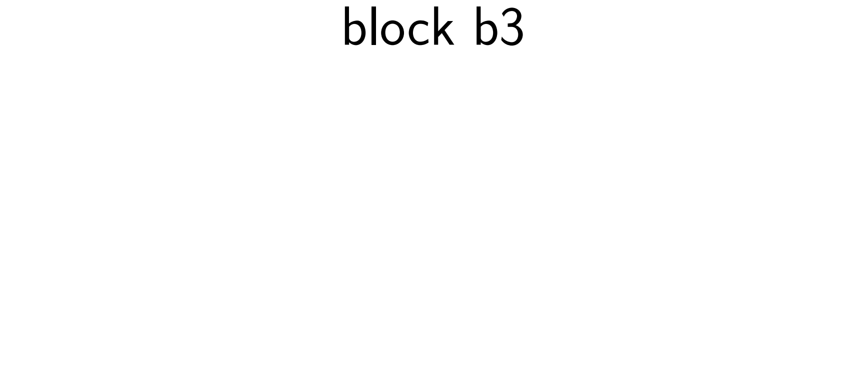
block b3



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

transformer

L

D

self-attention

L

D

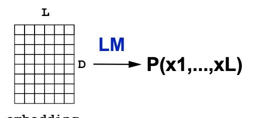
Input embedding

output embedding

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)

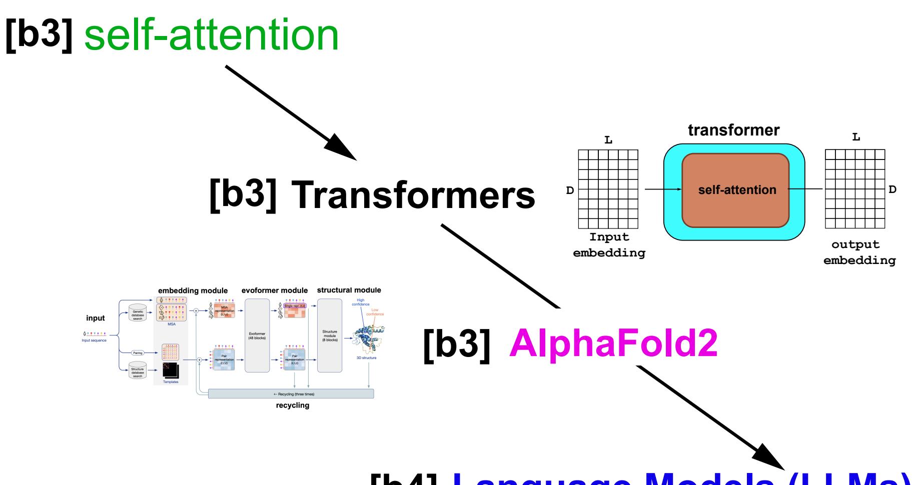
LMs usually autoregressive (which is exact, not an approximation)

x1 ~ P(x1)

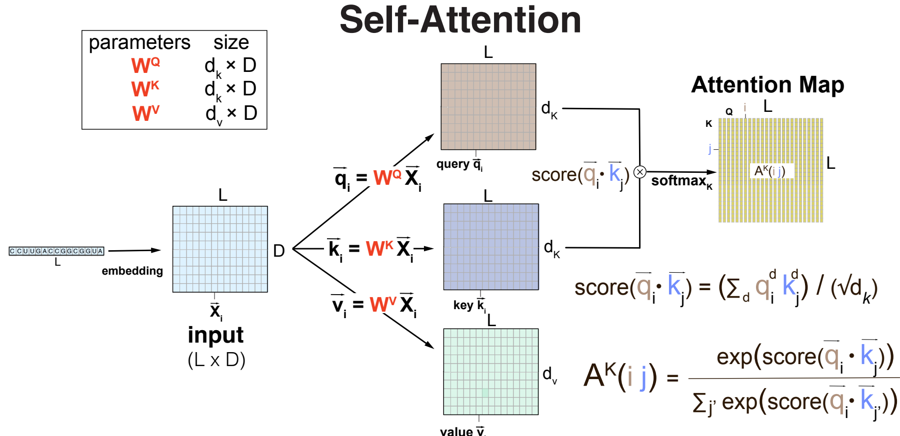
x2 ~ P(x2 | x1)

x3 ~ P(x3 | x1,x2)

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



[b4] Language Models (LLMs)


Self-Attention
parameters size
W^Q d_k × D
W^K d_k × D
W^V d_v × D
CCUUGACCGGCGGUA
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
d_K
query q_i
L
d_K
key k_i
L
d_v
value v_i
score(q_i · k_j)
softmax_K
Attention Map
Q i L
K
j
L
A^K(i j)
score(q_i · k_j) = (Σ_d q_i^d k_j^d) / (√d_k)
A^K(i j) = exp(score(q_i · k_j)) / Σ_j' exp(score(q_i · k_j'))

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

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

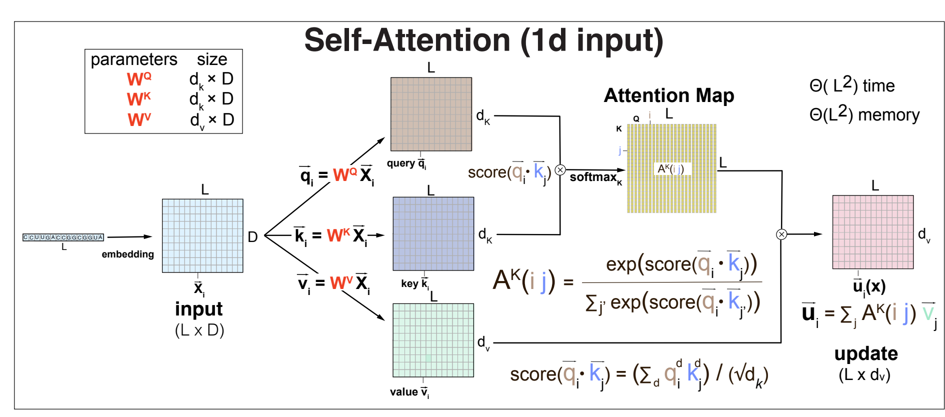
Attention between i,j is the softmax of the scores respect to the keys

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

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 $D = 3$, $(q \times k)[3]$

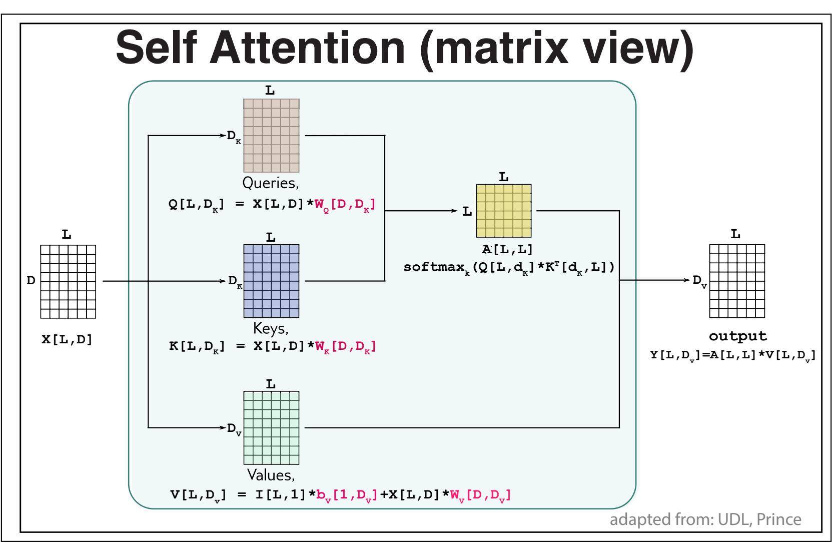
$$\begin{aligned} q \times k &= ||q|| \, ||k|| \sin(\theta) \hat{n} \\ &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ q_1 & q_2 & q_3 \\ k_1 & k_2 & k_3 \end{vmatrix} \\ &= (q_2 \, k_3 - q_3 \, k_2, -(q_1 \, k_3 - q_3 \, k_1), q_1 \, k_2 - q_2 \, k_1) \end{aligned}$$


Self-Attention (1d input)
parameters size
W^Q d_k × D
W^K d_k × D
W^V d_v × D
CCUUGACCGGCGGUA
L
embedding
L
D
input
(L x D)
q_i = W^Q X_i
k_i = W^K X_i
v_i = W^V X_i
query q_i
key k_i
value v_i
d_K
d_v
score(q_i · k_j)
softmax_K
Attention Map
A^K(i j)
Θ(L2) time
Θ(L2) memory
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)
u_i(x)
u_i = Σ_j A^K(i j) v_j
update
(L x dv)

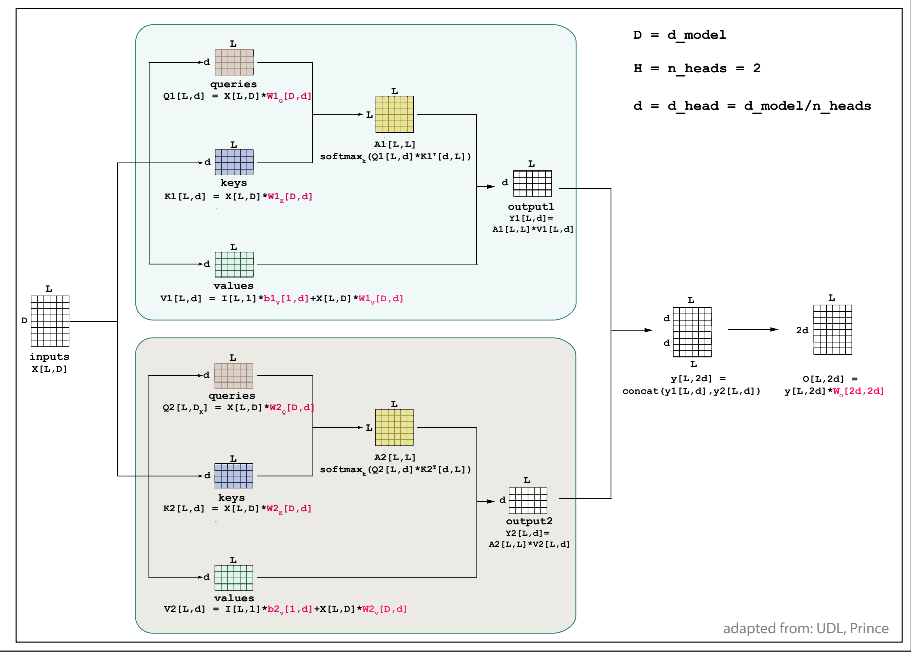
Self-Attention (code)

```
# 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]

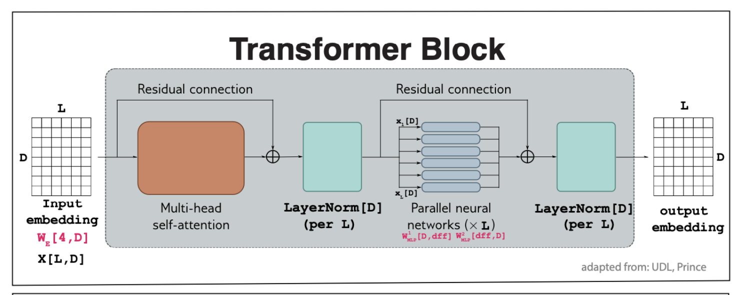


Self Attention (matrix view)
L
D
X[L,D]
L
D_K
Queries,
Q[L,D_K] = X[L,D]*W_Q[D,D_K]
L
D_K
Keys,
K[L,D_K] = X[L,D]*W_K[D,D_K]
L
D_V
Values,
V[L,D_V] = I[L,1]*b_V[1,D_V]+X[L,D]*W_V[D,D_V]
L
L
A[L,L]
softmax_k(Q[L,d_K]*K^T[d_K,L])
L
D_V
output
Y[L,D_V]=A[L,L]*V[L,D_V]
adapted from: UDL, Prince



D = d_model
H = n_heads = 2
d = d_head = d_model/n_heads
queries
Q1[L,d] = X[L,D]*W1Q[D,d]
keys
K1[L,d] = X[L,D]*W1K[D,d]
values
V1[L,d] = I[L,1]*b1V[1,d]+X[L,D]*W1V[D,d]
A1[L,L]
softmaxk(Q1[L,d]*K1T[d,L])
output1
Y1[L,d]=
A1[L,L]*V1[L,d]
inputs
X[L,D]
queries
Q2[L,DK] = X[L,D]*W2Q[D,d]
keys
K2[L,d] = X[L,D]*W2K[D,d]
values
V2[L,d] = I[L,1]*b2V[1,d]+X[L,D]*W2V[D,d]
A2[L,L]
softmaxk(Q2[L,d]*K2T[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]*WO[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
```




```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

$$vbins = [-32, -31, ..., 32]$$
$$d_{ij} = j - i$$
$$p_{ij} = LinearLayer(onehot(dij, vbins))$$

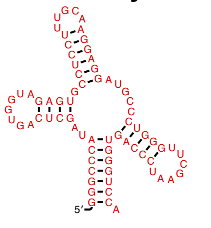
Why Self-attention on multiple sequence alignments (MSAs)?

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

RNA secondary structures

structural information

X-ray, NMR, CryoEM



RNA alignments

evolutionary information

genomes, homology

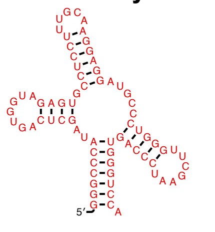
```
GGGCCCAUAGCUCAGU-GGU-AGAGUGCCUCCUUUGCAAGGAGGAUGC-CCUG-GGUUCGAAUCCCAGUGGGUCCA
CAUUAGAUGACUGAA--AG--CAAGUACUGGUCUCUUAAACCAUUUAA-UAGUAAAUUAGCACUUACUUCUAAUGA
GCGCCCGUAGCUCAAUUGGAUAGAGCGUUUGACUACGGAUCAAAAGGU-UAGG-GGUUCGACUCCUCUCGGGCGCG
GGGCCUGUAGCUCAGCUGGUUAGAGCGCACGCCUGAUAAGCGUGAGGU-CGGU-GGUUCGAGUCCACUCAGGCCCA
AGGGGCAUAGUUUAAC-GGU-AGAACAGAGGUCUCCAAAACCUCCGG--UGUG-GGUUCGAUUCCUACUGCCCCUG
AGGGGCGUAGUUCAAUUGGU-AGAGCACCGGUCUCCAAAACCGGGUGU-UGGG-AGUUCGAGUCUCUCCGCCCCUG
GCUGAUAUGGCUCAGUUGGU-AGAGCGCACCCUUGGUAAGGGUGAGGU-CCCC-AGUUCGACUCUGGGUAUCAGCA
UGGGGCGUGGCCAAGU-GGU-AAGGCAGCGGUUUUGGUCCCGUUACU-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



3'

5' \ 3'	A	C	G	U
A	0	0	0	1
C	0	0	10	0
G	0	5	0	2
U	4	0	0	0

maximum likelihood training

RNA alignments

evolutionary information

genomes, homology

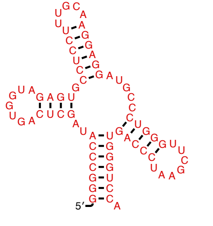
```
G..CA
C..GA
G..CG
G..CA
A..UG
A..UG
G..CA
U..AG
G..CU
G..CA
G..CA
A..UA
A..UA
G..UA
G..UU
A..UA
U..AG
```

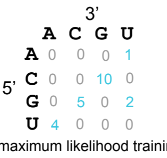
Why Self-attention on multiple sequence alignments (MSAs)?

RNA secondary structures

structural information

X-ray, NMR, CryoEM





RNA alignments

evolutionary information

genomes, homology

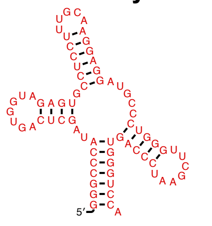
```
GG..CCA
CA..UGA
GC..GCG
GG..CCA
AG..CUG
AG..CUG
GC..GCA
UG..CAG
GC..GCU
GC..GCA
GC..GCA
AG..CUA
AA..UUA
GU..AUA
GG..CUU
AC..GUA
UA..UAG
CA..UGA
```

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

RNA secondary structures

structural information

X-ray, NMR, CryoEM



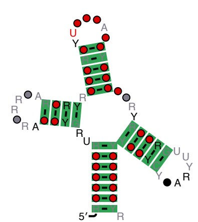
5' \ 3'	A	C	G	U
A	0	0	0	1
C	0	0	10	0
G	0	5	0	2
U	4	0	0	0

maximum likelihood training

RNA alignments

evolutionary information

genomes, homology



MSA attention by row/column

parameters	size
W^Q	$D \times d_k$
W^K	$D \times d_k$
W^V	$D \times d_v$

$\vec{x}_{sl}$ **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

L S D s l

$\vec{q}_{sl}$ $\vec{k}_{s1}$ $\vec{k}_{s2}$ ... $\vec{k}_{sL}$

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

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

self-attention by column

L S D s l

$\vec{q}_{sl}$ $\vec{k}_{Sl}$ ⋮ $\vec{k}_{2l}$ $\vec{k}_{1l}$

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

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

```
# 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	$D \times d_k$
W^K	$D \times d_k$
W^V	$D \times d_v$

query vectors

key vectors

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

softmax$_K$

attention map

Q, K, S x L, $P(K_{bj} \mid Q_{ai})$

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

Attention by row

query vectors, key vectors, S, L

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

softmax$_k$

attention map (Q, K, L, S x L)

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

tied row Attention

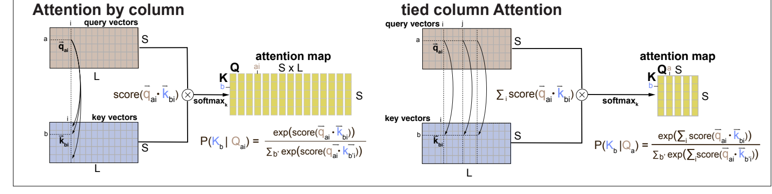
query vectors, key vectors, S, L

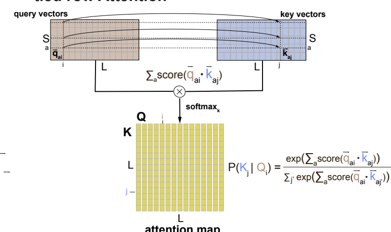
$\sum_a \text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj})$

softmax$_k$

attention map (Q, K, L)

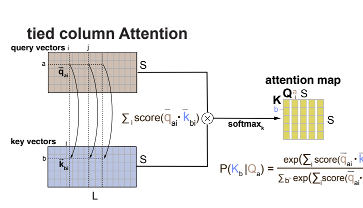
$$P(K_j \mid Q_i) = \frac{\exp(\sum_a \text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj}))}{\sum_{j'} \exp(\sum_a \text{score}(\vec{q}_{ai} \cdot \vec{k}_{aj'}))}$$





Tied row attention

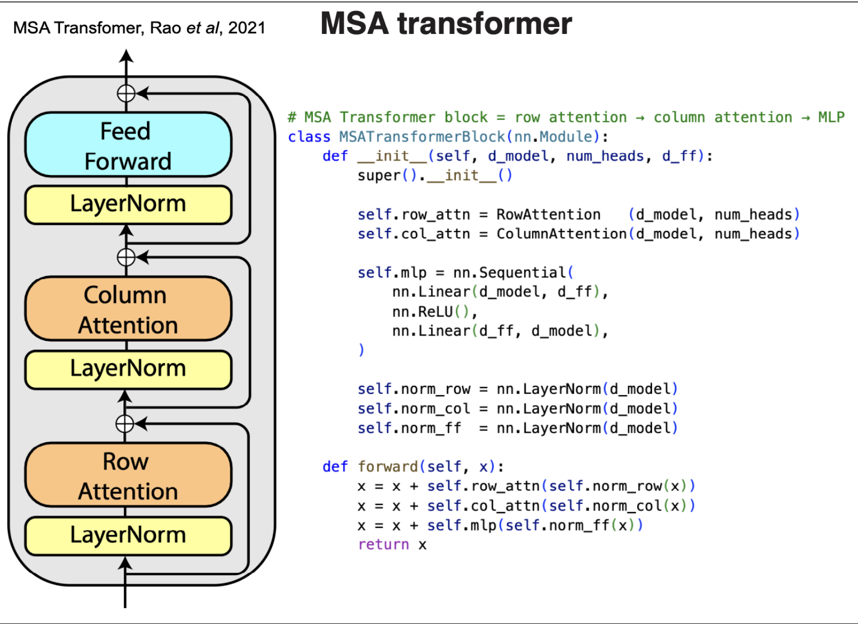
```python
class TiedRowAttention(nn.Module):
    def __init__(self, d_model, num_heads, tie_mode="mean"):
        super().__init__()
        self.num_heads = num_heads
        self.head_dim  = d_model // num_heads
        self.tie_mode = tie_mode    # "mean" or "sum"

        # 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

```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



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

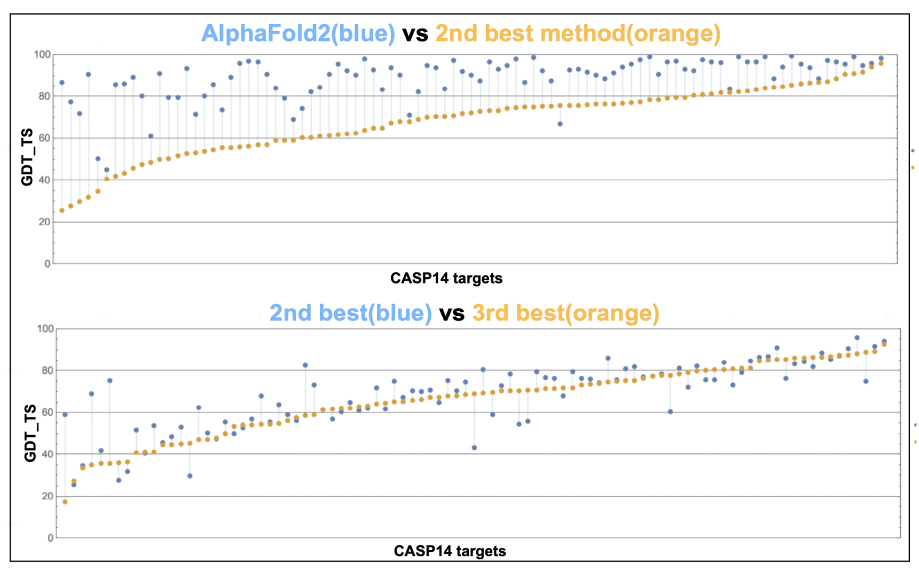
Accepted: 12 July 2021

Published online: 15 July 2021

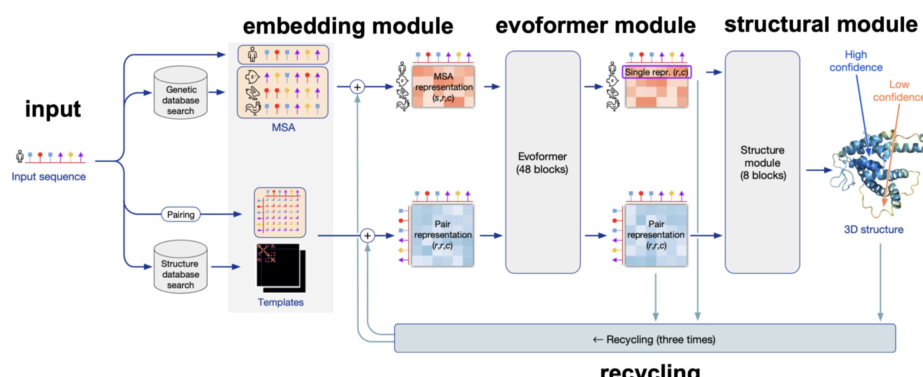
Open access

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

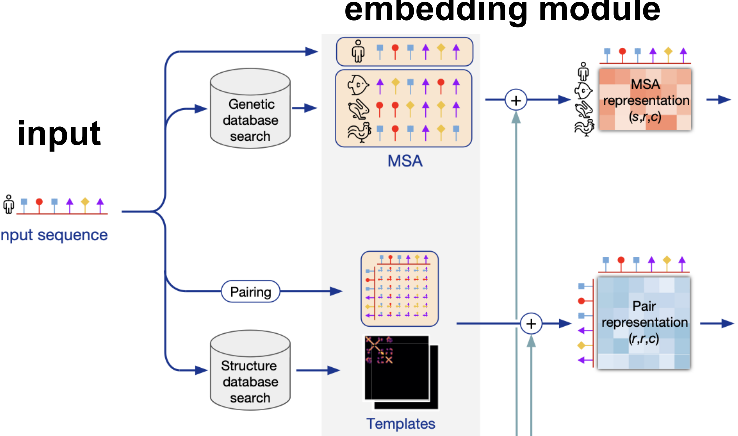
AlphaFold2 Summary (Figure 1)

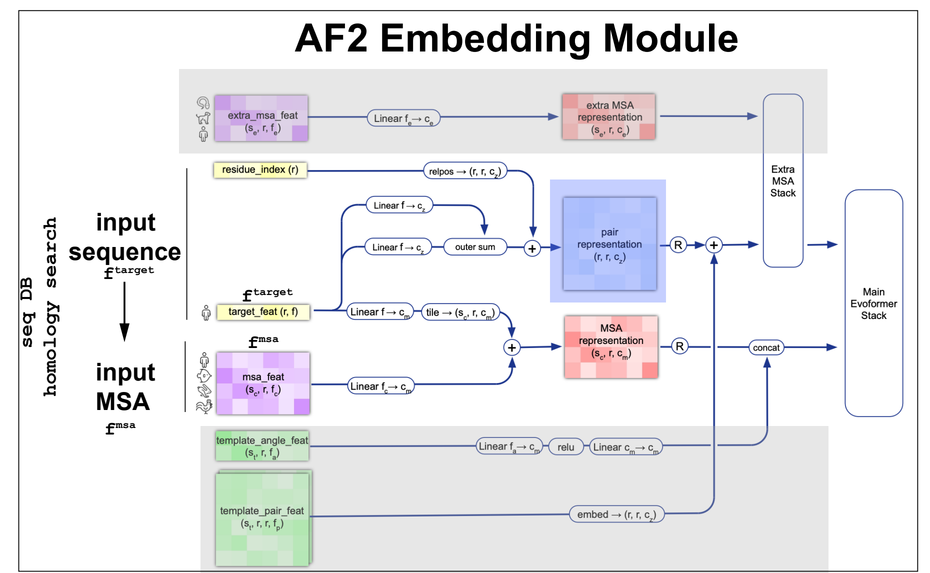


AlphaFold2 (self-reported) Key Innovations

1. **evoformer** Join embedding of alignments and pair prepresentations "Direct reasoning between the spatial and evolutionary relationships"
2. **structural module** Residue gas: a new representation of 3D structure as a collection of rotation and translation for each amino acid.
3. **structural module** Equivariant attention
4. **training** End-to-end structure prediction
5. **training** Use of intermediate losses
6. **training** Masked msa loss
7. **training** Distillation: learning from unlabelled sequences

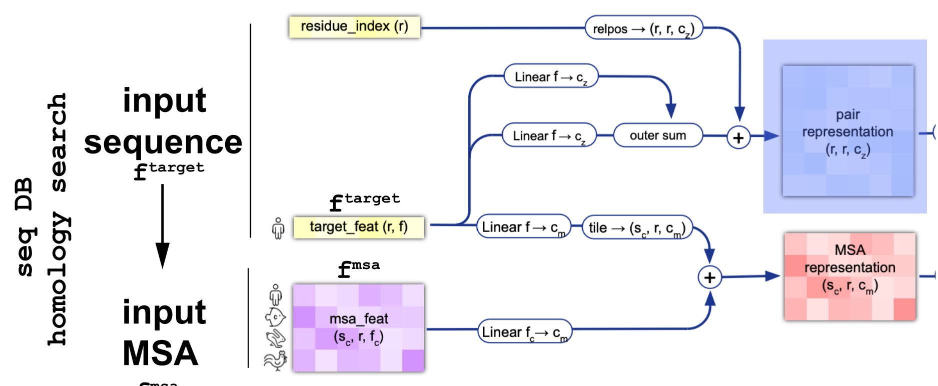
AlphaFold2 Summary (Figure 1)



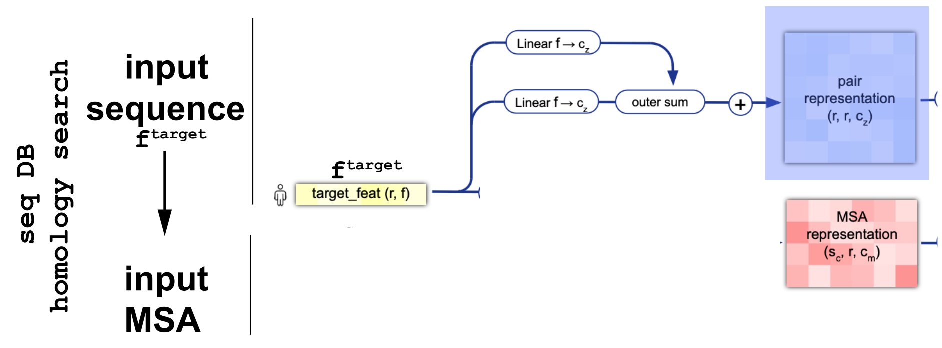


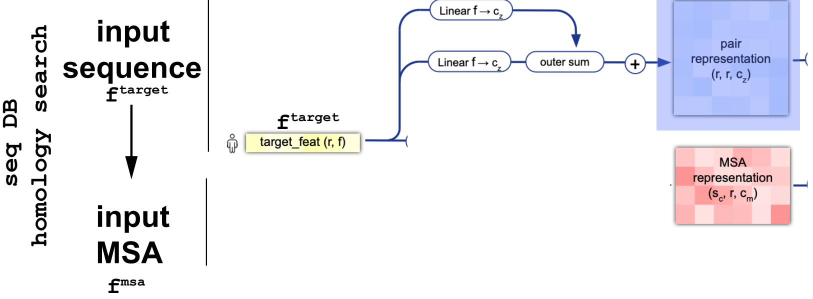
AF2 Embedding Module
extra_msa_feat
(s_e, r, f_e)
Linear f_e → c_e
extra MSA
representation
(s_e, r, c_e)
Extra
MSA
Stack
residue_index (r)
relpos → (r, r, c_z)
Linear f → c_z
Linear f → c_z
outer sum
pair
representation
(r, r, c_z)
R
Main
Evoformer
Stack
input
sequence
f^target
seq DB
homology search
f^target
target_feat (r, f)
Linear f → c_m
tile → (s_c, r, c_m)
f^msa
MSA
representation
(s_c, r, c_m)
R
concat
input
MSA
f^msa
msa_feat
(s_c, r, f_c)
Linear f_c → c_m
template_angle_feat
(s_t, r, f_a)
Linear f_a → c_m
relu
Linear c_m → c_m
template_pair_feat
(s_t, r, r, f_p)
embed → (r, r, c_z)

AF2 Embedding Module



AF2 Embedding Module





from seq to pair rep by "outer-sum"

Linear(f,C_z)

r

a_{ic}

c C_z

i

$a[r,C_z]$

r

f

$f^{target}[r,f]$

sequence

r

b_{jc}

Linear(f,C_z)

c C_z

j

$b[r,C_z]$

outer-sum

r

r

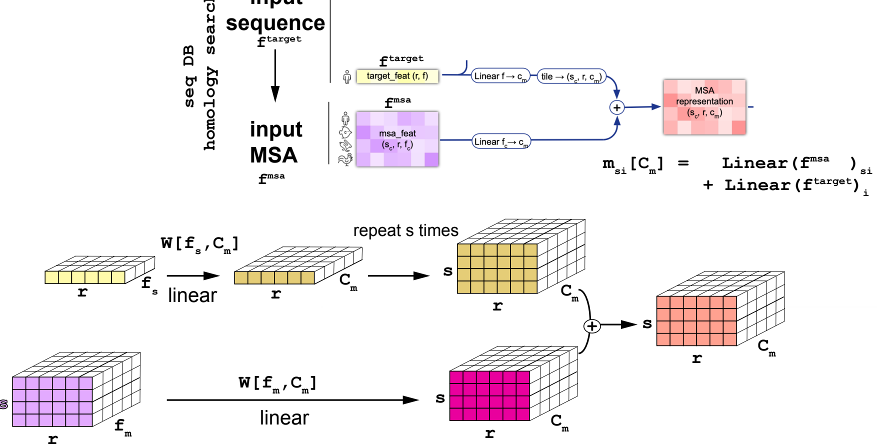
C_z

$Z[r,r,C_z]$

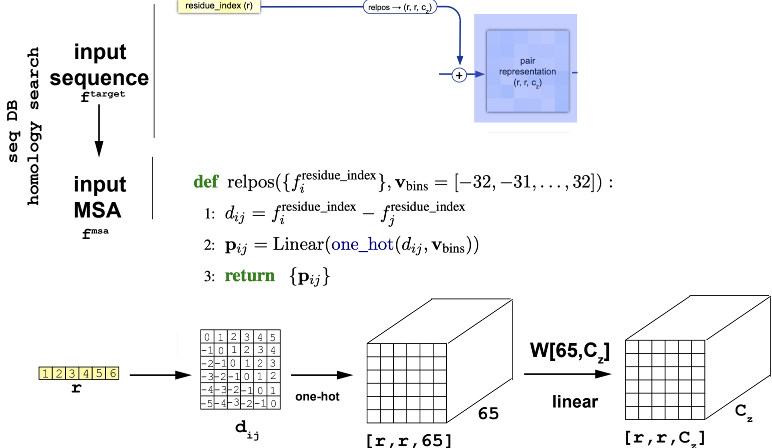
pair representation

$$z_{ijc} = a_{ic} + b_{jc}$$

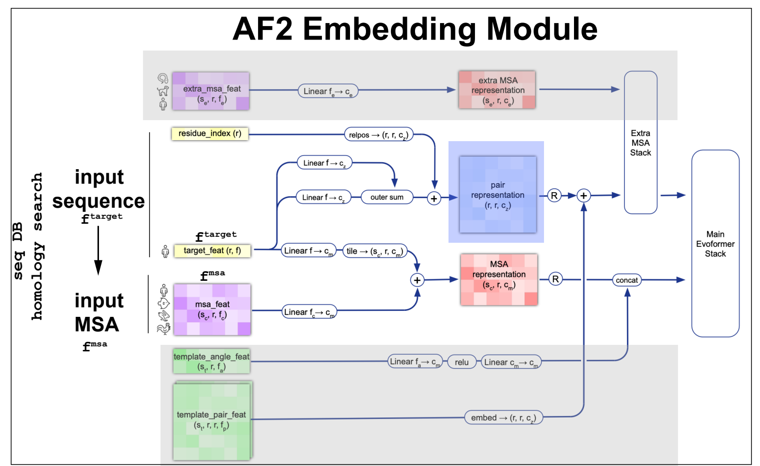
AF2 Embedding Module



AF2 Embedding Module



This inductive bias de-emphasizes primary sequence distances.
Compared to the more traditional approach of encoding positions in the frequency space, this relative encoding scheme empirically allows the network to be evaluated without quality degradation on much longer sequences than it was trained on.



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

$$a_i = \mathrm{Linear}(f_i^{target})$$

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

$a_i[C_z]$, $b_i[C_z]$ $C_z = 128$

$$z_{ij} = a_i \oplus b_j$$

$$z_{ij} = z_{ij} + \mathrm{relpos}(f^{index})$$

$z_{ij}[C_z]$

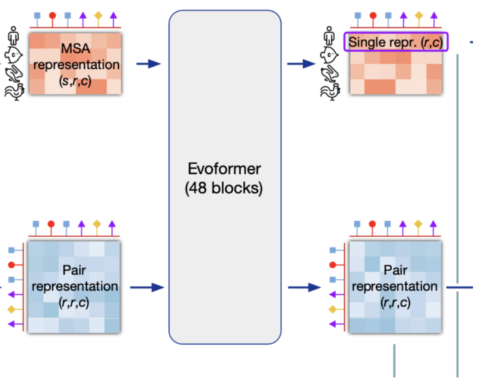
$$m_{si} = \mathrm{Linear}(f_{si}^{msa}) + \mathrm{Linear}(f_i^{target})$$

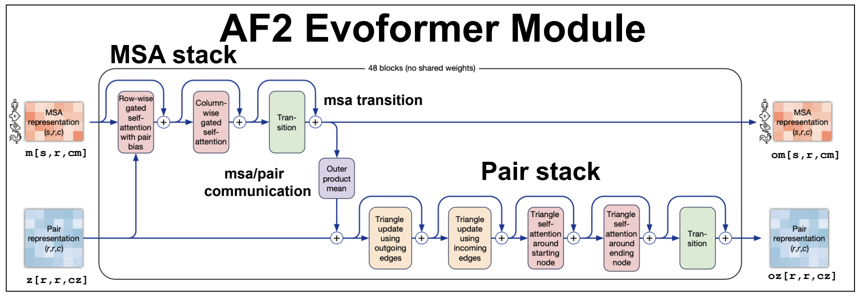
$m_{si}[C_m]$ $C_m = 256$

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

AlphaFold2 Summary (Figure 1)

evoformer module





MSA stack

RowAttentionBias(m,z):

$$m_{sj} = \mathrm{LayerNorm}(m_{sj})$$

$$q^h_{si}, k^h_{si}, v^h_{si} = \mathrm{Linear}(m_{si})$$

$$b^h_{ij} = \mathrm{Linear}(\mathrm{LayerNorm}(z_{ij}))$$

[s,r,r] $$a^h_{sij} = \mathrm{softmax}_K(q^{hT}_{si} k^h_{sj} + b_{ij})$$

$$g^h_{si} = \mathrm{sigmoid}(\mathrm{Linear}(m_{si}))$$

$$o^h_{si} = g^h_{si}\ \mathrm{sum}_j(a^h_{sij} v^h_{sj})$$

$$om_{si} = \mathrm{Linear}(\mathrm{concat}_h(o_{si}))$$

ColAttention(m,z):

$$m_{sj} = \mathrm{LayerNorm}(m_{sj})$$

$$q^h_{si}, k^h_{si}, v^h_{si} = \mathrm{Linear}(m_{si})$$

[s,s,r] $$a^h_{sti} = \mathrm{softmax}_K(q^{hT}_{si} k^h_{ti})$$

$$g^h_{si} = \mathrm{sigmoid}(\mathrm{Linear}(m_{si}))$$

$$o^h_{si} = g^h_{si}\ \mathrm{sum}_t(a^h_{sti} v^h_{ti})$$

$$om_{si} = \mathrm{Linear}(\mathrm{concat}_h(o_{si}))$$

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

Transition

msa transition

Outer product mean

msa/pair communication

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

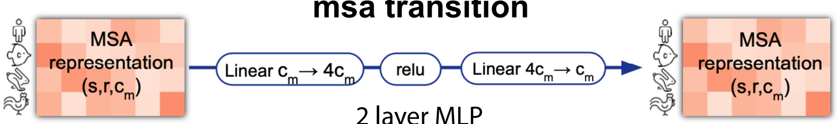
Triangle self-attention around ending node

Transition

MSA representation (s,r,c)
om[s,r,cm]

Pair representation (r,r,c)
oz[r,r,cz]

msa transition



2 layer MLP

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

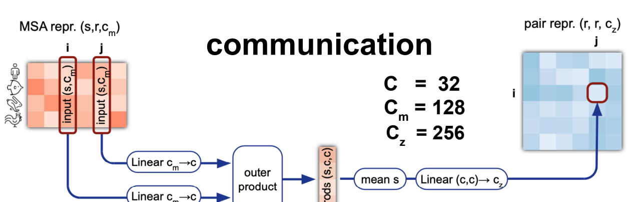
Triangle self-attention around starting node

Triangle self-attention around ending node

Tran-sition

Pair representation (r,r,c)

oz[r,r,cz]



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 → Transition

msa transition

MSA representation (s,r,c) — $om[s,r,cm]$

msa/pair communication

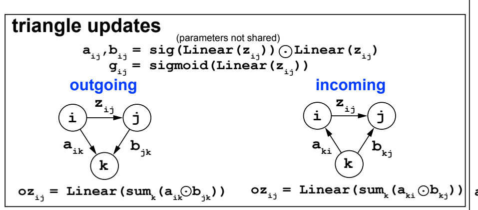
Outer product mean

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 → Transition

Pair representation (r,r,c) — $oz[r,r,cz]$



triangle self-attentions $a[r,r,r]$

(parameters not shared)

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

$$b_{ij} = \mathrm{Linear}(z_{ij})$$

$$g_{ij} = \mathrm{sigmoid}(\mathrm{Linear}(z_{ij}))$$

starting

q_{ij}: i → j; v_{ik}, k_{ik}: i → k; b_{jk}: j → k

$$a^h_{ijk} = \mathrm{softmax}_K(q^{hT}_{ij} k^h_{ik} + b^h_{jk})$$

$$o^h_{ij} = g^h_{ij} \odot \mathrm{sum}_k(a^h_{ijk} v^h_{ik})$$

$$oz_{ij} = \mathrm{Linear}(\mathrm{concat}_h(o^h_{ij}))$$

ending

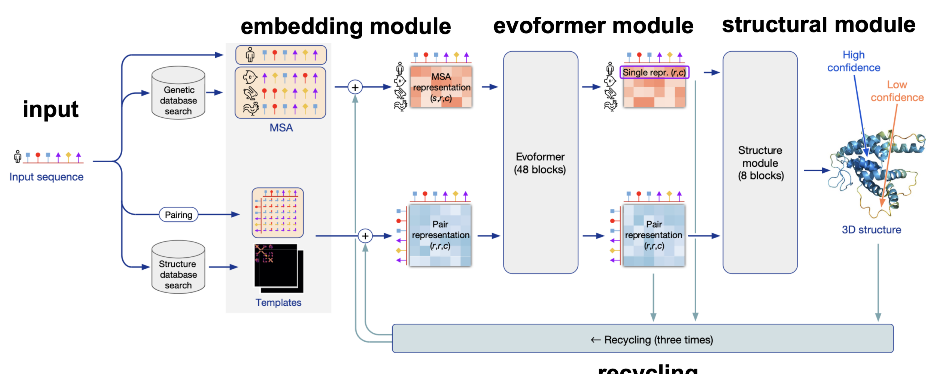
q_{ij}: i → j; v_{ki}, k_{ki}: k → i; b_{kj}: k → j

$$a^h_{ijk} = \mathrm{softmax}_K(q^{hT}_{ij} k^h_{ki} + b^h_{kj})$$

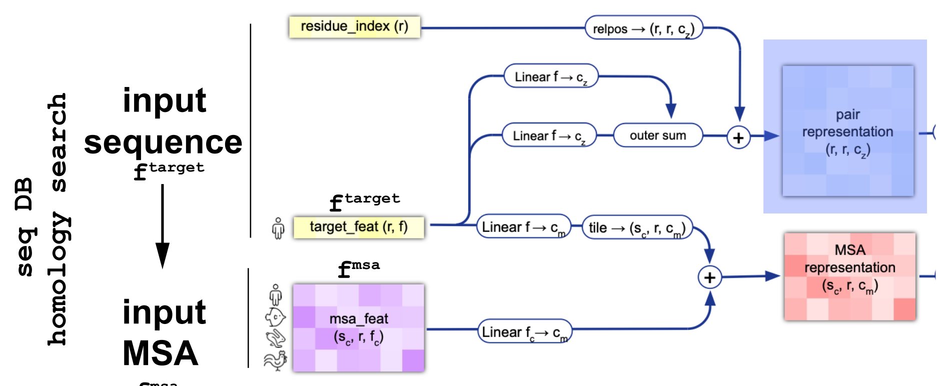
$$o^h_{ij} = g^h_{ij} \odot \mathrm{sum}_k(a^h_{ijk} v^h_{ki})$$

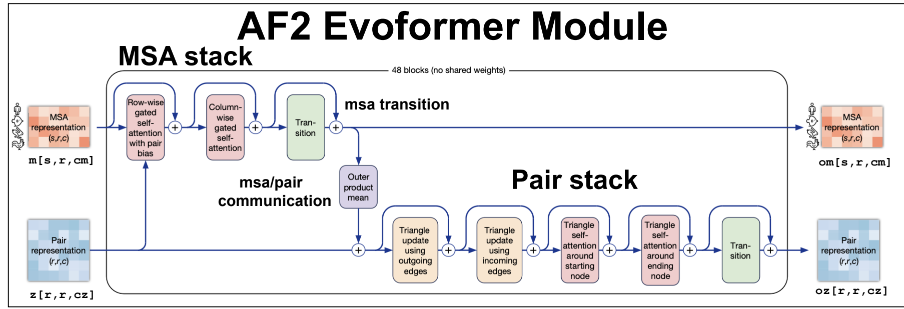
$$oz_{ij} = \mathrm{Linear}(\mathrm{concat}_h(o^h_{ij}))$$

AlphaFold2 Summary

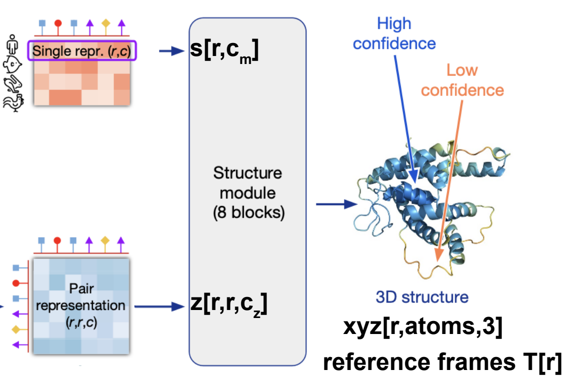


AF2 Embedding Module




AF2 Evoformer Module
MSA stack
48 blocks (no shared weights)
msa transition
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 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]

structural module



reference frames T_i = $t_i[3]$ translation
+ $R_i[3,3]$ rotation

amino acid backbone reference frame

ψ: N-C_α-C-N

ϕ ψ $\omega = \pi$

t_1 t_2

side-chain

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

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

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

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

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

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

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

1 Reference frame: N-C_α-C

1 backbone torsion angle ψ to determine O

4 side-chain torsion angles

amino acid chain = sequence of reference frames

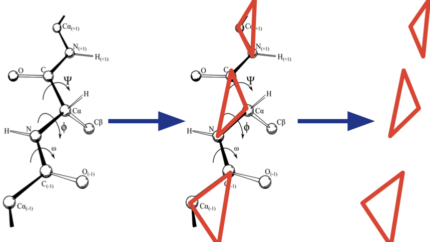
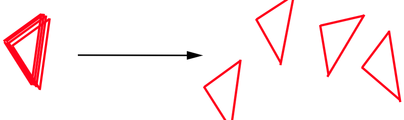


Image: Dcrjsr, vectorised Adam Rędzikowski (CC BY 3.0, Wikipedia)



Special Eucliean Group

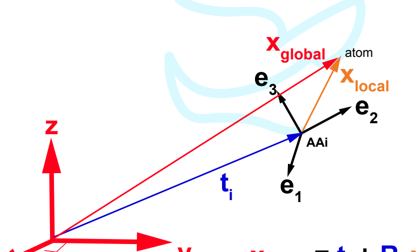
SE(3) = Translation (t) + Rotation (R)

$$t = (t_1, t_2, t_3) \qquad R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

SO(3) = Special Orthogonal group

$$T = (R,t) = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

AA3 AA2 AA1

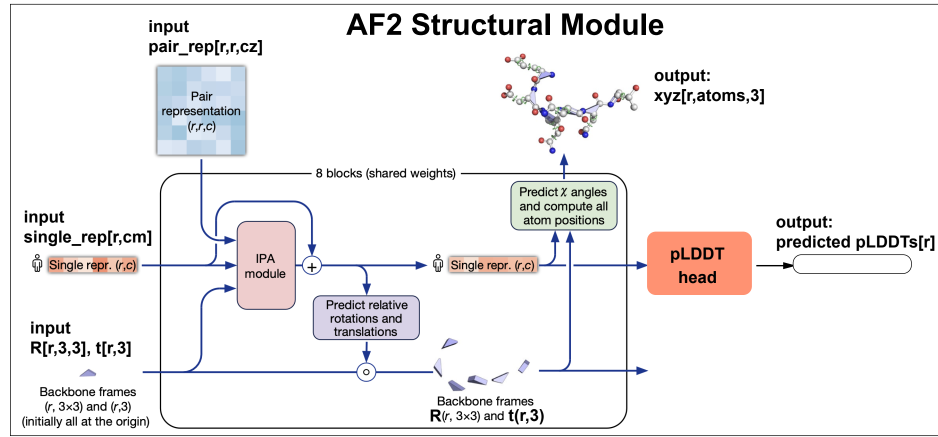


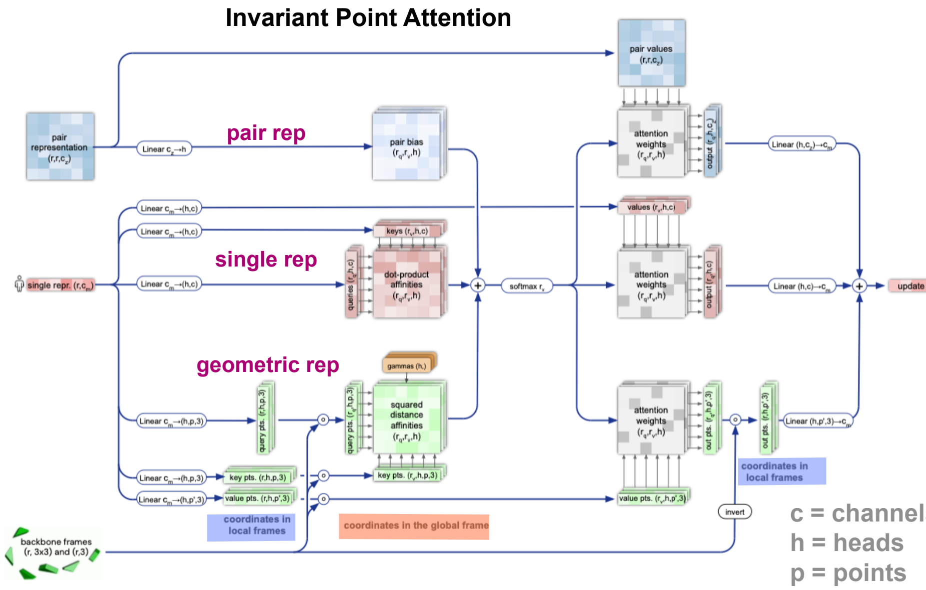
$$R_i = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix}$$

x

$$x_{global} = t_i + R_i\, x_{local} \quad = T_i\, x_{local}$$

$$x_{local} = R_i^{-1}(x_{global} - t_i) = T_i^{-1}\, x_{global}$$


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)



Invariant Point Attention
pair values (r,r,c_z)
pair representation (r,r,c_z)
Linear c_z→h
pair rep
pair bias (r_q,r_v,h)
attention weights (r_q,r_v,h)
output (r_q,h,c_z)
Linear (h,c_z)→c_m
Linear c_m→(h,c)
values (r_v,h,c)
Linear c_m→(h,c)
keys (r_v,h,c)
single rep
single repr. (r,c_m)
Linear c_m→(h,c)
queries (r_q,h,c)
dot-product affinities (r_q,r_v,h)
softmax r_v
attention weights (r_q,r_v,h)
output (r_q,h,c)
Linear (h,c)→c_m
update (r,
geometric rep
gammas (h,)
Linear c_m→(h,p,3)
query pts. (r,h,p,3)
query pts. (r_q,h,p,3)
squared distance affinities (r_q,r_v,h)
attention weights (r_q,r_v,h)
out pts. (r_q,h,p',3)
out pts. (r,h,p',3)
Linear (h,p',3)→c_m
Linear c_m→(h,p,3)
key pts. (r,h,p,3)
key pts. (r_v,h,p,3)
Linear c_m→(h,p',3)
value pts. (r,h,p',3)
value pts. (r_v,h,p',3)
coordinates in local frames
coordinates in the global frame
coordinates in local frames
invert
backbone frames (r, 3x3) and (r,3)
c = channels
h = heads
p = points

Algorithm 22 Invariant point attention (IPA)

def $\text{InvariantPointAttention}(\{\mathbf{s}_i\}, \{\mathbf{z}_{ij}\}, \{T_i\}, N_{\text{head}} = 12, c = 16, N_{\text{query points}} = 4, N_{\text{point values}} = 8):$

1: $\mathbf{q}_i^h, \mathbf{k}_i^h, \mathbf{v}_i^h = \text{LinearNoBias}(\mathbf{s}_i)$ $\quad \mathbf{q}_i^h, \mathbf{k}_i^h, \mathbf{v}_i^h \in \mathbb{R}^c,\ h \in \{1, \dots, N_{\text{head}}\}$

2: $\vec{\mathbf{q}}_i^{hp}, \vec{\mathbf{k}}_i^{hp} = \text{LinearNoBias}(\mathbf{s}_i)$ $\quad \vec{\mathbf{q}}_i^{hp}, \vec{\mathbf{k}}_i^{hp}, \in \mathbb{R}^3,\ p \in \{1, \dots, N_{\text{query points}}\}$, units: nanometres

3: $\vec{\mathbf{v}}_i^{hp} = \text{LinearNoBias}(\mathbf{s}_i)$ $\quad \vec{\mathbf{v}}_i^{hp} \in \mathbb{R}^3,\ p \in \{1, \dots, N_{\text{point values}}\}$, units: nanometres

4: $b_{ij}^h = \text{LinearNoBias}(\mathbf{z}_{ij})$

5: $w_C = \sqrt{\frac{2}{9N_{\text{query points}}}},$

6: $w_L = \sqrt{\frac{1}{3}}$

7: $a_{ij}^h = \text{softmax}_j\left(w_L\left(\frac{1}{\sqrt{c}}\,{\mathbf{q}_i^h}^\top \mathbf{k}_j^h + b_{ij}^h - \frac{\gamma^h w_C}{2}\sum_p \left\| T_i \circ \vec{\mathbf{q}}_i^{hp} - T_j \circ \vec{\mathbf{k}}_j^{hp} \right\|^2\right)\right)$

8: $\tilde{\mathbf{o}}_i^h = \sum_j a_{ij}^h \mathbf{z}_{ij}$

9: $\mathbf{o}_i^h = \sum_j a_{ij}^h \mathbf{v}_j^h$

10: $\vec{\mathbf{o}}_i^{hp} = T_i^{-1} \circ \sum_j a_{ij}^h \left(T_j \circ \vec{\mathbf{v}}_j^{hp}\right)$

11: $\tilde{\mathbf{s}}_i = \text{Linear}\left(\text{concat}_{h,p}(\tilde{\mathbf{o}}_i^h, \mathbf{o}_i^h, \vec{\mathbf{o}}_i^{hp}, \left\|\vec{\mathbf{o}}_i^{hp}\right\|)\right)$

12: **return** $\{\tilde{\mathbf{s}}_i\}$

$s[r, C_m]$ —IPA→ $update[r, C_m]$

After IPA.....

$update[r, C_m]$ —$W[C_m, 6]$ linear→ $quaterion[r,6] = b[r],\ c[r],\ d[r],\ t[r,3]$

T_i = rotation R_i [3,3] + translation t_i [3] per position 1 ≤ i ≤ r

$$(a_i, b_i, c_i, d_i) \leftarrow (1, b_i, c_i, d_i) / \sqrt{1 + b_i^2 + c_i^2, + d_i^2}$$

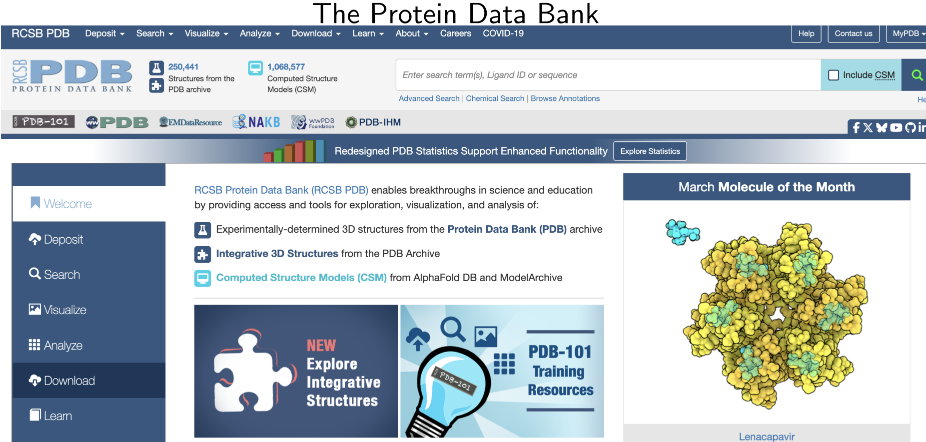
$$R_i = \begin{pmatrix} a_i^2 + b_i^2 - c_i^2 - d_i^2 & 2b_ic_i - 2a_id_i & 2b_id_i + 2a_ic_i \\ 2b_ic_i + 2a_id_i & a_i^2 - b_i^2 + c_i^2 - d_i^2 & 2c_id_i - 2a_ib_i \\ 2b_id_i - 2a_ic_i & 2c_id_i + 2a_ib_i & a_i^2 - b_i^2 - c_i^2 + d_i^2 \end{pmatrix}$$

$$T_i = (R_i, \vec{\mathbf{t}}_i)$$

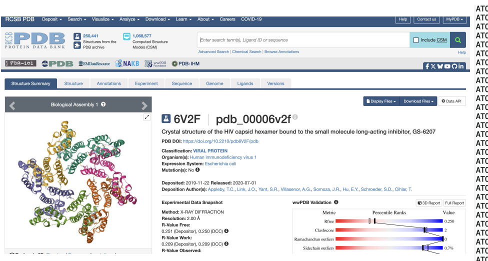
AF2 training

Labels: xyz coordenates of all heavy atoms in the 3D structure

The Protein Data Bank



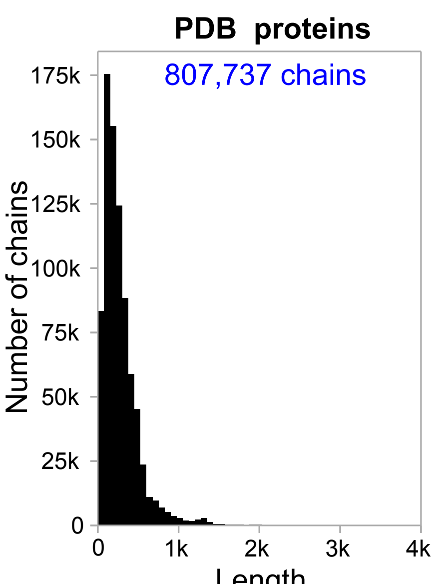
PDB entry: 6V2F



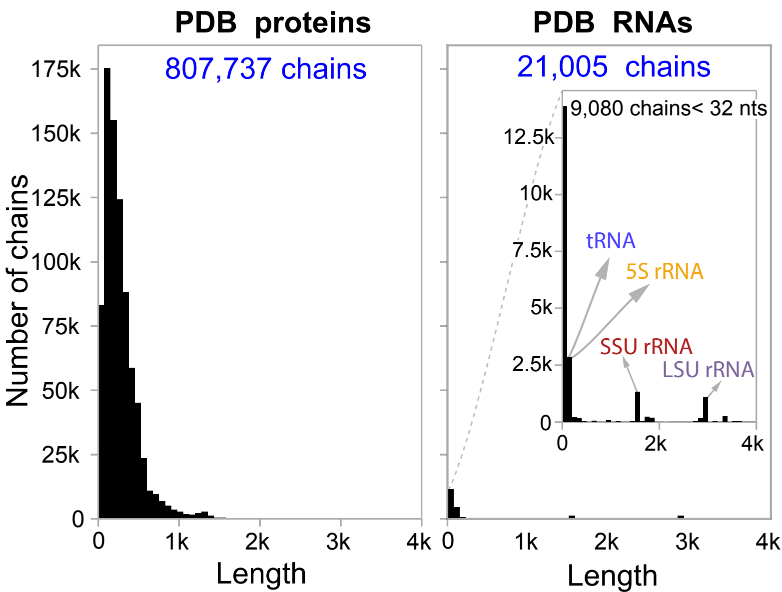
6V2F.mmcif

```
ATOM   1    N N   . PRO A 1 2  ? 15.01600 -6.21200 10.40700 1.000 23.70516 ? 1  PRO A N   1
ATOM   2    C CA  . PRO A 1 2  ? 15.72400 -5.12100 11.08300 1.000 24.88439 ? 1  PRO A CA  1
ATOM   3    C C   . PRO A 1 2  ? 15.53300 -5.12000 12.59400 1.000 27.70449 ? 1  PRO A C   1
ATOM   4    O O   . PRO A 1 2  ? 14.76000 -5.91700 13.12500 1.000 27.15048 ? 1  PRO A O   1
ATOM   5    C CB  A PRO A 1 2  ? 15.10100 -3.86000 10.46300 1.000 28.08434 ? 1  PRO A CB  1
ATOM   6    C CG  A PRO A 1 2  ? 13.81200 -4.32400 9.84400  1.000 31.53571 ? 1  PRO A CG  1
ATOM   7    C CD  A PRO A 1 2  ? 14.10300 -5.70700 9.37000  1.000 29.62990 ? 1  PRO A CD  1
ATOM   8    N N   . ILE A 1 3  ? 16.24500 -4.22700 13.27300 1.000 26.78383 ? 2  ILE A N   1
ATOM   9    C CA  . ILE A 1 3  ? 16.07600 -3.98900 14.70100 1.000 31.45883 ? 2  ILE A CA  1
ATOM   10   C C   . ILE A 1 3  ? 15.42700 -2.62200 14.85900 1.000 32.50870 ? 2  ILE A C   1
ATOM   11   O O   . ILE A 1 3  ? 15.98200 -1.60900 14.41600 1.000 36.22115 ? 2  ILE A O   1
ATOM   12   C CB  . ILE A 1 3  ? 17.41200 -4.05800 15.45700 1.000 29.32705 ? 2  ILE A CB  1
ATOM   13   C CG1 . ILE A 1 3  ? 18.14500 -5.36100 15.13200 1.000 31.84898 ? 2  ILE A CG1 1
ATOM   14   C CG2 . ILE A 1 3  ? 17.18500 -3.92200 16.95800 1.000 36.91482 ? 2  ILE A CG2 1
ATOM   15   C CD1 . ILE A 1 3  ? 17.55500 -6.58300 15.79800 1.000 31.74506 ? 2  ILE A CD1 1
ATOM   16   N N   . VAL A 1 4  ? 14.25100 -2.59200 15.47600 1.000 35.11352 ? 3  VAL A N   1
ATOM   17   C CA  . VAL A 1 4  ? 13.52800 -1.34300 15.66600 1.000 40.31307 ? 3  VAL A CA  1
ATOM   18   C C   . VAL A 1 4  ? 13.18100 -1.15000 17.13600 1.000 39.94655 ? 3  VAL A C   1
ATOM   19   O O   . VAL A 1 4  ? 13.18200 -2.10300 17.91400 1.000 41.95933 ? 3  VAL A O   1
ATOM   20   C CB  . VAL A 1 4  ? 12.25800 -1.29600 14.79800 1.000 40.74538 ? 3  VAL A CB  1
ATOM   21   C CG1 . VAL A 1 4  ? 12.62300 -1.30500 13.31900 1.000 40.23604 ? 3  VAL A CG1 1
ATOM   22   C CG2 . VAL A 1 4  ? 11.34200 -2.46300 15.13200 1.000 46.80582 ? 3  VAL A CG2 1
ATOM   23   N N   . MET A 1 11 ? 14.58600 -2.31000 21.90800 1.000 44.72441 ? 10 MET A N   1
ATOM   24   C CA  . MET A 1 11 ? 15.02700 -2.68200 20.56800 1.000 42.41279 ? 10 MET A CA  1
ATOM   25   C C   . MET A 1 11 ? 14.76900 -4.16300 20.31000 1.000 41.99084 ? 10 MET A C   1
ATOM   26   O O   . MET A 1 11 ? 15.33700 -5.02500 20.98100 1.000 40.16552 ? 10 MET A O   1
ATOM   27   C CB  . MET A 1 11 ? 16.51000 -2.36000 20.38300 1.000 42.16612 ? 10 MET A CB  1
ATOM   28   C CG  . MET A 1 11 ? 16.86800 -0.90800 20.66200 1.000 45.27050 ? 10 MET A CG  1
ATOM   29   S SD  . MET A 1 11 ? 15.98700 0.24500  19.59100 1.000 54.71942 ? 10 MET A SD  1
ATOM   30   C CE  . MET A 1 11 ? 16.68100 -0.17700 17.99500 1.000 46.37049 ? 10 MET A CE  1
```

AF2 Training data (2024)



AF2 Training data



AF2 losses

1. **evoformer losses**
   1.1 Distograms
   1.2 MSA loss
2. **structural losses**
   2.1 FAPE = Frame aligned position error
   2.2 Torsion angles loss
   2.3 Model confidence prediction (pLDDT) local distance difference test

Distogram loss

To associate the pair representation for the structural model

supervised training

Pair representation $z_{ij} + z_{ji}$

one-hot 64 distance bins covering 2-22 Amstrongs

$$p_{ij} = softmax(one - hot(z_{ij} + z_{ji}, 64bins))$$

$$y_{ij} = softmax(one - hot(distance_{ji}, 64bins))$$

$$Loss_{dist} = -\frac{1}{r^2} \sum_{b=1}^{64} y_{ij}^b \log p_{ij}^b$$

MSA loss

To encourage inter-sequence and phylogenetic relationships

self-supervised training

$$m_{si}[C_m] \rightarrow p_{si}[23]$$

where $23 = 20aa + unknown + gap + mask$

$y_{si}[23]$ are the one-hot ground-truth values in the msa.

$$Loss_{msa} = -\frac{1}{N_{masked}} \sum_{s,i \in mark} \sum_{c=1}^{23} y_{si}^{c} \log p_{si}^{c}$$

FAPE loss (Frame aligned point error)

Structural loss

predicted coordenates of a atom $\bar{x}_j$ under a predicted local frame T_i

$$T_i^{-1}\bar{x}_j = R_i^{-1}(\bar{x}_j - \bar{t}_i)$$

compared to the true ones

$$(T^{true})_i^{-1}\bar{x}_j^{true}$$

$$d_{ij} = ||T_i^{-1}\bar{x}_j - (T^{true})_i^{-1}\bar{x}_j^{true}||^2$$

AF2 Losses

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

Training tricks

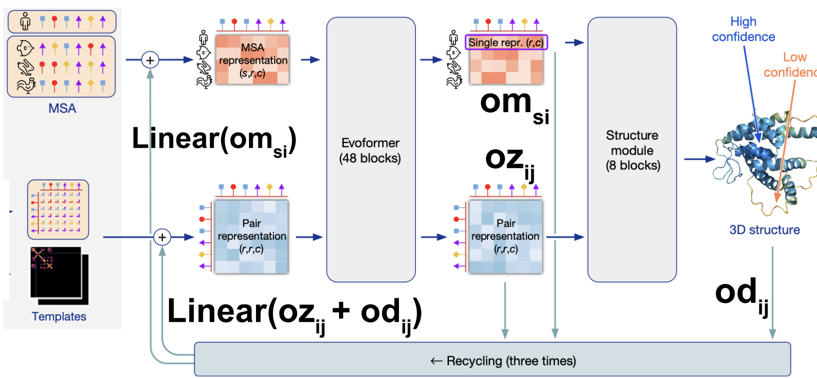
Recycling (3 times)

Iterative refinement using the whole network

Self-distillation

Training tricks

Iterative refinement using the whole network



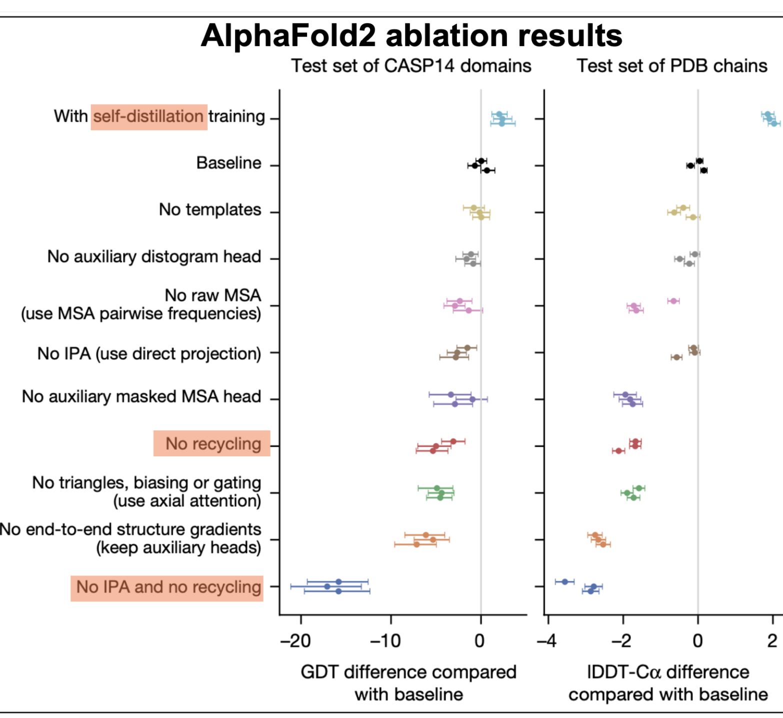
Training tricks

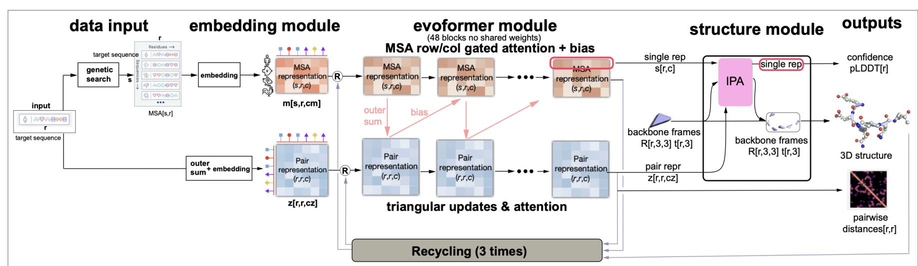
self-destillation

training with labeled and unlabeled data

- Train AF2 supervised on proteins with PDB structures (labeled data)
- Predict structures of proteins without known structures (unlabeled data)
- Retrain model from scratch on
  (proteins with known PDB structures) +
  (proteins with high-confidence predictions)

effective use of the unlabelled sequence data
considerably improves the accuracy of the resulting network


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
target sequence
genetic search
s
r
MSA[s,r]
input
target sequence
embedding
MSA representation (s,r,c)
m[s,r,cm]
R
outer sum
bias
single rep
s[r,c]
IPA
single rep
confidence
pLDDT[r]
backbone frames
R[r,3,3] t[r,3]
3D structure
outer sum + embedding
Pair representation (r,r,c)
z[r,r,cz]
pair repr
z[r,r,cz]
triangular updates & attention
pairwise distances[r,r]
Recycling (3 times)

Some important characteristics of AlphaFold2 are

- **Directly predicts the 3D coordinates of all heavy atoms** from the amino acid sequence using aligned sequences of homologs as inputs.
- Uses **alignments m[r,s] and pair representations z[r,r]** with different attention mechanisms for each of them, but treated jointly such that they exchange information with each other.
- **trained end-to-end**: all parameters are updated in the one single gradient descent optimization routine
- Uses many **different loss functions** some used at intermediate stages, and a final global loss function used to optimize all parameters of the model at once.
- Uses **self-distillation** to learn from unlabeled proteins. Self-distillation uses predictions of the model for further training.
- It is able to **estimate accuracy**
- Introduces a **new formalism for protein 3D structure prediction**: a rotation and translation to identify the location of each amino acid in a protein molecule. All residues are trivially initialized at the same location which evolves with training into an accurate description of all atoms of the amino acid chain.