockman, Daniel Chang, Alison Fanton, Gabriel A. Gonzalez, Samuel H. King, David B. Li, Aditi T. Merchant, Eric Nguyen, Chiara Ricci-Tam, David W. Romero, Jonathan C. Schmok, Ali Taghibakhshi, Anton Vorontsov, Brandon Yang, Myra Deng, Liv Gorton, Nam Nguyen, Nicholas K. Wang, ... Brian L. Hie

+ Show authors

Unsupervised Learning

Unsupervised Learning

Variational Inference

Unsupervised Learning

Learning from data without labels

$$P_{data} \approx P_\theta$$

Such that

1. **inference** given x, $P_\theta(x)$
2. **sample**$x \sim P_\theta$
3. **learn** meaningfull things about data

assuption: dataset is a representative sample of P_{data}.
because: we are goig to achieve $P_{data} \approx P_\theta$ by

$$\min_\theta D_{KL}(P_{data}||P_\theta) = \min_\theta \sum_{x_i \, in \, dataset} \log \frac{P_{data}(x_i)}{P_\theta(x_i)}$$

Variational Inference

Minimizing the D_{KL} divergence between P_{data} and P_θ is equivalent to

$$\max_\theta \sum_{x_i \in dataset} \log P_\theta(x_i)$$

How do we calculate

$$\log P_\theta(x)??$$

In unsupervised learning, $P_\theta(x)$ usually utilizes a **latent variable** z

$$P_\theta(x) = \sum_z P_\theta(x, z)dz = \sum_z P_\theta(x \mid z)P(z)dz$$

Variational Inference

What you could do but it is usually **too hard** is

Take samples $z^{(i)} \sim P(z)$ and then calculate

$$\log P_\theta(x) = \log \left[\int_z P_\theta(x \mid z) P(z)\, dz \right] \approx \log \left[\frac{1}{N} \sum_{i=1}^{N} P_\theta(x \mid z^{(i)}) \right]$$

Monte Carlo sampling

Variational Inference

What you do is relying on an auxiliary (variational) distribution $q_\lambda(z)$
easy to manipulate

$$
\begin{aligned}
\log P_\theta(x) &= \log \left[\int_z P_\theta(x\, z)\, dz \right] \\
&= \log \left[\int_z \frac{q_\lambda(z)}{q_\lambda(z)} P_\theta(x\, z)\, dz \right] \\
&\geq \int_z q_\lambda(z) \log \left[\frac{P_\theta(x\, z)}{q_\lambda(z)} \right] dz \qquad \text{Jensen's inequality} \\
&= E_{q_\lambda} \log \left[\frac{P_\theta(x\, z)}{q_\lambda(z)} \right] \\
&= \mathsf{ELBO}(\lambda, \theta, x)
\end{aligned}
$$

ELBO = Evidence Lower Bound

One optimizes the ELBO

Optimizing the ELBO is tractable

$$\text{ELBO}(\lambda, \theta, x) = E_{q_\lambda} \log \left[\frac{P_\theta(x\, z)}{q_\lambda(z)} \right]$$

Using a Monte Carlo sampling $z^{(i)} \sim q_\lambda(z)$,

$$\text{ELBO}(\lambda, \theta, x) \approx \frac{1}{N} \sum_{i=1}^{N} \log \left[\frac{P_\theta(x^{(i)})}{q_\lambda(z^{(i)})} \right]$$