block b3

block b3

block b3

Transformers

block b3

Transformers

Protein Folding-AlphaFold2

Transformers

Transformers → **Natural Languages**

inputs: words in an embedding

inputs are large

inputs are variable in length | no MLPs

parameter sharing

(CNNs RNNs)

connections between words (context)

[self-attention]

Transformers → **Biological sequence languages**

Self-attention

[self-attention] → **Transformers**

Attention Is All You Need 2017

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

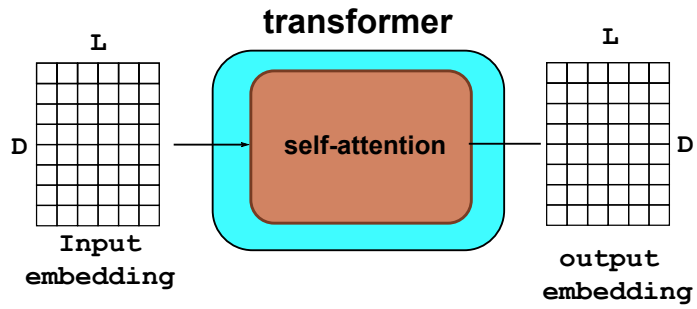
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

3 transformers for natural language translation

self-attention

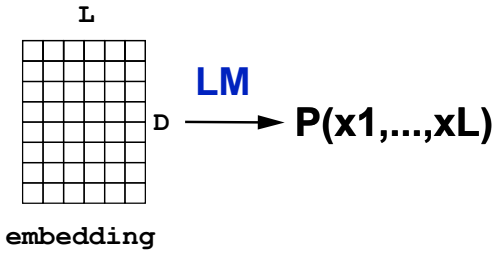


Transformers

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)

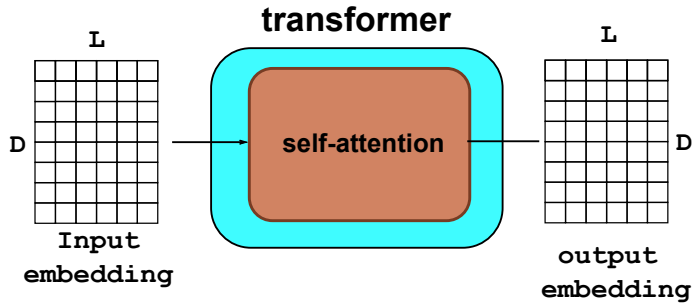
x2 ~ P(x2 | x1)

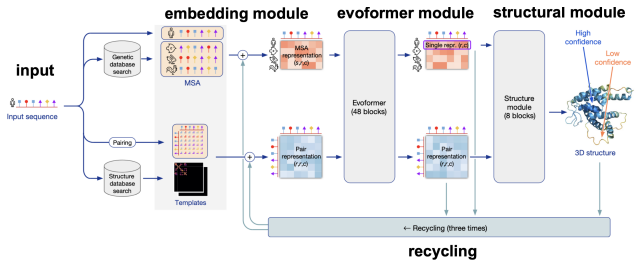
x3 ~ P(x3 | x1,x2)

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

[b3] self-attention

[b3] Transformers

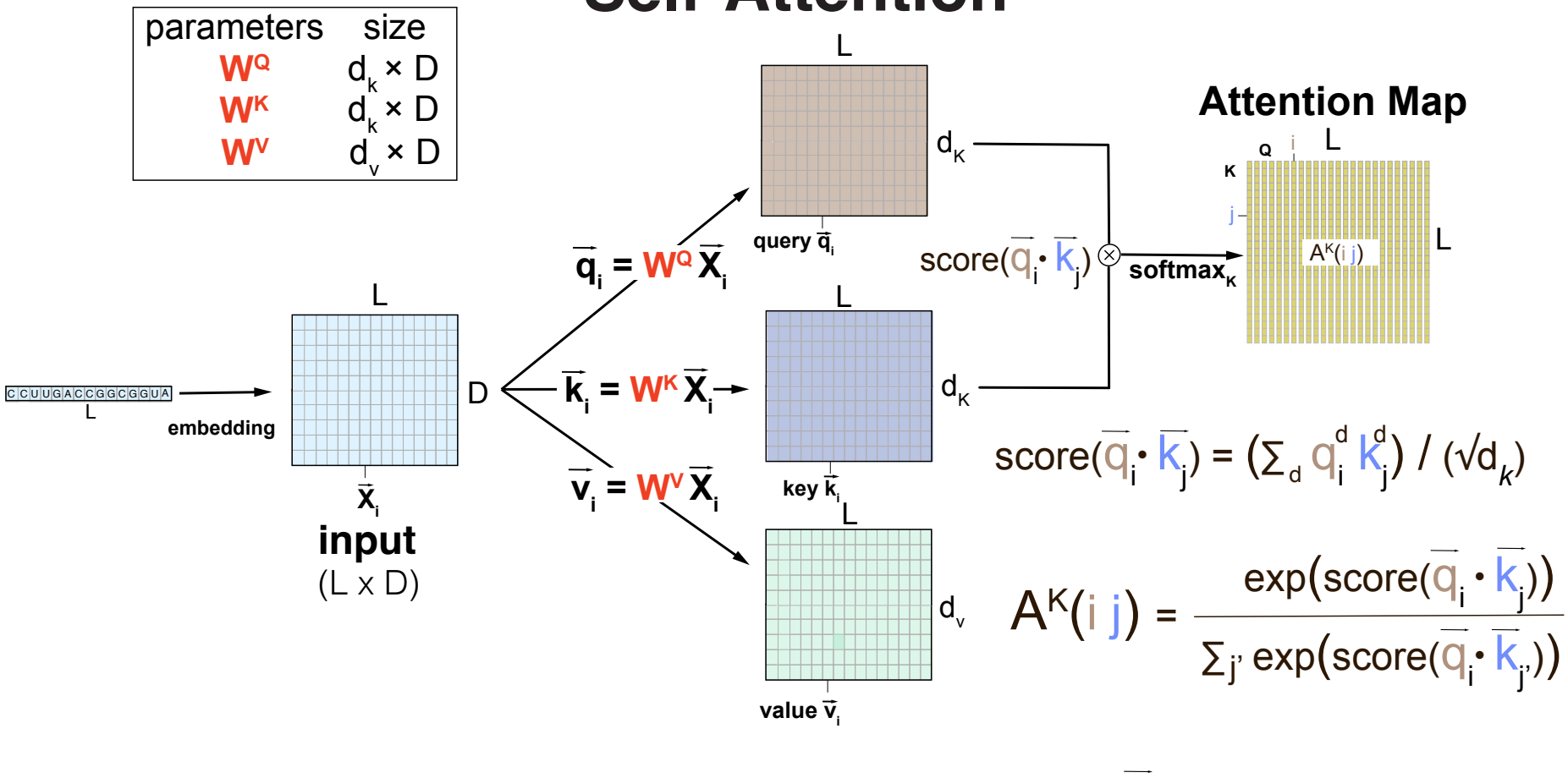




[b3] AlphaFold2

[b4] Language Models (LLMs)

Self-Attention



Dot product self-attention maps

1. **Queries**
   $q[L, d_k] = x[L, D] * W^Q[D, d_k] + b^Q[d_K], \quad q_i(d)$
2. **Keys**
   $k[L, d_k] = x[L, D] * W^K[D, d_k] + b^K[d_k], \quad k_i(d)$
3. **Values**
   $v[L, d_v] = x[L, D] * W^V[D, d_v] + b^V[d_v], \quad v_i(d)$

Linear dot-product score

$$\text{score}(q_i \cdot k_j) = \sum_{d=1}^{d_K} q_i(d)\, k_j(d) \,/\, \sqrt{(d_K)}.$$

Attention between i,j is the softmax of the scores respect to the keys

$$A_{i,j} = \frac{e^{\text{score}(q_i \cdot k_j)}}{\sum_{l=1}^{L} e^{\text{score}(q_i \cdot k_l)}}.$$

Products between vectors $q[D]$ and $k[D]$

1. **scalar product** $(q \cdot k)[1]$

$$q \cdot k = \sum_{d=1}^{D} q_d \, k_d$$

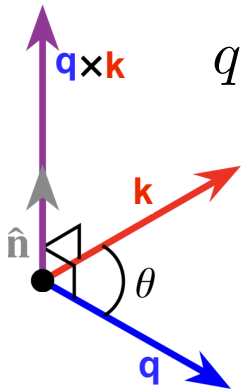
2. **element-wise product** $(q \odot k)[D]$

$$q \odot k = [q_1 \, k_1, ..., q_D \, k_D]$$

3. **outer product** $(q \otimes k)[D, D]$

$$q \otimes k = \begin{bmatrix} q_1 \, k_1 & \dots & q_1 \, k_D \\ \vdots & \vdots & \vdots \\ q_D \, k_1 & \dots & q_D \, k_D \end{bmatrix}$$

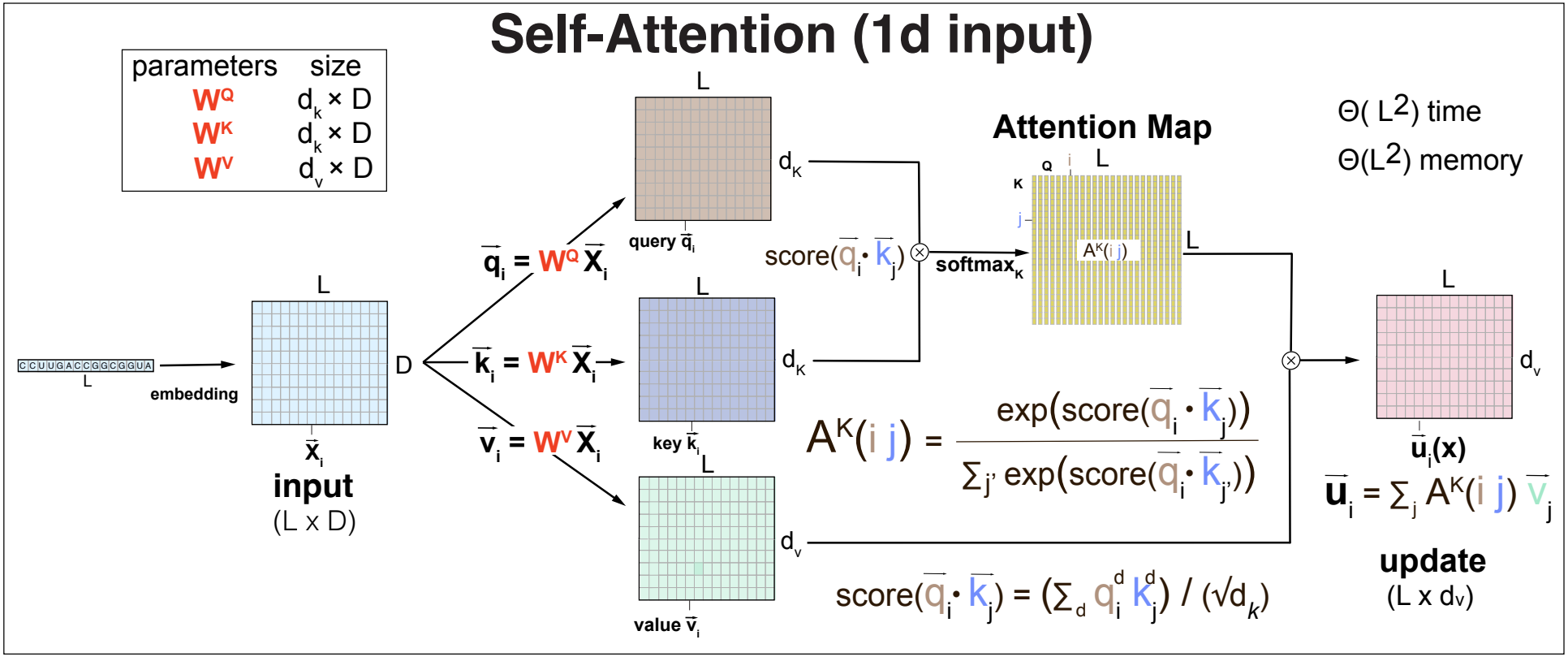
4. **cross product** for $D = 3$, $(q \times k)[3]$



$$q \times k = ||q|| \, ||k|| \sin(\theta) \hat{n}$$

$$= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ q_1 & q_2 & q_3 \\ k_1 & k_2 & k_3 \end{vmatrix}$$

$$= (q_2 \, k_3 - q_3 \, k_2, -(q_1 \, k_3 - q_3 \, k_1), q_1 \, k_2 - q_2 \, k_1)$$

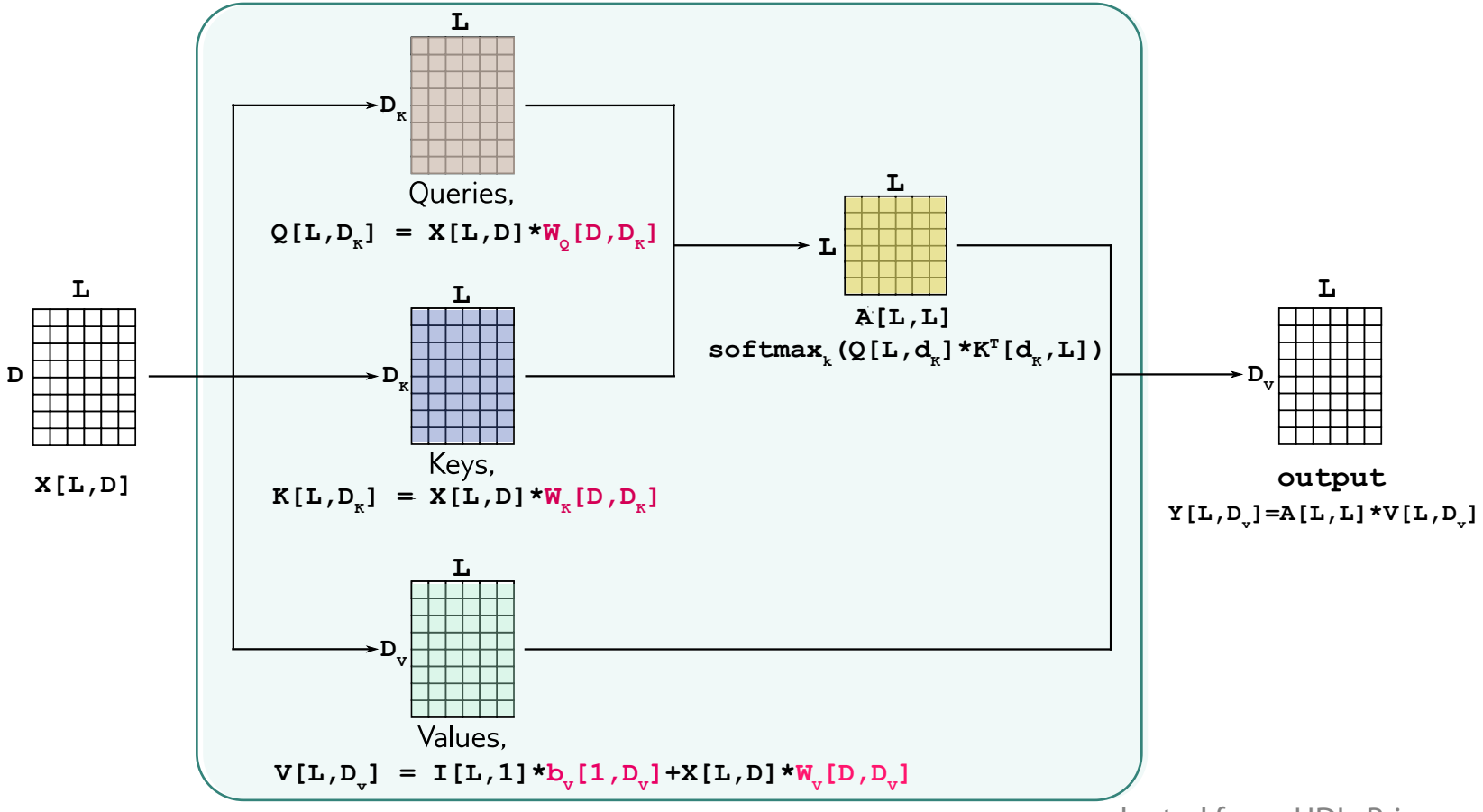

Self-Attention (1d input)
parameters size
W^Q d_k × D
W^K d_k × D
W^V d_v × D
C C U U G A C C G G C G G U A
L
embedding
L
D
x_i
input
(L x D)
q_i = W^Q x_i
k_i = W^K x_i
v_i = W^V x_i
L
query q_i
d_K
L
key k_i
d_K
L
value v_i
d_v
score(q_i • k_j)
softmax_K
Attention Map
Q
K
i
j
L
A^K(i j)
L
A^K(i j) = exp(score(q_i • k_j)) / Σ_j' exp(score(q_i • k_j'))
score(q_i • k_j) = (Σ_d q_i^d k_j^d) / (√d_k)
Θ(L^2) time
Θ(L^2) memory
L
d_v
u_i(x)
u_i = Σ_j A^K(i j) v_j
update
(L x d_v)

Self-Attention (code)

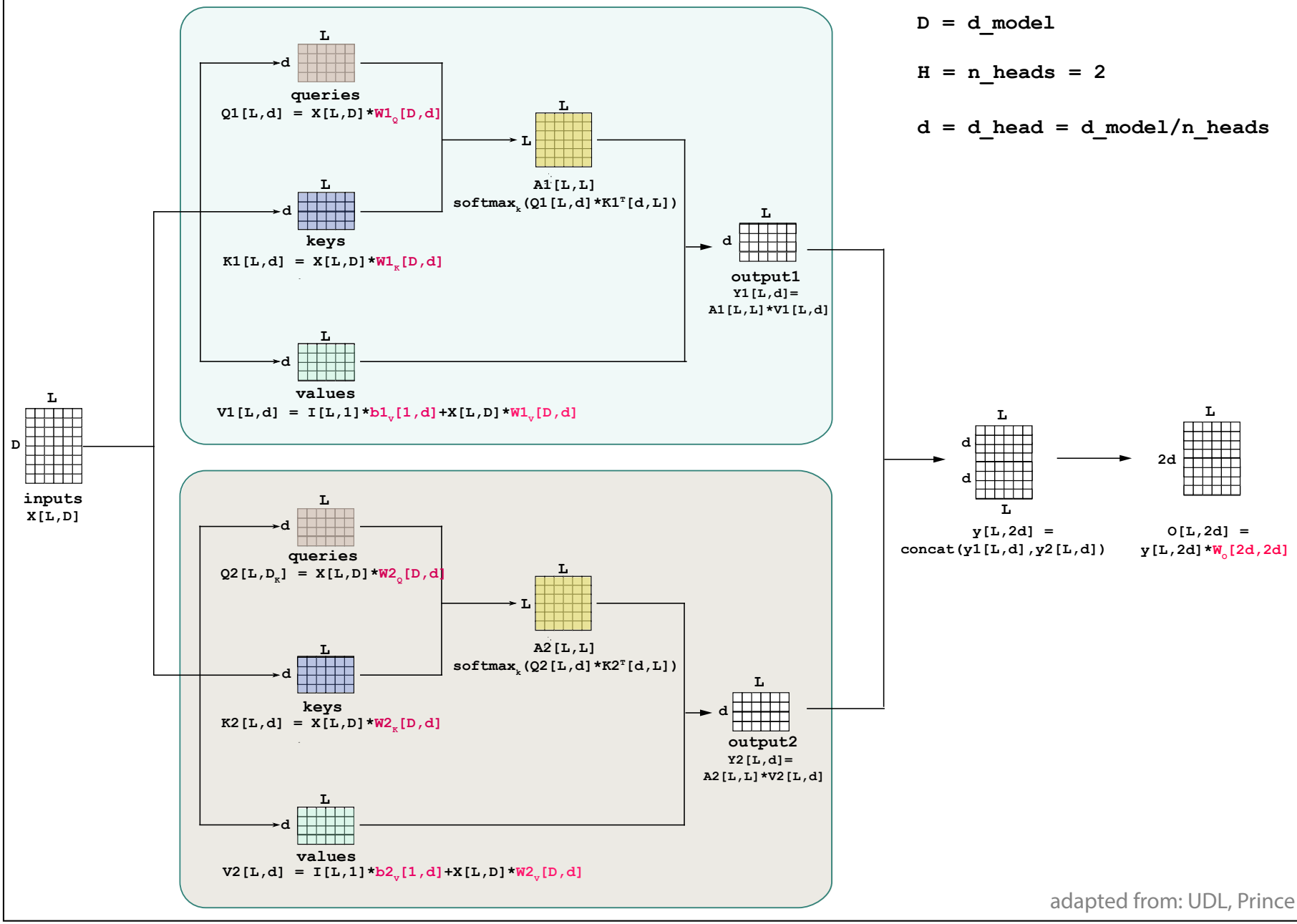
```python
# Simple self-attention
class SelfAttention(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.d_model = d_model

        # uses all dimensions equal to d_model
        self.q_proj = nn.Linear(d_model, d_model)
        self.k_proj = nn.Linear(d_model, d_model)
        self.v_proj = nn.Linear(d_model, d_model)

    def forward(self, x):
        # x: (B, alen, d_model)
        B, L, D = x.shape

        # q[B, L, d_model]
        # k[B, L, d_model]
        # v[B, L, d_model]
        q = self.q_proj(x)
        k = self.k_proj(x)
        v = self.v_proj(x)

        # q                 [B, L,      d_model]
        # k.transpose(-2,-1)[B, d_model, L]
        # attn[B, L, L]
        # v   [B, L, d_model]
        # out[ B, L, d_model]
        # softmax wrt the L dimension comming from the keys
        attn = torch.softmax(q @ k.transpose(-2, -1) / (self.d_model ** 0.5), dim=-1)
        out = attn @ v  # (B, L, d_model)

        return out
```

sequence input = X[B,L, D]

Self Attention (matrix view)

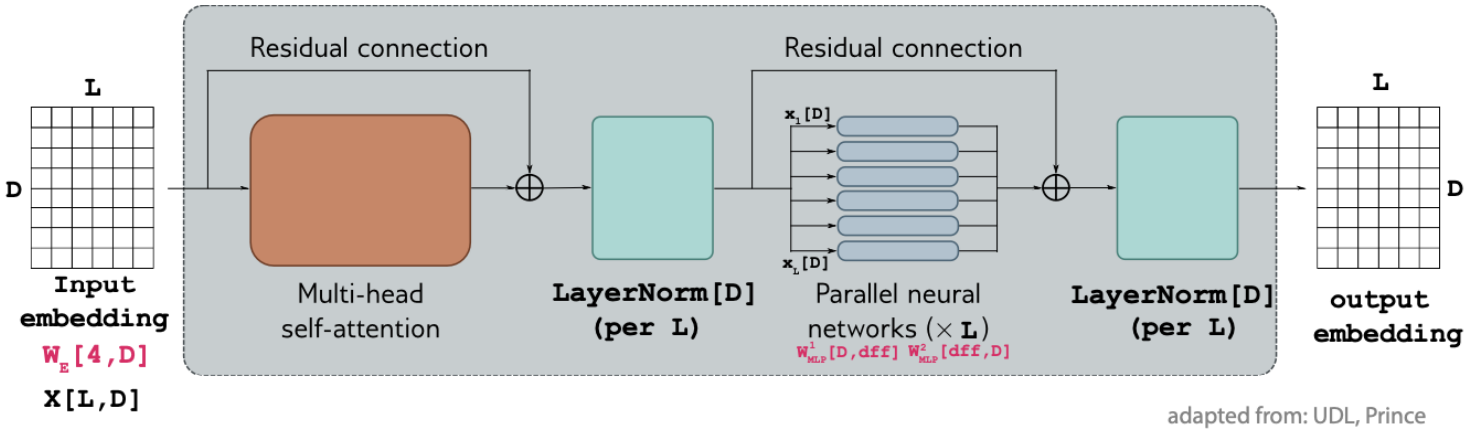


adapted from: UDL, Prince


D = d_model
H = n_heads = 2
d = d_head = d_model/n_heads
inputs
X[L,D]
queries
Q1[L,d] = X[L,D]*W1_Q[D,d]
keys
K1[L,d] = X[L,D]*W1_K[D,d]
values
V1[L,d] = I[L,1]*b1_V[1,d]+X[L,D]*W1_V[D,d]
A1[L,L]
softmax_k(Q1[L,d]*K1^T[d,L])
output1
Y1[L,d]=
A1[L,L]*V1[L,d]
queries
Q2[L,D_K] = X[L,D]*W2_Q[D,d]
keys
K2[L,d] = X[L,D]*W2_K[D,d]
values
V2[L,d] = I[L,1]*b2_V[1,d]+X[L,D]*W2_V[D,d]
A2[L,L]
softmax_k(Q2[L,d]*K2^T[d,L])
output2
Y2[L,d]=
A2[L,L]*V2[L,d]
y[L,2d] =
concat(y1[L,d],y2[L,d])
O[L,2d] =
y[L,2d]*W_O[2d,2d]
adapted from: UDL, Prince

```python
# multi-head sefl-attention
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_model   = d_model
        self.num_heads = num_heads
        self.d_head    = self.d_model // self.num_heads # H = D/n_heads


        # uses all dimensions equal to d_model
        self.q_proj = nn.Linear(d_model, d_model)
        self.k_proj = nn.Linear(d_model, d_model)
        self.v_proj = nn.Linear(d_model, d_model)
        self.o_proj = nn.Linear(d_model, d_model)

    def forward(self, x):
        # x: (B, alen, d_model)
        B, L, D = x.shape

        # H x d_head = d_model
        #
        # q[B, H, L, d_head]
        # k[B, H, L, d_head]
        # v[B, H, L, d_head]
        q = self.q_proj(x).reshape(B, L, self.num_heads, self.d_head).transpose(1, 2)
        k = self.k_proj(x).reshape(B, L, self.num_heads, self.d_head).transpose(1, 2)
        v = self.v_proj(x).reshape(B, L, self.num_heads, self.d_head).transpose(1, 2)

        # q[B, H, L, d_head]
        # k.transpose(-2,-1)[B, H, d_head, L]
        # attn[B, H, L, L]
        # v   [B, H, L, d_head]
        # out [B, H, L, d_head]
        attn = torch.softmax(q @ k.transpose(-2, -1) / (self.d_head ** 0.5), dim=-1)
        out = attn @ v  # (B, heads, L, d_head)

        # concatenate all heads together
        # out.transpose(1, 2)[B, L, H, d_head]
        # out[B, L, H*d_head]
        out = out.transpose(1, 2).reshape(B, L, D)

        # apply one last FC layer
        out = self.o_proj(out)

        return out
```

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
```

Layer Normalization

For an input embedding $x[L, D]$, the LayerNorm, calculates:

1. the **mean**
   $m_i = \frac{1}{D} \sum_{d=1}^{D} x_i^d$

2. the **variance**
   $v_i = \frac{1}{D} \sum_{d=1}^{D} (x_i^d - m_i)^2$

3. Then, the **values get updated** as
   $x_i[D] \longleftarrow \frac{x_i[D] - m_i}{\sqrt{(v_i + \epsilon)}} \gamma + \delta$, γ and δ are trained by the model.

After the normalization layer, each $x_i[D]$ vector has mean δ and standard deviation γ.

Positional encoding

$$
\begin{aligned}
vbins &= [-32, -31, ..., 32] \\
dij &= j - i \\
pij &= LinearLayer(onehot(dij, vbins))
\end{aligned}
$$

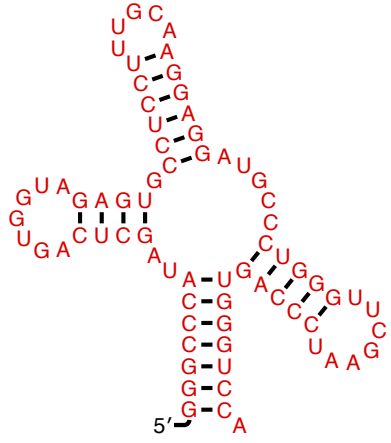
Why Self-attention on multiple sequence alignments (MSAs)?

Why Self-attention on multiple sequence alignments (MSAs)?

RNA secondary structures

structural information

X-ray, NMR, CryoEM



		A	C	G	U
	A	0	0	0	1
5’	C	0	0	10	0
	G	0	5	0	2
	U	4	0	0	0

3’

maximum likelihood training

RNA alignments

evolutionary information

genomes, homology

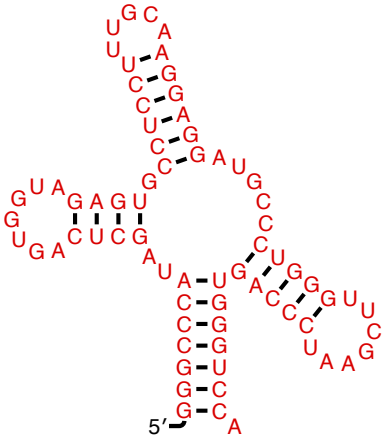
```
GGGCCCAUAGCUCAGU-GGU-AGAGUGCCUCCUUUGCAAGGAGGAUGC-CCUG-GGUUCGAAUCCCAGUGGGUCCA
CAUUAGAUGACUGAA--AG--CAAGUACUGGUCUCUUAAACCAUUUAA-UAGUAAAUUAGCACUUACUUCUAAUGA
GCGCCCGUAGCUCAAUUGGAUAGAGCGUUUGACUACGGAUCAAAAGGU-UAGG-GGUUCGACUCCUCUCGGGCGCG
GGGCCUGUAGCUCAGCUGGUUAGAGCGCACGCCUGAUAAGCGUGAGGU-CGGU-GGUUCGAGUCCACUCAGGCCCA
AGGGGCAUAGUUUAAC-GGU-AGAACAGAGGUCUCCAAAACCUCCGG--UGUG-GGUUCGAUUCCUACUGCCCCUG
AGGGGCGUAGUUCAAUUGGU-AGAGCACCGGUCUCCAAAACCGGGUGU-UGGG-AGUUCGAGUCUCUCCGCCCCUG
GCUGAUAUGGCUCAGUUGGU-AGAGCGCACCCUUGGUAAGGGUGAGGU-CCCC-AGUUCGACUCUGGGUAUCAGCA
UGGGGCGUGGCCAAGU-GGU-AAGGCAGCGGGUUUUGGUCCCGUUACU-CGGA-GGUUCGAAUCCUUCCGUCCCAG
GCCCUUUUAACUCAGU-GGU-AGAGUAAUGCCAUGGUAAGGCAUAAGU-CAUC-GGUUCAAAUCCGAUAAAGGGCU
GCGUCCAUUGUCUAAU-GGAUAGGACAGAGGUCUUCUAAACCUUUGG--UAUA-GGUUCAAAUCCUAUUGGACGCA
GCAUCCAUGGCUGAAU-GGUUAAAGCGCCCAACUCAUAAUUGGUAAAUUUGCG-GGUUCAAUUCCUGCUGGAUGCA
AGUAAGGUCAGCUAA--UU--AAGCUAUCGGGCCCAUACCCCGAAAA--CGUU-GGUUUAAAUCCUUCCCGUACUA
AAGAUAUUAGUAAAAU-CA--AUUACAUAACUUUGUCAAAGUUAAAU--UAUA-GAUCAAUAAUCUAUAUAUCUUA
GUUUCUGUAGUUGAA--UU--ACAACGAUGAUUUUUCAUGUCAUUGG--UCGC-AGUUGAAUGCUGUGUAGAAAUA
GGUCUUAAGGUGAUA------UUCAUGUCGAAUUGCAAAUUCGAAGG--UGUA-GAGAAAU-CUCUACUAAGACUU
ACUCCCUUAGUAUAA--UU--AAUAUAACUGACUUCCAAUUAGUAGA--UUCU-GAAUAAAC-CCAGAAGAGAGUA
UAGAUUGAAGCCAGUA-AU--AGGGUAUUUAGCUGUUAACUAAAUUUU-CGUA-GGUUUAAUUCCUGCCAAUCUAG
CACUAUGAAGCUA--------AGAGCGUUAACCUUUUAAGUUAAAGUU-AGAG-ACCUUAAA-AUCUCCAUAGUGA
GUUAAUGUAGCUUAAUAAC--AAAGCAAAGCACUGAAAAUGCUUAGA--UGGA-U-AAUUGU--AUCCCAUAAACA
ACUUUUAUAGGAUAAU-AGU-AAUCCAUUGGUCUUAGGAACCAAAAA--CCUU-GGUGCAAAUCCAAAUAAAAGUA
GGGCCUAUAGCUCAGCUGGUUAGAGUGCACCCCUGAUAAGGGUGAGGU-CACA-AGUUCAAGUCUUGUUAGGCCCA
```

Why Self-attention on multiple sequence alignments (MSAs)?

RNA secondary structures

structural information

X-ray, NMR, CryoEM



		A	C	G	U
	A	0	0	0	1
5’	C	0	0	10	0
	G	0	5	0	2
	U	4	0	0	0

3’

maximum likelihood training

RNA alignments

evolutionary information

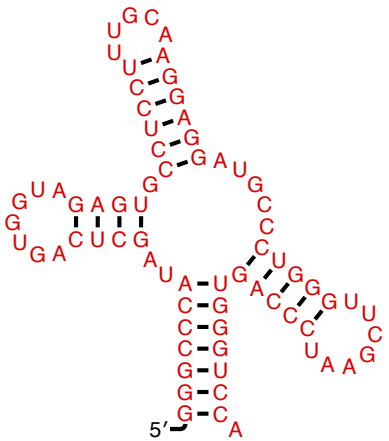
genomes, homology

Why Self-attention on multiple sequence alignments (MSAs)?

RNA secondary structures

structural information

X-ray, NMR, CryoEM



		A	C	G	U
	A	0	0	0	1
5’	C	0	0	10	0
	G	0	5	0	2
	U	4	0	0	0

3’

maximum likelihood training

RNA alignments

evolutionary information

genomes, homology

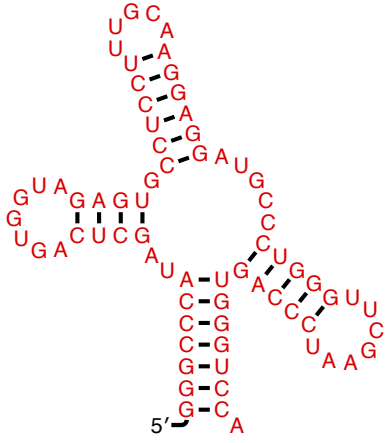
```
GG..CCA
CA..UGA
GC..GCG
GG..CCA
AG..CUG
AG..CUG
GC..GCA
UG..CAG
GC..GCU
GC..GCA
GC..GCA
AG..CUA
AA..UUA
GU..AUA
GG..CUU
AC..GUA
UA..UAG
CA..UGA
```

Why Self-attention on multiple sequence alignments (MSAs)?

RNA secondary structures

structural information

X-ray, NMR, CryoEM



		A	C	G	U
	A	0	0	0	1
5'	C	0	0	10	0
	G	0	5	0	2
	U	4	0	0	0

3'

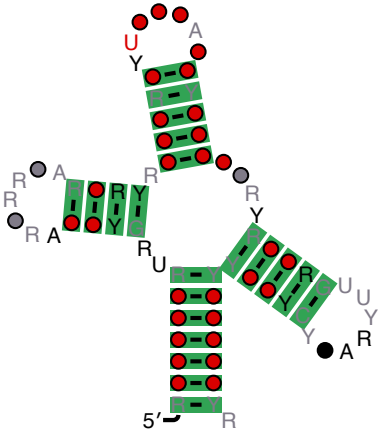
maximum likelihood training

RNA alignments

evolutionary information

genomes, homology

```
GGGCCCAUAGCUCAGU-GGU-AGAGUGCCUCCUUUGCAAGGAGGAUGC-CCUG-GGUUCGAAUCCCAGUGGGUCCA
CAUUAGAUGACUGAA--AG--CAAGUACUGGUCUCUUAAACCAUUUAA-UAGUAAAUUAGCACUUACUUCUAAUGA
GCGCCCCGUAGCUCAAUUGGAUAGAGCGUUUGACUACGGAUCAAAAGGU-UAGG-GGUUCGACUCCUCUCGGGCGCG
GGGCCUGUAGCUCAGCUGGUUAGAGCGCACGCCUGAUAAGCGUGAGGU-CGGU-GGUUCGAGUCCACUCAGGCCCA
AGGGGCAUAGUUUAAC-GGU-AGAACAGAGGUCUCCAAAACCUCCGG--UGUG-GGUUCGAUUCCUACUGCCCCUG
AGGGGCGUAGUUCAAUUGGU-AGAGCACCGGUCUCCAAAACCGGGUGU-UGGG-AGUUCGAGUCUCUCCGCCCCUG
GCUGAUAUGGCUCAGUUGGU-AGAGCGCACCCUUGGUAAGGGUGAGGU-CCCC-AGUUCGACUCUGGGUAUCAGCA
UGGGGCGUGGCCAAGU-GGU-AAGGCAGCGGGUUUUGGUCCCGUUACU-CGGA-GGUUCGAAUCCUUCCGUCCCAG
GCCCUUUUAACUCAGU-GGU-AGAGUAAUGCCAUGGUAAGGCAUAAGU-CAUC-GGUUCAAAUCCGAUAAAGGGCU
GCGUCCAUUGUCUAAU-GGAUAGGACAGAGGUCUUCUAAACCUUUGG--UAUA-GGUUCAAAUCCUAUUGGACGCA
GCAUCCAUGGCUGAAU-GGUUAAAGCGCCCAACUCAUAAUUGGUAAAUUUGCG-GGUUCAAUUCCUGCUGGAUGCA
AGUAAGGUCAGCUAA--UU--AAGCUAUCGGGCCCAUACCCCGAAAA--CGUU-GGUUUAAAUCCUUCCCGUACUA
AAGAUAUUAGUAAAAU-CA--AUUACAUAACUUUGUCAAAGUUAAAU--UAUA-GAUCAAUAAUCUAUAUAUCUUA
GUUUCUGUAGUUGAA--UU--ACAACGAUGAUUUUUCAUGUCAUUGG--UCGC-AGUUGAAUGCUGUGUAGAAAUA
GGUCUUAAGGUGAUA------UUCAUGUCGAAUUGCAAAUUCGAAGG--UGUA-GAGAAAU-CUCUACUAAGACUU
ACUCCCUUAGUAUAA--UU--AAUAUAACUGACUUCCAAUUAGUAGA--UUCU-GAAUAAAC-CCAGAAGAGAGUA
UAGAUUGAAGCCAGUA-AU--AGGGUAUUUAGCUGUUAACUAAAUUUU-CGUA-GGUUUAAUUCCUGCCAAUCUAG
CACUAUGAAGCUA--------AGAGCGUUAACCUUUUAAGUUAAAGUU-AGAG-ACCUUAAA-AUCUCCAUAGUGA
GUUAAUGUAGCUUAAUAAC--AAAGCAAAGCACUGAAAAUGCUUAGA--UGGA-U-AAUUGU--AUCCCAUAAACA
ACUUUUAUAGGAUAAU-AGU-AAUCCAUUGGUCUUAGGAACCAAAAA--CCUU-GGUGCAAAUCCAAAUAAAAGUA
GGGCCUAUAGCUCAGCUGGUUAGAGUGCACCCCUGAUAAGGGUGAGGU-CACA-AGUUCAAGUCUUGUUAGGCCCA
```



MSA attention by row/column

parameters	size
W^Q	D x d_k
W^K	D x d_k
W^V	D x d_v

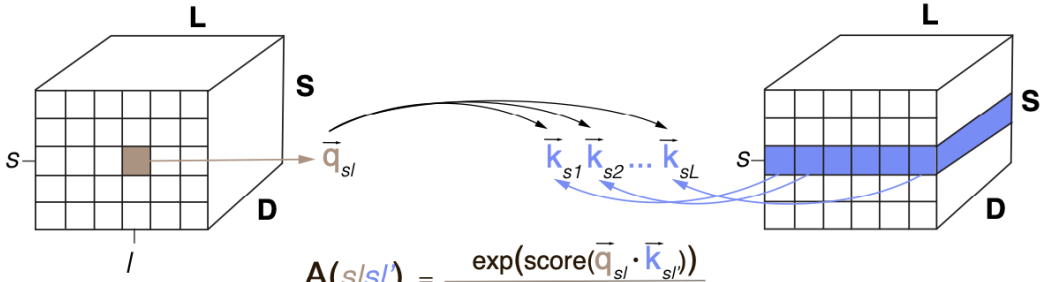
$\vec{x}_{sl}$ **input**, size = D

$\vec{q}_{sl} = \vec{x}_{sl} W^Q$ **query**, size = d_k

$\vec{k}_{sl} = \vec{x}_{sl} W^K$ **key**, size = d_k

$\vec{v}_{sl} = \vec{x}_{sl} W^V$ **value**, size = d_v

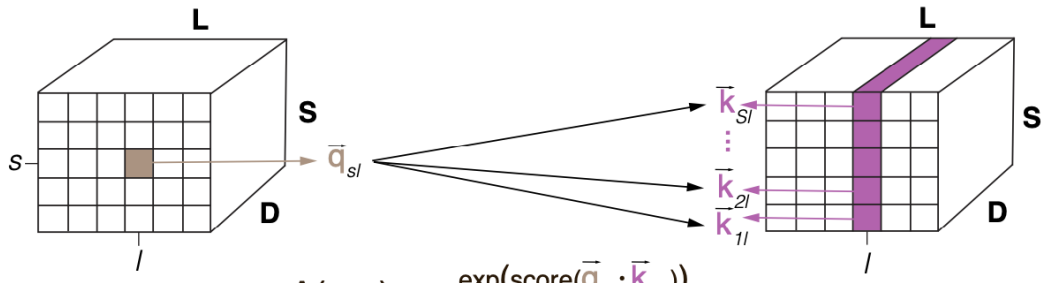
self-attention by row



$$A(s l s l') = \frac{\exp(\text{score}(\vec{q}_{sl} \cdot \vec{k}_{sl'}))}{\sum_m \exp(\text{score}(\vec{q}_{sl} \cdot \vec{k}_{sm}))}$$

$$\vec{R}_{sl} = \sum_{l'} A(s l s l')\, \vec{v}_{sl'} \qquad \Theta(S \times L^2)$$

self-attention by column



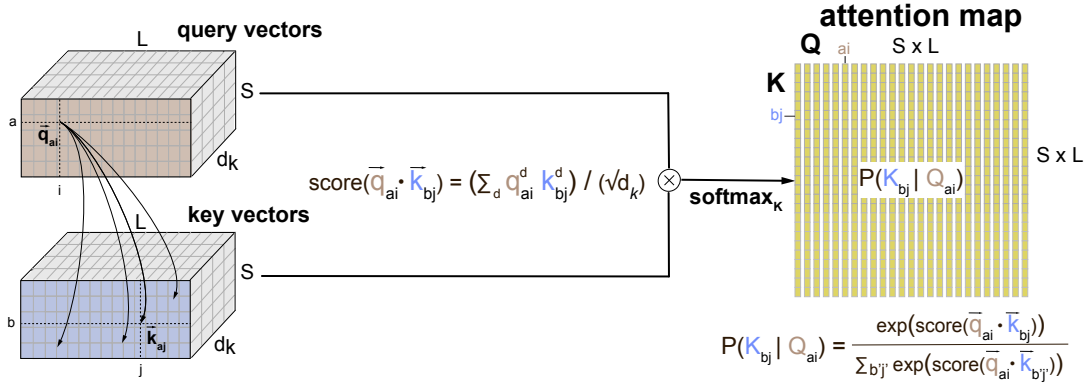
$$A(s l s' l) = \frac{\exp(\text{score}(\vec{q}_{sl} \cdot \vec{k}_{s'l}))}{\sum_t \exp(\text{score}(\vec{q}_{sl} \cdot \vec{k}_{tl}))}$$

$$\vec{C}_{sl} = \sum_{s'} A(s l s' l)\, \vec{v}_{s'l} \qquad \Theta(L \times S^2)$$

```python
# Row attention =
# attention across residues for each sequence (MSA row)
# thus: row[S] is fixed,  L and D move
class RowAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, num_heads)

    def forward(self, x):
        # x: (B, S, L, d_model)
        B, S, L, D = x.shape
        x = x.reshape(B * S, L, D)
        x = self.attn(x)
        return x.reshape(B, S, L, D)

# Column attention =
# attention across MSA sequences at each residue (MSA column)
# L is fixed, we move on S and D.
# Thus we need to put S, D as the last two dimentsion
class ColumnAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, num_heads)

    def forward(self, x):
        # x: (B, S, alen, d_model)
        B, S, L, D = x.shape
        x = x.transpose(1, 2).reshape(B * L, S, D)
        x = self.attn(x)
        return x.reshape(B, L, S, D).transpose(1, 2)
```

Alignment Self-Attention

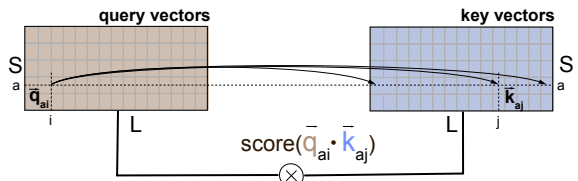
parameters	size
W^Q	D x d_k
W^K	D x d_k
W^V	D x d_v



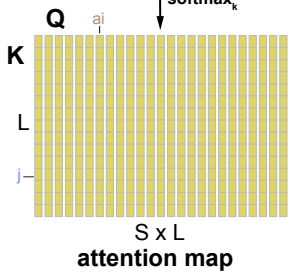
$$\text{score}(\vec{q}_{ai} \cdot \vec{k}_{bj}) = (\sum_d q^d_{ai}\ k^d_{bj}) \,/\, (\sqrt{d_k})$$

$$P(K_{bj} \mid Q_{ai}) = \frac{\exp(\text{score}(\vec{q}_{ai} \cdot \vec{k}_{bj}))}{\sum_{b'j'} \exp(\text{score}(\vec{q}_{ai} \cdot \vec{k}_{b'j'}))}$$

Attention by row

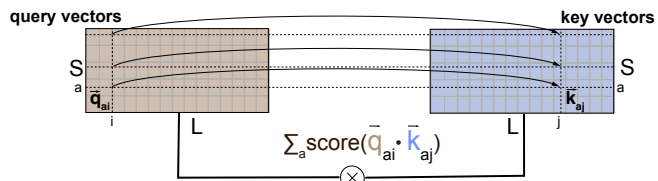


$$\text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj})$$

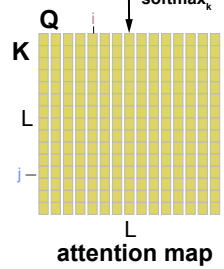


$$P(K_j \mid Q_{ai}) = \frac{\exp(\text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj}))}{\sum_{j'} \exp(\text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj'}))}$$

tied row Attention

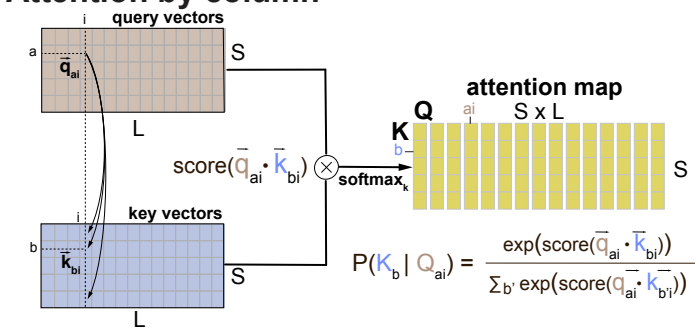


$$\sum_a \text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj})$$



$$P(K_j \mid Q_i) = \frac{\exp(\sum_a \text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj}))}{\sum_{j'} \exp(\sum_a \text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj'}))}$$

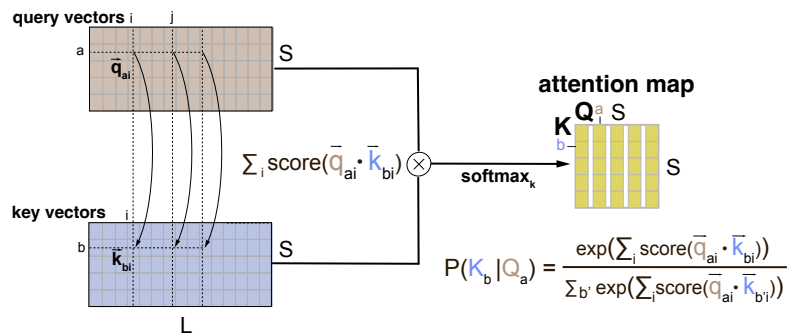
Attention by column



$$\text{score}(\vec{q}_{ai} \cdot \vec{k}_{bi})$$

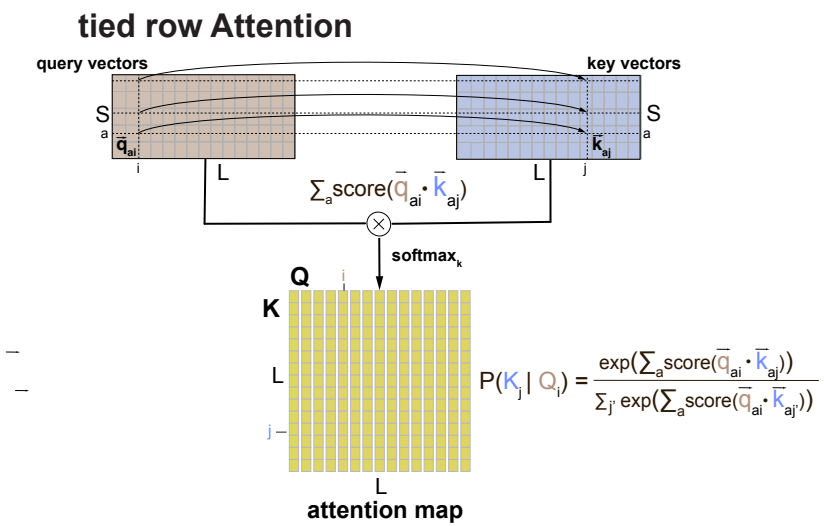
$$P(K_b \mid Q_{ai}) = \frac{\exp(\text{score}(\vec{q}_{ai} \cdot \vec{k}_{bi}))}{\sum_{b'} \exp(\text{score}(\vec{q}_{ai} \cdot \vec{k}_{b'i}))}$$

tied column Attention



$$\sum_i \text{score}(\vec{q}_{ai} \cdot \vec{k}_{bi})$$

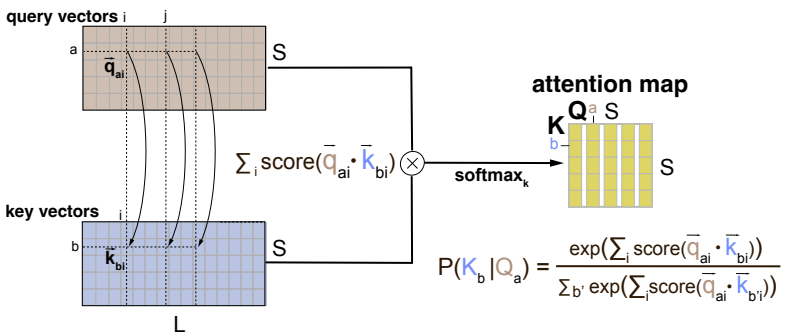
$$P(K_b \mid Q_a) = \frac{\exp(\sum_i \text{score}(\vec{q}_{ai} \cdot \vec{k}_{bi}))}{\sum_{b'} \exp(\sum_i \text{score}(\vec{q}_{ai} \cdot \vec{k}_{b'i}))}$$



Tied row attention

```python
class TiedRowAttention(nn.Module):
    def __init__(self, d_model, num_heads, tie_mode="mean"):
        super().__init__()
        self.num_heads = num_heads
        self.head_dim  = d_model // num_heads
        self.tie_mode = tie_mode   # "mean" or "sum"

        # shared projections across all rows
        self.q_proj = nn.Linear(d_model, d_model)
        self.k_proj = nn.Linear(d_model, d_model)
        self.v_proj = nn.Linear(d_model, d_model)
        self.o_proj = nn.Linear(d_model, d_model)

    def forward(self, x):
        # x: (B, S, L, D)
        B, S, L, D = x.shape

        # flatten rows so projections are tied
        x_flat = x.reshape(B*S, L, D)

        # queries, keys, values [B*S, L, D]
        Q = self.q_proj(x_flat)
        K = self.k_proj(x_flat)
        V = self.v_proj(x_flat)

        # multi-head split
        # queries, keys, values [B*S, L, H, DH] (reshape)
        # queries, keys, values [B*S, H, L, DH] (transpose 1,2)
        Q = Q.reshape(B*S, L, self.num_heads, self.head_dim).transpose(1, 2)
        K = K.reshape(B*S, L, self.num_heads, self.head_dim).transpose(1, 2)
        V = V.reshape(B*S, L, self.num_heads, self.head_dim).transpose(1, 2)

        # per-row attention scores
        # R_scores[B*S, H, L, L] = Q[B*S, H, L, DH]*KT[B*S, H, DH, L]
        R_scores = torch.matmul(Q, K.transpose(-2, -1)) / (self.head_dim ** 0.5)
        # R_scores[B. S, H, L, L]
        R_scores = R_scores.reshape(B, S, self.num_heads, L, L)

        # tie at the attention-weight level
        # TR_scores[B, 1, H, L, L]
        TR_scores = R_scores.mean(dim=1, keepdim=True)  # (B, 1, H, L, L)

        #softmax AFTER tying
        # TR_attn[B, 1, H, L, L]
        TR_attn = torch.softmax(TR_scores, dim=-1)    # (B, 1, H, L, L)

        # expand tied attention to all rows
        # TR_attn[B, S, H, L, L]
        TR_attn = TR_attn.repeat(1, S, 1, 1, 1)       # (B, S, H, L, L)

        # print TR_attn shape
        print("\nTied-row attention (TR_attn) maps dimensions [B, S, H, L, L]", TR_attn.shape)

        # reshape V back to [B, S, H, L, DH] (from [B*S, H, L, DH])
        V = V.reshape(B, S, self.num_heads, L, self.head_dim)

        # out[B, S, H, L, DH] = TR_attn[B, S, H, L, L] * V[B, S, H, L, DH]
        out = torch.matmul(TR_attn, V)     # (B, S, H, L, DH)

        # merge heads
        # out[B, S, H, L, DH]
        # out[B, S, L, H, DH] (transposed 2,3)
        # out[B, S, L, D]     (merged all heads)
        out = out.transpose(2, 3).reshape(B, S, L, D)

        # project
        out = self.o_proj(out)

        return out
```

tied column Attention

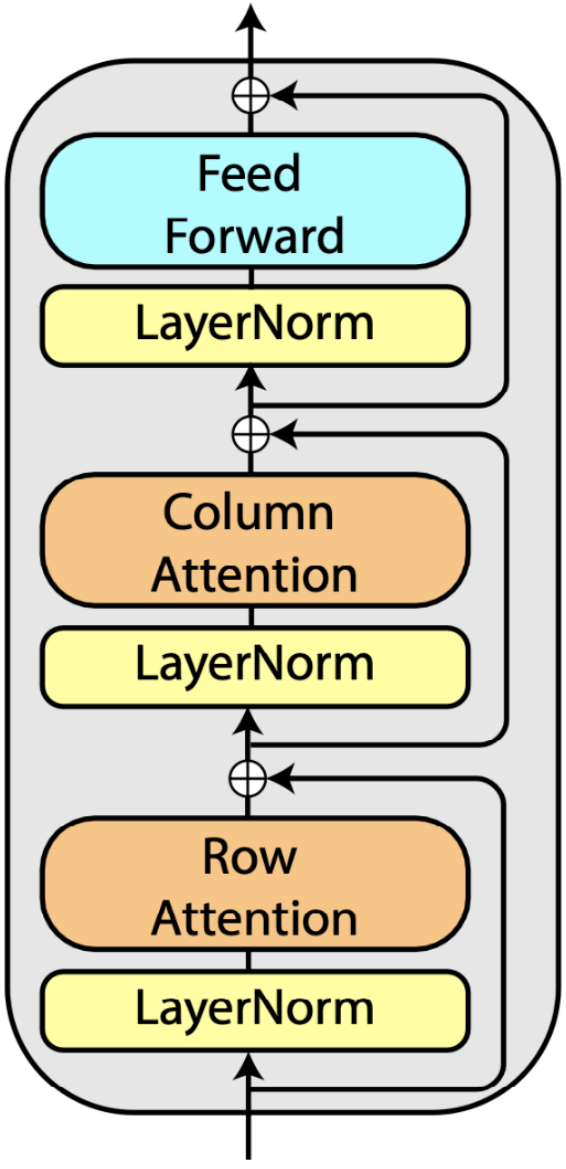


Tied column attention

```python
class TiedColumnAttention(nn.Module):
    def __init__(self, d_model, num_heads, tie_mode="mean"):
        super().__init__()
        self.num_heads = num_heads
        self.head_dim  = d_model // num_heads
        self.tie_mode = tie_mode   # "mean" or "sum"

        # shared projections across all rows
        self.q_proj = nn.Linear(d_model, d_model)
        self.k_proj = nn.Linear(d_model, d_model)
        self.v_proj = nn.Linear(d_model, d_model)
        self.o_proj = nn.Linear(d_model, d_model)

    def forward(self, x):
        # x: (B, S, L, D)
        B, S, L, D = x.shape

        # transpose
        # x[B, S, L, D] -> x[B, L, S, D]
        x = x.transpose(1,2)

        # flatten rows so projections are tied
        x_flat = x.reshape(B*L, S, D)

        # queries, keys, values [B*L, S, D]
        Q = self.q_proj(x_flat)
        K = self.k_proj(x_flat)
        V = self.v_proj(x_flat)

        # multi-head split
        # queries, keys, values [B*L, S, H, DH] (reshape)
        # queries, keys, values [B*L, H, S, DH] (transpose 1,2)
        Q = Q.reshape(B*L, S, self.num_heads, self.head_dim).transpose(1, 2)
        K = K.reshape(B*L, S, self.num_heads, self.head_dim).transpose(1, 2)
        V = V.reshape(B*L, S, self.num_heads, self.head_dim).transpose(1, 2)

        # per-column attention scores
        # C_scores[B*L, H, S, S] = Q[B*L, H, S, DH]*KT[B*L, H, DH, S]
        C_scores = torch.matmul(Q, K.transpose(-2, -1)) / (self.head_dim ** 0.5)
        # scores[B, L, H, S, S]
        C_scores = C_scores.reshape(B, L, self.num_heads, S, S)

        # tie at the attention-weight level
        # TC_scores[B, 1, H, S, S]
        TC_scores = C_scores.mean(dim=1, keepdim=True)  # (B, 1, H, S, S)

        #softmax AFTER tying
        # TC_attn[B, 1, H, S, S]
        TC_attn = torch.softmax(TC_scores, dim=-1)    # (B, 1, H, S, S)

        # expand tied attention to all rows
        # TC_attn[B, L, H, S, S]
        TC_attn = TC_attn.repeat(1, L, 1, 1, 1)         # (B, L, H, S, S)

        # print TC_attn shape
        print("\ntied-column attention (TC_attn) maps dimensions [B, L, H, S, S]", TC_attn.shape)

        # reshape V to [B, L, H, S, DH] (from [B*L, H, S, DH])
        V = V.reshape(B, L, self.num_heads, S, self.head_dim)

        # out[B, L, H, S, DH] = TC_attn[B, L, H, S, S] * V[B, L, H, S, DH]
        out = torch.matmul(TC_attn, V)     # (B, L, H, S, DH)

        # merge heads
        # out[B, L, H, S, DH]
        # out[B, L, S, H, DH] (transposed 2,3)
        # out[B, L, S, D]     (merged all heads)
        out = out.transpose(2, 3).reshape(B, L, S, D)

        # out[B, S, L, D] <- out[B, L, S, D]
        out = out.transpose(1, 2)

        # project
        out = self.o_proj(out)

        return out
```

MSA self-attention

MSA Transfomer, Rao *et al*, 2021



MSA transformer

```python
# MSA Transformer block = row attention → column attention → MLP
class MSATransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff):
        super().__init__()

        self.row_attn = RowAttention   (d_model, num_heads)
        self.col_attn = ColumnAttention(d_model, num_heads)

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

        self.norm_row = nn.LayerNorm(d_model)
        self.norm_col = nn.LayerNorm(d_model)
        self.norm_ff  = nn.LayerNorm(d_model)

    def forward(self, x):
        x = x + self.row_attn(self.norm_row(x))
        x = x + self.col_attn(self.norm_col(x))
        x = x + self.mlp(self.norm_ff(x))
        return x
```

AlphaFold2

Article

Highly accurate protein structure prediction with AlphaFold

2024 Nobel in Chemistry
Jumper, Hassabis, Baker

https://doi.org/10.1038/s41586-021-03819-2

Received: 11 May 2021

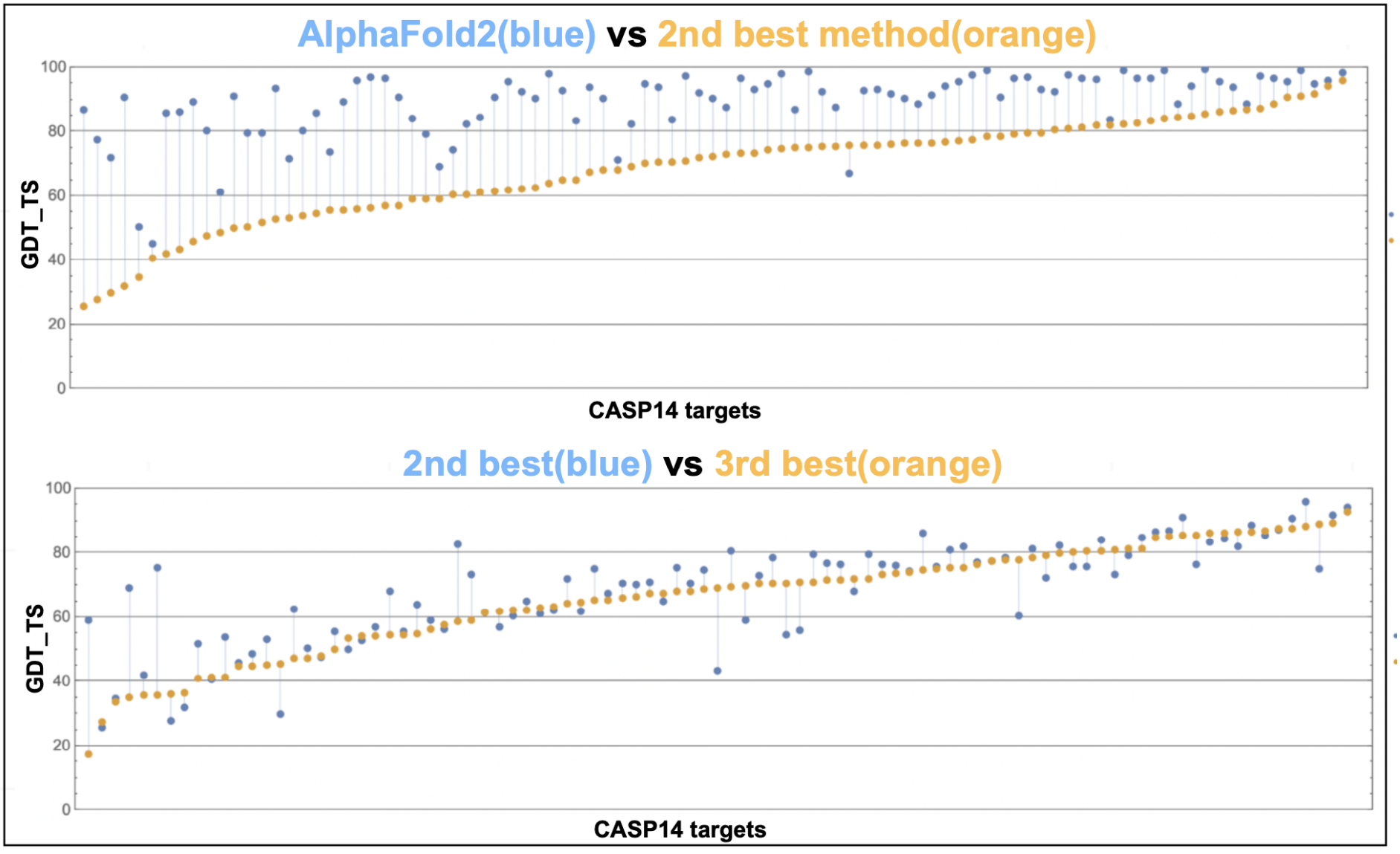
Accepted: 12 July 2021

Published online: 15 July 2021

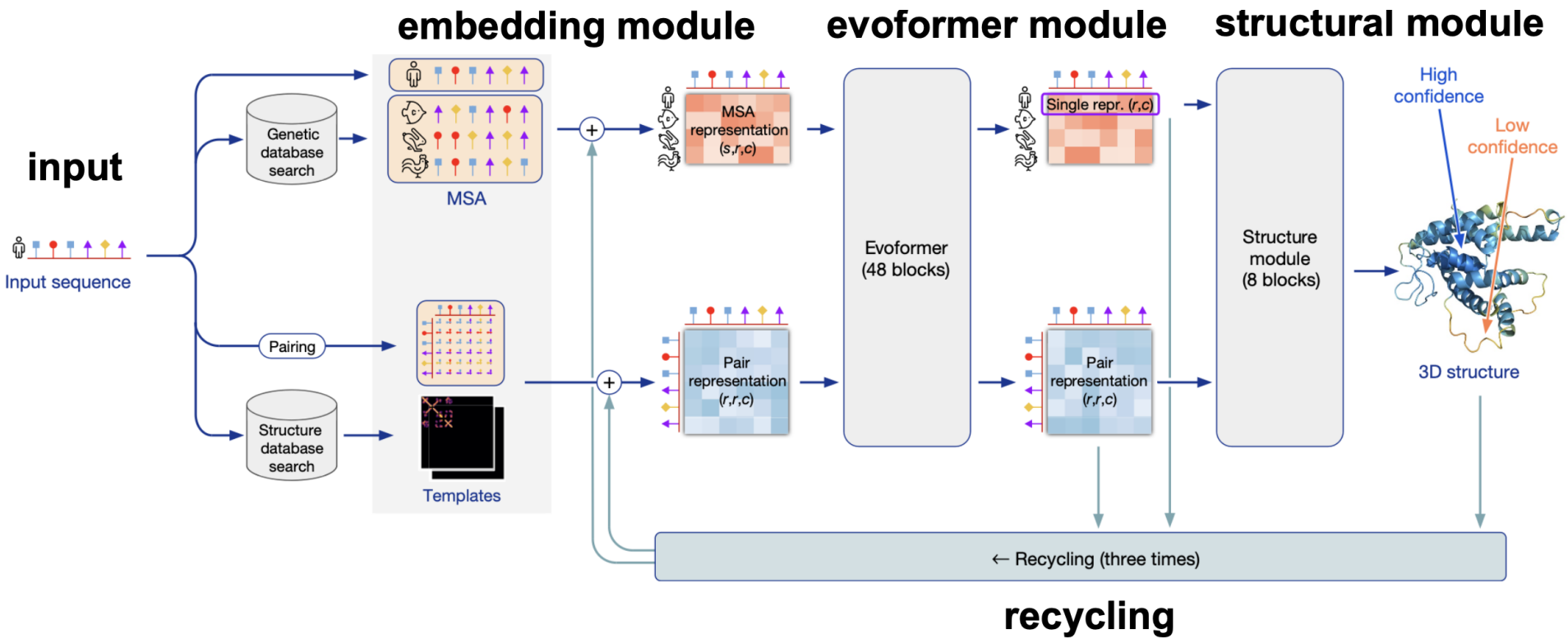
Open access

Check for updates

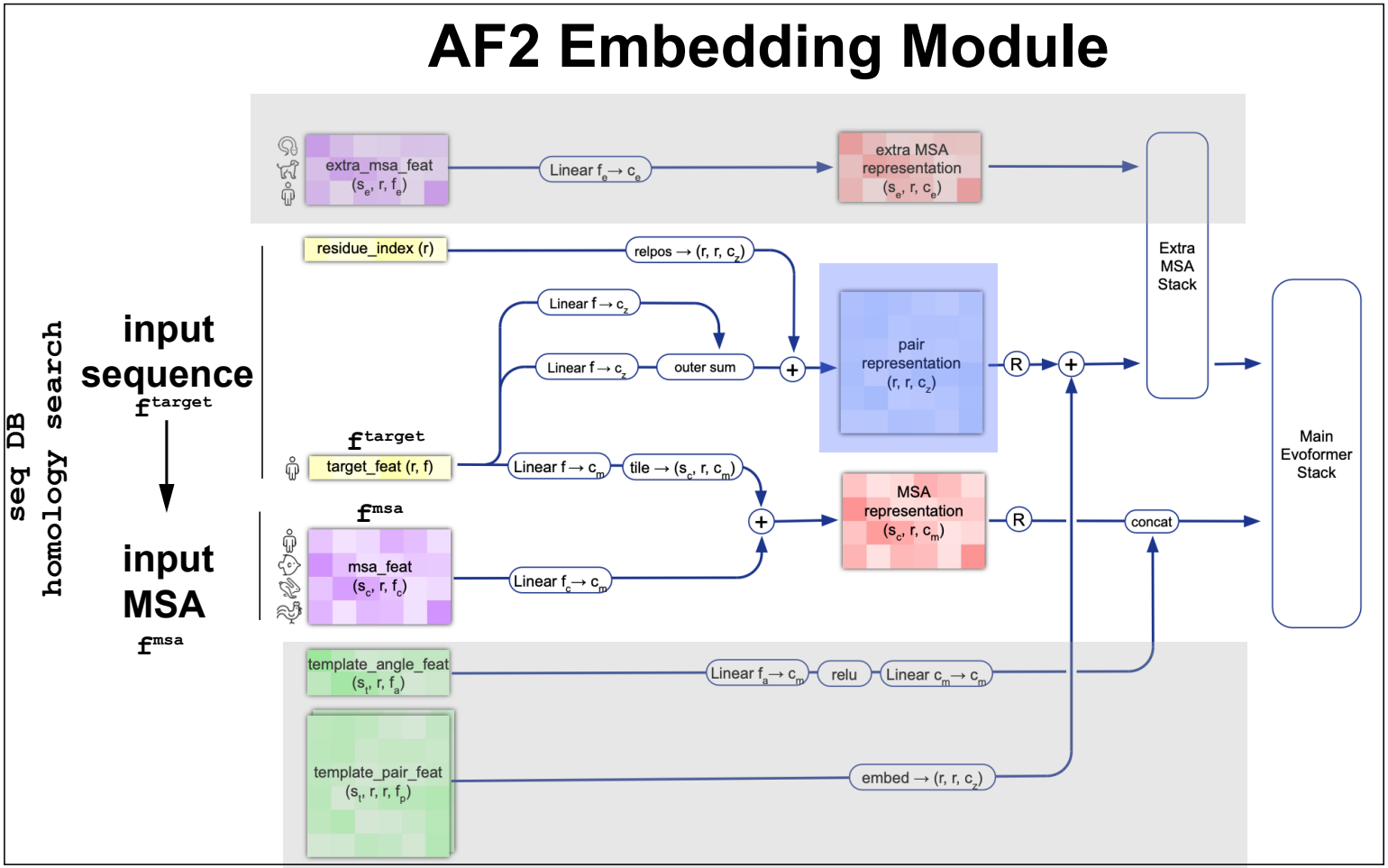
John Jumper[1,4]✉, Richard Evans[1,4], Alexander Pritzel[1,4], Tim Green[1,4], Michael Figurnov[1,4], Olaf Ronneberger[1,4], Kathryn Tunyasuvunakool[1,4], Russ Bates[1,4], Augustin Žídek[1,4], Anna Potapenko[1,4], Alex Bridgland[1,4], Clemens Meyer[1,4], Simon A. A. Kohl[1,4], Andrew J. Ballard[1,4], Andrew Cowie[1,4], Bernardino Romera-Paredes[1,4], Stanislav Nikolov[1,4], Rishub Jain[1,4], Jonas Adler[1], Trevor Back[1], Stig Petersen[1], David Reiman[1], Ellen Clancy[1], Michal Zielinski[1], Martin Steinegger[2,3], Michalina Pacholska[1], Tamas Berghammer[1], Sebastian Bodenstein[1], David Silver[1], Oriol Vinyals[1], Andrew W. Senior[1], Koray Kavukcuoglu[1], Pushmeet Kohli[1] & Demis Hassabis[1,4]✉


AlphaFold2(blue) vs 2nd best method(orange)
GDT_TS
100
80
60
40
20
0
CASP14 targets
2nd best(blue) vs 3rd best(orange)
GDT_TS
100
80
60
40
20
0
CASP14 targets

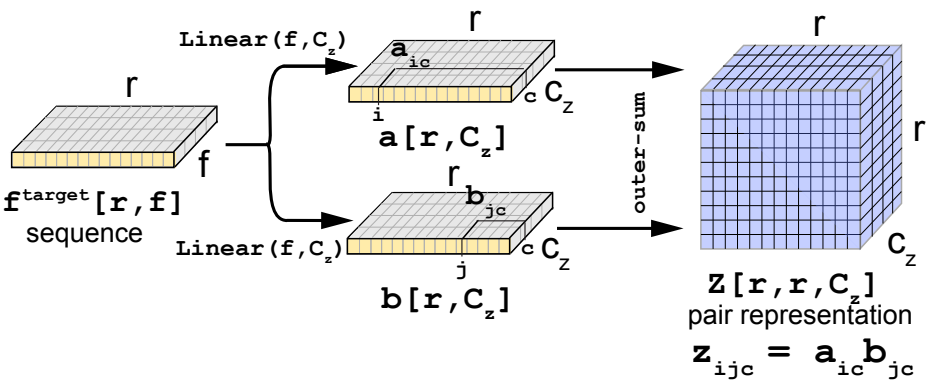
AlphaFold2 Summary (Figure 1)



AF2 Embedding Module



from seq to pair rep by "outer-sum"



ImputEmbedding(f^{target}, f^{msa}):

$a_i = Linear(f_i^{target})$ — $a_i[C_z]$, $b_i[C_z]$ $C_z = 128$

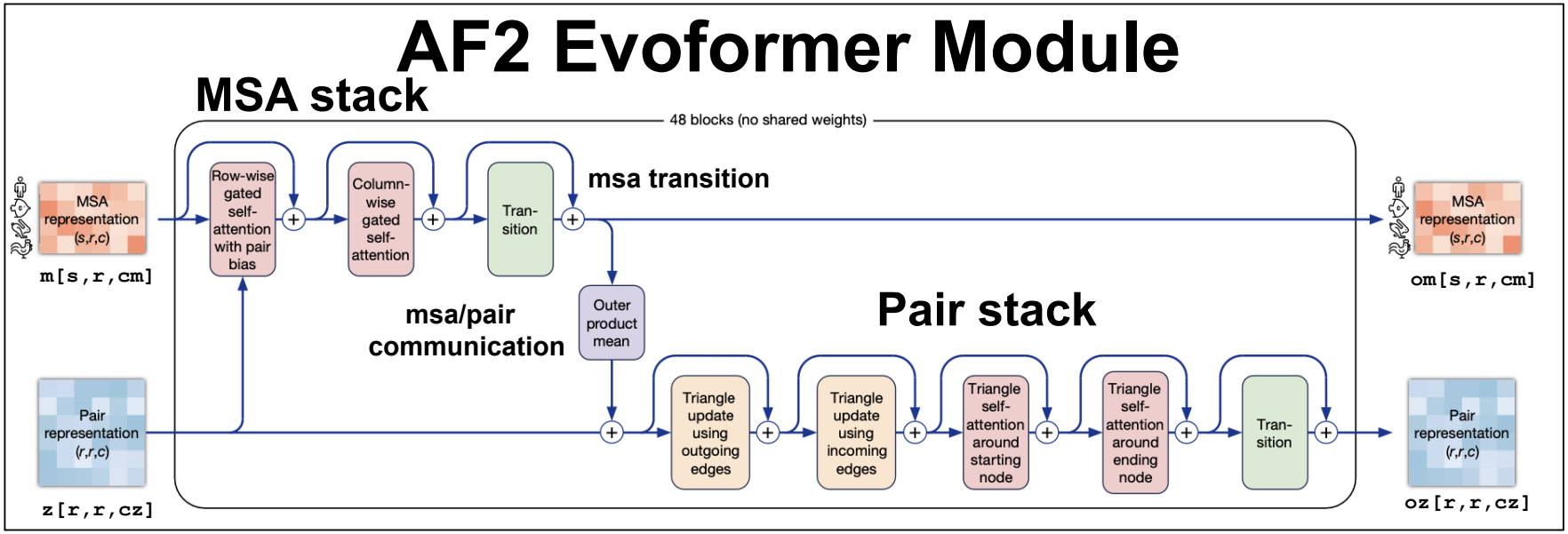
$b_i = Linear(f_i^{target})$

$z_{ij} = a_i + b_j$ (outer sum) — $z_{ij}[C_z]$

$m_{si} = Linear(f_{si}^{msa}) + Linear(f_i^{target})$ — $m_{si}[C_m]$ $C_m = 256$

return $\{m_{si}\}, \{z_{ij}\}$

AF2 Evoformer Module



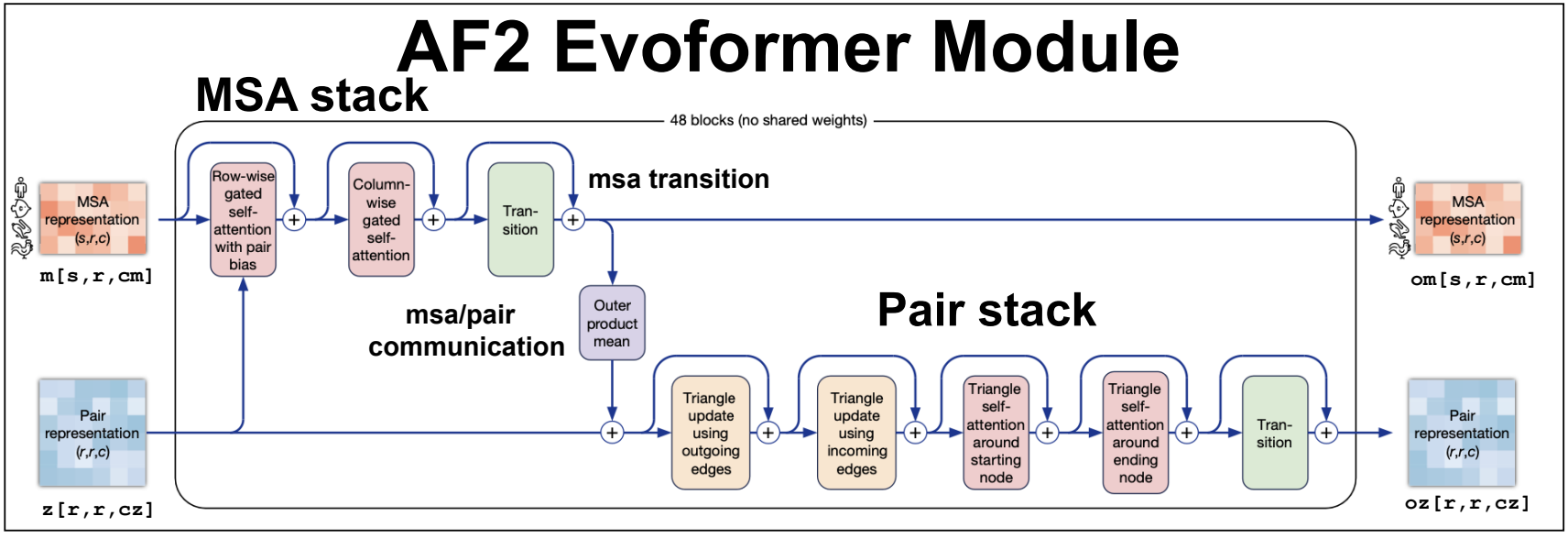
MSA stack

RowAttentionBias(m,z):

$$m_{sj} = \text{LayerNorm}(m_{sj})$$
$$q^h_{si}, k^h_{si}, v^h_{si} = \text{Linear}(m_{si})$$
$$b^h_{ij} = \text{Linear}(\text{LayerNorm}(z_{ij}))$$
[s,r,r] $$a^h_{sij} = \text{softmax}_K(q^{hT}_{si} k^h_{sj} + b_{ij})$$
$$g^h_{si} = \text{sigmoid}(\text{Linear}(m_{si}))$$
$$o^h_{si} = g^h_{si} \, \text{sum}_j(a^h_{sij} v^h_{sj})$$
$$om_{si} = \text{Linear}(\text{concat}_h(o_{si}))$$

ColAttention(m,z):

$$m_{sj} = \text{LayerNorm}(m_{sj})$$
$$q^h_{si}, k^h_{si}, v^h_{si} = \text{Linear}(m_{si})$$
[s,s,r] $$a^h_{sti} = \text{softmax}_K(q^{hT}_{si} k^h_{ti})$$
$$g^h_{si} = \text{sigmoid}(\text{Linear}(m_{si}))$$
$$o^h_{si} = g^h_{si} \, \text{sum}_t(a^h_{sti} v^h_{ti})$$
$$om_{si} = \text{Linear}(\text{concat}_h(o_{si}))$$



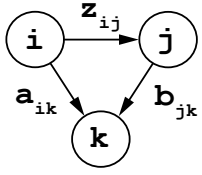
Pair stack

triangle updates

(parameters not shared)

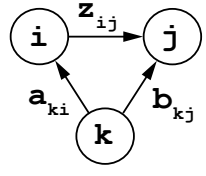
$$a_{ij}, b_{ij} = \text{sig}(\text{Linear}(z_{ij}))\,\text{Linear}(z_{ij})$$

outgoing



$$oz_{ij} = \text{Linear}(\text{sum}_k(a_{ik} b_{jk}))$$

ingoing



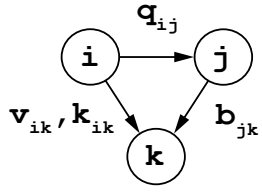
$$o_{ij} = \text{Linear}(\text{sum}_k(a_{ki} b_{kj}))$$

triangle attentions `a[r,r,r]`

(parameters not shared)

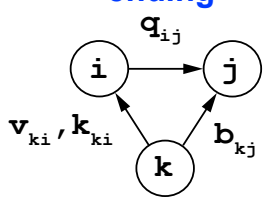
$$q^h_{ij}, k^h_{ij}, v^h_{ij} = \text{Linear}(z_{ij})$$
$$b^h_{ij} = \text{Linear}(z_{ij})$$

starting

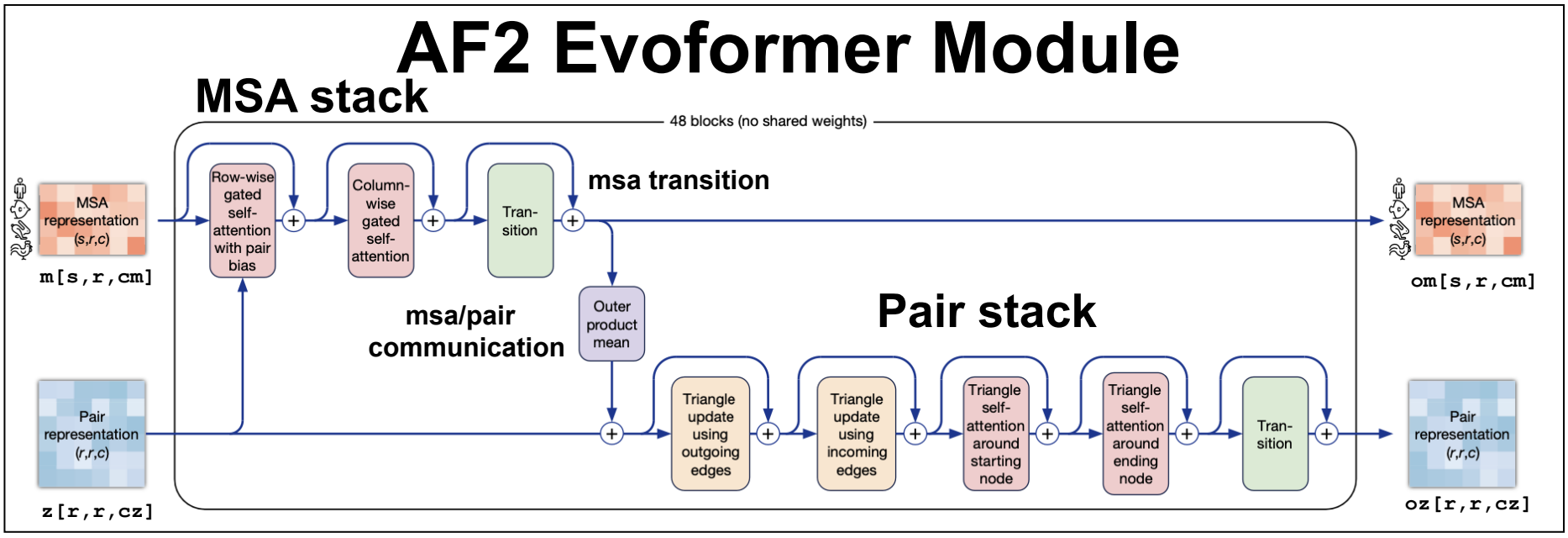


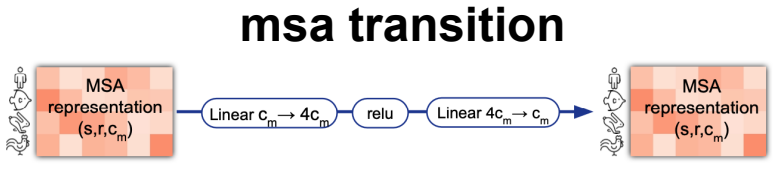
$$a^h_{ijk} = \text{softmax}_K(q^h_{ij} k^h_{ik} + b^h_{jk})$$
$$o^h_{ij} = g^h_{ij}\ \text{sum}_k(a^h_{ijk} v^h_{ik})$$
$$oz_{ij} = \text{Linear}(\text{concat}_h(o^h_{ij}))$$

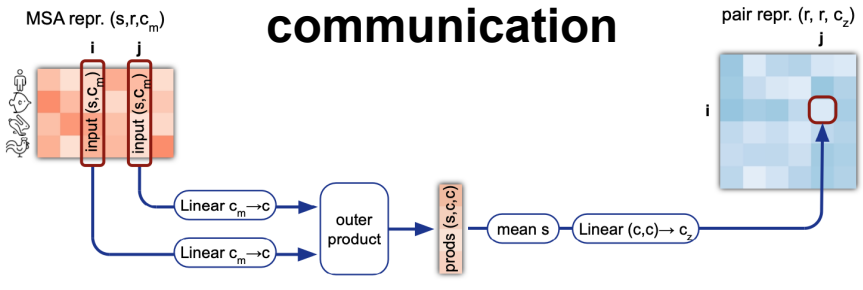
ending



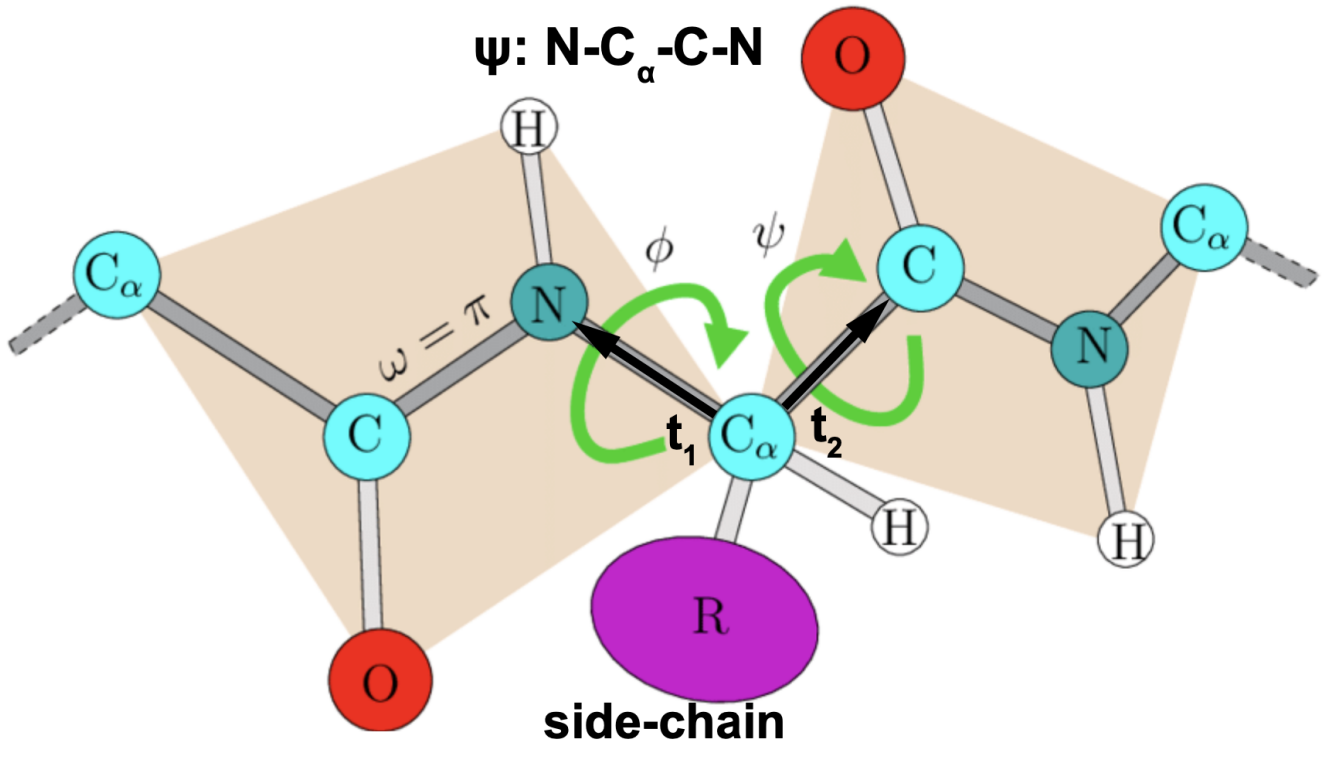
$$a^h_{ijk} = \text{softmax}_K(q^h_{ij} k^h_{ki} + b^h_{kj})$$
$$o^h_{ij} = g^h_{ij}\ \text{sum}_k(a^h_{ijk} v^h_{ki})$$
$$oz_{ij} = \text{Linear}(\text{concat}_h(o^h_{ij}))$$


AF2 Evoformer Module
MSA stack
48 blocks (no shared weights)
MSA representation (s,r,c)
m[s,r,cm]
Row-wise gated self-attention with pair bias
Column-wise gated self-attention
Tran-sition
msa transition
MSA representation (s,r,c)
om[s,r,cm]
msa/pair communication
Outer product mean
Pair stack
Pair representation (r,r,c)
z[r,r,cz]
Triangle update using outgoing edges
Triangle update using incoming edges
Triangle self-attention around starting node
Triangle self-attention around ending node
Tran-sition
Pair representation (r,r,c)
oz[r,r,cz]


msa transition
MSA representation (s,r,cm)
Linear cm→ 4cm
relu
Linear 4cm→ cm
MSA representation (s,r,cm)


communication
MSA repr. (s,r,cm)
i
j
input (s,cm)
input (s,cm)
Linear cm→c
Linear cm→c
outer product
prods (s,c,c)
mean s
Linear (c,c)→ cz
pair repr. (r, r, cz)
j
i

amino acid backbone reference frame



$\vec{t}_1 = \vec{N} - \vec{C}_\alpha$

$\vec{t}_2 = \vec{C} - \vec{C}_\alpha$

$\vec{e}_1 = \text{norm}(\vec{t}_1)$

$\vec{e}_2 = \text{norm}(\vec{t}_1 - (\vec{t}_2 \vec{e}_1)\, \vec{e}_1)$

$\vec{e}_3 = \vec{e}_1 \times \vec{e}_2$

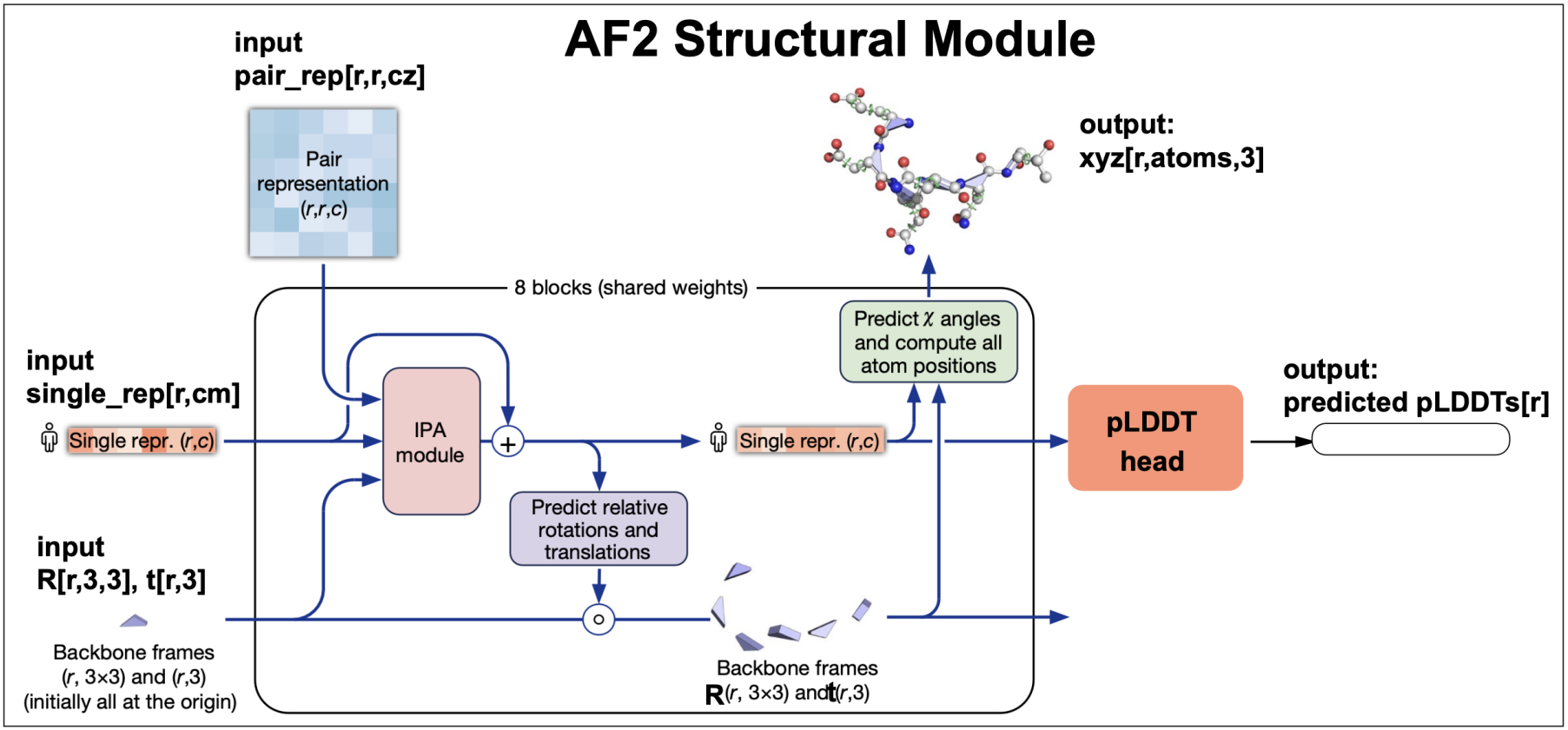
$R = \begin{bmatrix} \vec{e}_1 \\ \vec{e}_2 \\ \vec{e}_3 \end{bmatrix}$ $\quad \vec{t} = \vec{C}_\alpha$

$T = (R, \vec{t}\,)$

1 Reference frame: N-C_α-C

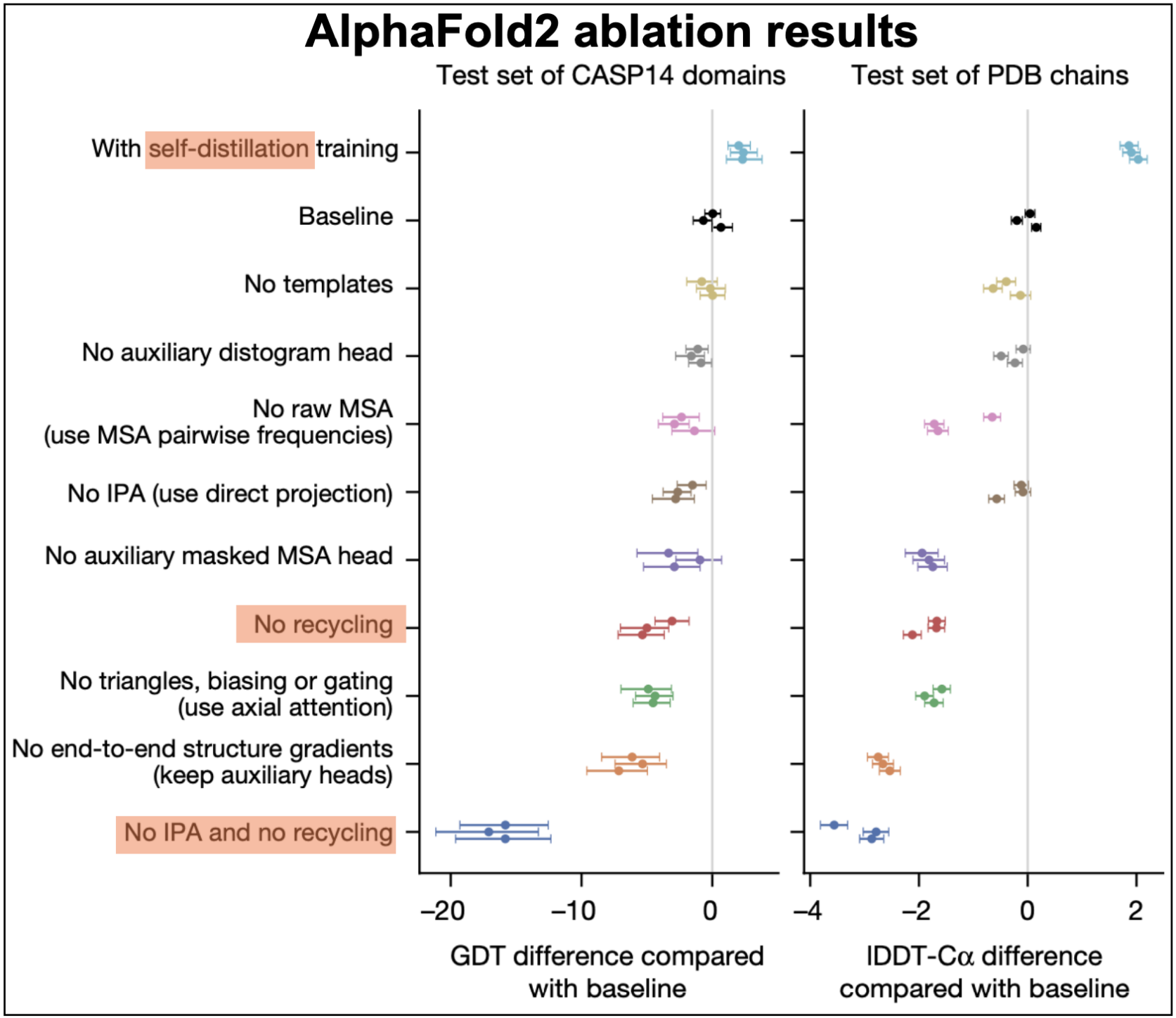
1 backbone torsion angle ψ to determine O

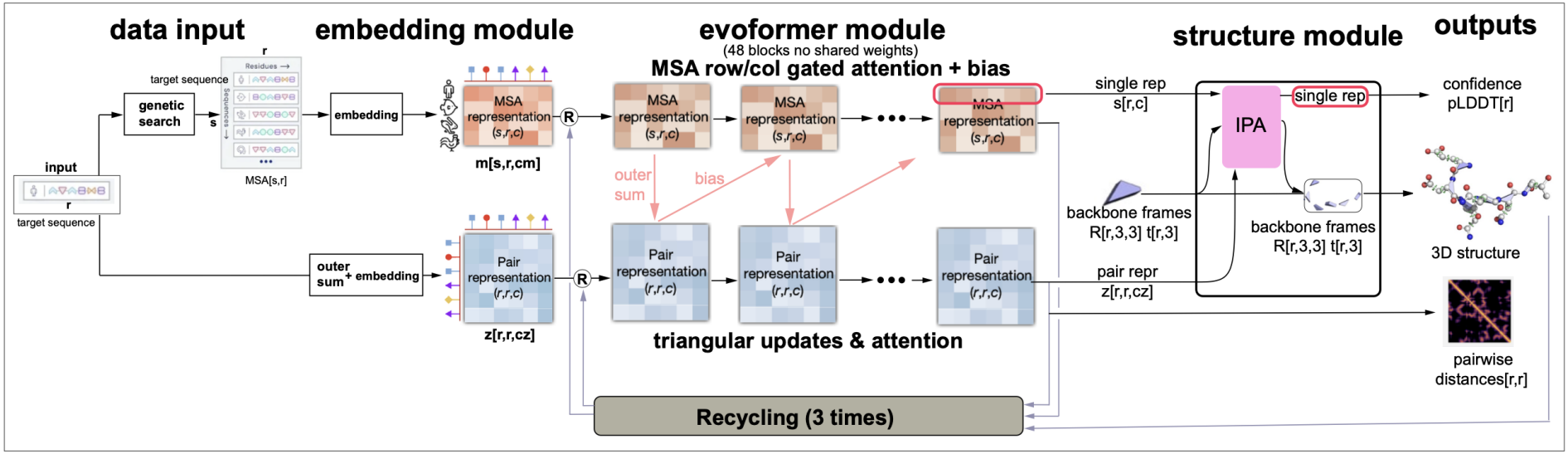
4 side-chain torsion angles


AF2 Structural Module
input
pair_rep[r,r,cz]
Pair representation (r,r,c)
output:
xyz[r,atoms,3]
8 blocks (shared weights)
Predict χ angles and compute all atom positions
input
single_rep[r,cm]
Single repr. (r,c)
IPA module
+
Single repr. (r,c)
pLDDT head
output:
predicted pLDDTs[r]
Predict relative rotations and translations
input
R[r,3,3], t[r,3]
Backbone frames (r, 3×3) and (r,3) (initially all at the origin)
Backbone frames R(r, 3×3) and t(r,3)

AF2 Losses

$$\mathcal{L} = \begin{cases} 0.5\mathcal{L}_{\text{FAPE}} + 0.5\mathcal{L}_{\text{aux}} + 0.3\mathcal{L}_{\text{dist}} + 2.0\mathcal{L}_{\text{msa}} + 0.01\mathcal{L}_{\text{conf}} & \text{training} \\ 0.5\mathcal{L}_{\text{FAPE}} + 0.5\mathcal{L}_{\text{aux}} + 0.3\mathcal{L}_{\text{dist}} + 2.0\mathcal{L}_{\text{msa}} + 0.01\mathcal{L}_{\text{conf}} + 0.01\mathcal{L}_{\text{exp resolved}} + 1.0\mathcal{L}_{\text{viol}} & \text{fine-tuning} \end{cases}$$


AlphaFold2 ablation results
Test set of CASP14 domains
Test set of PDB chains
With self-distillation training
Baseline
No templates
No auxiliary distogram head
No raw MSA
(use MSA pairwise frequencies)
No IPA (use direct projection)
No auxiliary masked MSA head
No recycling
No triangles, biasing or gating
(use axial attention)
No end-to-end structure gradients
(keep auxiliary heads)
No IPA and no recycling
−20
−10
0
−4
−2
0
2
GDT difference compared
with baseline
lDDT-Cα difference
compared with baseline


data input
embedding module
evoformer module
(48 blocks no shared weights)
MSA row/col gated attention + bias
structure module
outputs
input
target sequence
r
genetic search
s
target sequence
MSA[s,r]
embedding
MSA representation (s,r,c)
m[s,r,cm]
outer sum
+ embedding
Pair representation (r,r,c)
z[r,r,cz]
outer sum
bias
triangular updates & attention
single rep s[r,c]
backbone frames R[r,3,3] t[r,3]
pair repr z[r,r,cz]
IPA
single rep
backbone frames R[r,3,3] t[r,3]
confidence pLDDT[r]
3D structure
pairwise distances[r,r]
Recycling (3 times)