RibonanzaNet3D: RNA Residue Sequence 3D Folding Prediction

Kristiyan Garchev Violeta Kastreva

Sofia University

Abstract

Predicting the three-dimensional (3D) structure of RNA from its nucleic acid sequence is a fundamental challenge in computational biology. Recent breakthroughs in protein structure prediction, notably AlphaFold [2–4], have inspired new approaches for protein 3D folding. However, RNA presents additional difficulties, including limited experimental data and distinct structural elements. In this work, inspired by the Stanford RNA 3D Folding Kaggle challenge [1], we present a simplified AlphaFold-like model for RNA 3D structure prediction, based on RibonanzaNet [8–10] tailored to the available data and constraints of RNA. Our model predicts coarse-grained residue coordinates (one per nucleotide) instead of all atomic positions, reducing complexity. It incorporates key neural network modules analogous to AlphaFold's architecture - including multiple sequence alignment processing, an Outer Product Mean module, triangular updates, and attention-based mechanisms - adapted for RNA. We detail the model's design and implementation, data loading pipeline with augmentation (including possible synthetic data usage in the future and knowledge distillation), and evaluation setup. Preliminary results indicate that leveraging as much data as possible (e.g., remaining "product" sequences for each RNA) and simplifying the representation can yield efficient learning despite the data scarcity. We provide analysis of the dataset (length distributions and sequence counts), and training and outline planned future experiments.

1 Introduction

RNA molecules fold into complex 3D structures that underpin their biological functions. Determining RNA 3D structure from sequence alone is notoriously challenging due to the flexibility of RNA backbones and relatively scarce structural data [6]. By contrast, protein structure prediction has seen tremendous progress, culminating in the success of AlphaFold for proteins [2–4]. AlphaFold demonstrated that a deep learning model can achieve atomic-level accuracy by leveraging evolutionary information from multiple sequence alignments (MSAs) and innovative neural network architectures. This raises the question: can similar techniques be applied to RNA?

There are important differences between protein and RNA folding. RNAs often form distinct secondary structures (stems, loops, junctions) and tertiary interactions (e.g. pseudoknots). Furthermore, the available experimentally solved RNA structures are only a tiny fraction of those for proteins in the Protein Data Bank (PDB) [14] - on the order of a few thousand RNA structures, versus hundreds of thousands of protein structures [5, 6]. These limitations have spurred development of RNA-specific prediction methods. Early approaches relied on physics-based simulations or fragment assembly for RNA folding, which are computationally intensive and less effective for larger RNAs. More recent methods incorporate machine learning; for example, Townshend *et al.* [5] used graph neural networks to rank candidate RNA structures, and other works have introduced transformer-based models for RNA [6–8, 10]. In 2024, a Kaggle community challenge was organized to accelerate RNA structure prediction (Stanford RNA 3D Folding challenge [1]), highlighting the growing interest in this problem. In the first edition, the goal was to predict the secondary structures (i.e. probabilities for base-pairing for each pair of residues) using chemical probing data, and in the current (second) edition of the competition the goal is to predict the 3D coordinates of the residues. Our project is motivated by these advances: although we did not participate in the competition, we draw inspiration from the state-of-the-art while designing a simplified yet effective model suited for our course project scope.

In this paper, we present an end-to-end RNA 3D structure prediction model "RibnanzaNet3D" influenced by RibonanzaNet's design [8–10](which is heavily inspired from AlphaFold [2–4]). Our model takes

an RNA sequence of residues, (possibly) several RNA sequences from evolutionary close species, and (possibly) the remaining product sequences, found experimentally with the main sequence, as inputs and outputs 3D coordinates for each residue (nucleotide) of the main RNA sequence. The goal is simplified compared to AlphaFold by predicting a single triplet of 3D coordinates per residue (for example, the position of the nucleotide's base or backbone centroid), whereas AlphaFold predicts the full atomic positions for every atom in amino acid residues [2]. This simplification reduces the output dimensionality and avoids dealing with internal atomic geometry, which is appropriate given the limited data. We also adapt the idea of using multiple sequence alignments (MSAs): whenever possible, we supply the model with not just the query RNA sequence but also additional *RNA sequences from evolutionary close species* related to the query (akin to homologous sequences in protein MSAs), aligned with the main sequence. The multiple sequence alignment with sequences from evolutionary close species helps in identifying stable residues, which have high probability to form a base-pair with another residue.

The contributions of our work are:

- We implement a neural network architecture for RNA folding that incorporates key modules from AlphaFold's *Evoformer* architecture (Outer Product Mean, Triangular Multiplicative Module, Triangular Attention) but tailored to RNA data and reduced complexity.
- We introduce a data pipeline that maximizes use of available data, including augmentations, synthetic sequences.
- We apply rigorous software engineering practices (object-oriented design and unit testing) to ensure each model component is reliable and extensible.
- We present initial analyses of the dataset and outline results, including distributions of sequence lengths and counts of related sequences, and discuss the model's performance qualitatively.

The remainder of the paper is organized as follows: Section 2 provides background on the AlphaFold architecture and related RNA prediction methods that informed our design. Section 3 details what data did we use, dataset statistics, & our data pipeline methodology. Section 4 presents our model architecture. Section 5 details our training procedure. Section 6 provides experimental results, analysis, and example predictions. Section 7 concludes the paper and suggests directions for future work.

2 Background and Related Work

Predicting RNA structure has been tackled with various approaches. Traditional *de novo* methods like FARFAR2 [24] and SimRNA [25] use fragment assembly or stochastic sampling to explore RNA conformations, but they require significant computational effort and often struggle with RNAs longer than about 100 nucleotides [7]. Template-based methods (e.g., ModeRNA, RNAComposer [26]) can produce accurate models when a closely related RNA structure is known [7], but such templates are rarely available for novel RNAs.

The success of AlphaFold [2–4] in protein folding has spurred interest in analogous deep learning approaches for RNA. AlphaFold's architecture introduced novel network components to capture pairwise relationships between residues and iteratively refine a structural representation. Key innovations include encoding an MSA of homologous sequences, computing outer products of embeddings to aggregate coevolutionary signals, and special attention-based modules that enforce geometric consistency (the triangular update blocks). While direct application of AlphaFold to RNA is not straightforward (due to fewer homologous sequences and different chemistry), the underlying ideas can be transferred.

Recent works have started to apply deep learning to RNA 3D prediction. Townshend *et al.* [5] applied geometric deep learning: using graph neural networks to predict distances between nucleotides and rank candidate structures. More recent models like RhoFold+ [6] leverage transformer encoders and even large-scale pretraining on RNA sequences (an RNA language model) to improve accuracy, achieving state-of-the-art results on RNA-Puzzles benchmarks. Another notable approach is trRosettaRNA [7], which takes inspiration from protein contact prediction (trRosetta) and uses transformer networks to predict inter-nucleotide geometry, followed by an energy minimization step to yield 3D coordinates. These methods demonstrate that with appropriate neural architectures and training strategies, the accuracy of RNA structure prediction can be significantly improved.

Despite these advances, challenges remain. RNA data is limited: there are orders of magnitude fewer known RNA structures than protein structures, and generating RNA structural data experimentally is difficult [6,7]. As a result, many models incorporate strategies like transfer learning or data augmentation. For instance, RhoFold+ addressed data scarcity by pretraining on millions of RNA sequences (without structures) and by predicting intermediate structural features like secondary structure and helix orientations to constrain the problem [6]. Another approach to augment data

is to use synthetic or simulated data. The Das Lab at Stanford, for example, has used crowd-sourced games (Eterna) and machine-generated RNA sequences to expand the training pool of RNA structures [**?**]. Knowledge distillation [17] can also be employed, wherein a larger or ensemble model's predictions on unlabeled sequences serve as training targets for a smaller model.

RibonanzaNet [8–10] represents a significant advancement in RNA structure prediction, becoming the de facto choice for models operating on RNA sequences due to its large-scale training on millions of chemically probed RNA sequences and its flexible, end-to-end architecture. The model's successor, RibonanzaNet2, utlilizes diffusion & boasts 100M parameters and incorporates experimental reactivity profiles, offering state-of-the-art performance on both secondary and tertiary RNA prediction tasks.

Importantly, RibonanzaNet's design is heavily inspired by AlphaFold [2], borrowing the paradigm of transformer-based architectures and coordinate prediction head modules. By coupling sequence encodings with geometric prediction blocks—similar to AlphaFold's structure — RibonanzaNet adeptly captures both local nucleotide relationships and global 3D folding patterns. This architecture reflects a convergence in methodology: it leverages AlphaFold's success in protein structure modeling to address RNA's unique structural challenges through a unified, differentiable pipeline.

In summary, the field is moving toward deep learning solutions that borrow from protein folding breakthroughs while introducing RNA-specific adaptations. Our project fits within this context: we implement a modular transformer-based model inspired by RibonanzaNet's network, but simplify certain aspects and incorporate as much input data as we can. In the next section, we describe the design of our model and how it implements these concepts.

3 Training Data and Data Pipeline

3.1 Experimental Dataset

For training and evaluating our model, we utilize a dataset derived from known RNA structures, provided in the competition's Data tab [1]. The core of our dataset consists of RNA sequences with corresponding 3D coordinates for each residue (nucleotide). These come from experimentally determined structures, from the Protein Data Bank [14]. Each RNA example in the dataset includes:

- The **main sequence**: the primary nucleotide sequence of the RNA of interest.
- (Optional) **MSA**: A variable number of additional RNA residue sequences evolutionary similar to the main one, all aligned in a Multiple Sequence alignment (MSA).
- (Optional) The **product sequences**: A variable number of residue sequences which are the products found in the experiment probing in which the main sequence 3D structure is found. It may include several copies of the main sequence. They do not have to be RNA sequences.
- The **target 3D coordinates**: the ground-truth 3D positions of each residue in the main sequence, given in a consistent coordinate frame. We represent each residue by a single point (e.g., at the atom that best represents the nucleotide's position in space, such as the phosphorus atom or a centroid) to simplify the prediction task. Also, for each sequence there can be multiple 3D positioning (stemmed from different probings, or multiple copy products in the same probing). Additionally, some of the nucleotides can lack coordinates for a specific ground truth, i.e. they can be missing.

The dataset comes divided into train, validation test dataset. Almost all data samples are in the train dataset (5379); only 12 data samples are in the validation dataset, and the test dataset is a copy of the validation dataset.

You can view the train dataset's main statistics in Figures 1 & 2. Out of all 3718038 residues in the train dataset, 41462 don't have ground truths (i.e. 1.12%). Out of all 5379 training samples, 3020 have MSA (i.e. 56.14%) All training data samples have only 1 ground truth, but the ones in the validation/test dataset can have multiple.

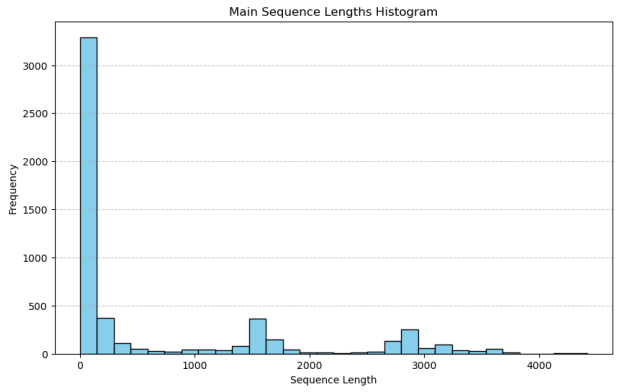
3.2 Synthetic Dataset

In the competition [1], there is also a link to a synthetic dataset by UW [15] generated by a model for known RNA sequences. It may be used to teach the current model stereochemistry in the early epochs. Each RNA example in the dataset includes:

- The **main RNA sequence** of residues.
- The **ground truth**: only 1 set of 3D coordinates for each residue in the main sequence.

The dataset has 446034 data samples.

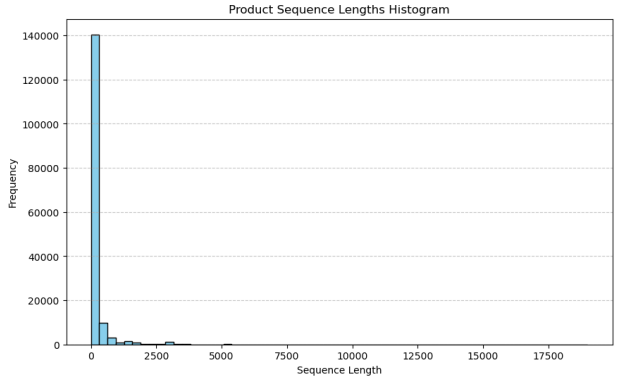
In our current progress, we have not utilized this dataset at all, as training on the main one takes long time alone. However, we have created an adapter for using it, and also an option for combining it with the main one, so, it can be used in the future right away.



(a) Main Sequences Length Distribution Histogram

Statistic	Value
Count	5379
Mean	691.21
Std Dev	1045.47
Min	3
25th Percentile	46
Median (50th Percentile)	94
75th Percentile	1418
Max	4417

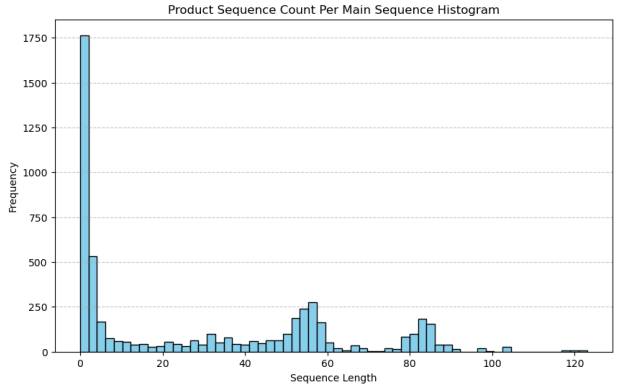
(b) Main Sequence Lengths Summary



(c) Product sequence Length Distribution Histogram

Statistic	Value
Count	159307
Mean	244.25
Std Dev	446.17
Min	1
25th Percentile	99
Median (50th Percentile)	141
75th Percentile	211
Max	19000

(d) Product Sequence Lengths Summary



(e) Product Sequence Count Per Main RNA Histogram

Statistic	Value
Count	5379
Mean	29.62
Std Dev	30.80
Min	0
25th Percentile	2
Median (50th Percentile)	14
75th Percentile	55
Max	123

(f) Product Sequence Counts Per Main Sequence Summary

Figure 1: Experimental train dataset sequence length statistics.

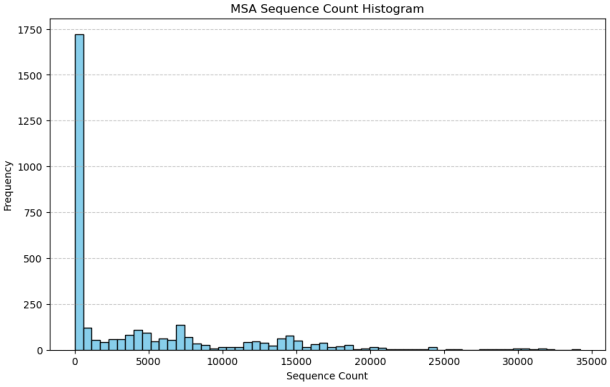
3.3 Data Preprocessing Pipeline

3.3.1 Residue Encoding

We encode each residue letter as an integer index. The main residues are: "A", "C", "G", "U", and also "-" for missing residue in MSAs. However, the product sequences may not be RNAs, so we encode all capital english letters to be able to handle the product sequences too. We don't tokenize more residues into 1 token, as we need to predict coordinates per residue.

3.3.2 MSA Processing

The advantage of the Multiple Alignment sequences (MSAs) is that all sequences in them are of the same length (i.e. aligned). The main practical problem is that MSAs have various number of sequences (Figure 2). To mitigate that, we don't use all MSA sequences, but rather we choose a fixed number of representatives. But this way we don't use all the available MSA data. Therefore, after we choose representatives, we aggregate information about the rest, so in



(a) MSA Sequence Count Distribution Histogram

Statistic	Value
Count	3390
Mean	4403.91
Std Dev	6329.53
Min	1
25th Percentile	17
Median (50th Percentile)	483
75th Percentile	6977
Max	34214

(b) MSA Sequence Counts Summary

Figure 2: Experimental train dataset MSA sequence count statistics.

a way, all MSA sequences have influence on the output. Also, to artificially increase our dataset size, we augment/preprocess our MSA dataset with the following techniques: MSA blocks removal, MSA clustering, MSA sequence mutation. All of this transformations are taken as an idea from the AlphaFold's Supplementary Notes [3].

3.3.2.1 MSA Blocks Removal. We first sort the MSA sequences by similarity to the query (i.e. main) sequence. There, the sequences are sorted by e-value, but since we didn't have access to that info for our dataset, and couldn't generate such accurate one, we sorted the sequences by Hamming distance to the query. This means that similar sequences are more likely to be adjacent in the MSA and continuous block deletions are more likely to generate diversity that removes whole branches of the phylogeny.

3.3.2.2 MSA clustering. The first choice for this procedure is to randomly chose a fixed-size subset of sequences without replacement when the MSA was too large (originally 128 sequences). This procedure has the clear downside that sequences not included in the random subset have no influence on the prediction.

The presented version is a modification of this procedure where we still choose a random subset of sequences without replacement to be representatives, but for each sequence in the full MSA we associate that sequence with the nearest sequence in the set of representatives (this is called "clustering" with random cluster centres though no attempt is made to ensure the cluster centres are well-distributed). To maintain the fixed-size and bounded cost properties, we use only the amino acid and deletion frequencies of all sequences associated with each representative sequence and provide these mini-profiles (called **MSA Profiles**) as extra features in addition to the representative sequence.

We roughly increase the number of input features of each representative sequence by a rather small factor (x10-20, in our experiment, but this can be changed) without increasing the number of representative sequences. This allows all sequences to have some influence on the final prediction, which seems desirable. Since this procedure achieves the goal of bounding computational cost, we have not experimented much with alternatives to the procedure.

In detail the MSA is clustered (grouped) using the following procedure:

- N_{clust} sequences are sampled randomly without replacement from all MSA sequences.
- Each of the remaining sequences is assigned to the clustering sequence with which it has minimal Hamming distance.
- For each cluster, the remaining sequences in it (i.e. all without the representative) are aggregated into fixed amount of information. The aggregated info we use: distribution of the main residues at that position; min, max, mean number of each main residue in the whole cluster.

3.3.2.3 MSA sequence mutation. For each residue in each representative of the MSA clustering we do the following:

- With 10% chance we replace the residue by random sample of the aggregated distribution in the previous step (the distribution is per residue per cluster).
- with 5% chance we replace the residue by random sample from the uniform distribution of the main residues.
- with 85% chance we leave it unchanged.

3.3.3 Data Augmentation

Given the scarcity of real RNA structures, we augmented the dataset in the following ways:

- The MSA sequence mutation from 3.3.2.3.

- With 50% chance we flip the whole data sample (i.e. all sequential properties, like main sequence, MSA, ground truth, etc.), as the forward order imposed on the sequential data doesn't have any meaning.

3.3.4 Data Batching

All sequential data is encoded into tensors and during batched ins combined in 1 tensor with padding. That causes us also to introduce masks, so we can differentiate between padded and real positions.

3.3.4.1 Proximity Batch Sampler. To reduce the amount of padding, that leads to unnecessary computation, we created a custom batch sampler which sorts all data samples by main sequence length during batching and picks batches of samples of roughly the same length. To prevent the batch sampler from creating the same batches each epoch, we shuffle the samples so that each sample is at most at a fixed distance away from its initial position (we call this fixed distance a **shuffle radius**). Furthermore, we randomly permute all batches.

3.3.4.2 Synthetically Balanced Proximity Batch Sampler. For training with synthetic & experimental dataset (although we left that for the future) we introduced a version of the Proximity batch sampler, which also balances the number of synthetic and real data samples in batch. More specifically, the number of synthetic data samples in the batch is a specific ratio from the whole batch. That ratio is predefined but may be dynamic, i.e. can be controlled with a scheduler. This is useful for training with synthetic data only in the early stages. The only caveat of this batch sampler is that the batch size may not be uniform, but the difference between the max and min batch size is at most 1, so it is not a big issue.

3.3.5 Data loading

We used a stateful data loader [27], and also the batch samplers we created (3.3.4) are also stateful, so the training can be resumed mid epoch. This is extremely useful, as training for 1 epoch takes a significant amount of time. Furthermore, the data loader uses asynchronous prefetching for faster loading of data.

Additionally, we've composed all the data-related objects into a single data manager. That includes datasets & token maps. During training, all the need data is issued through the data manager.

4 Model Architecture

Our model's architecture is heavily influenced by Ribonanza's design [8–10], which in turn is inspired by AlphaFold's modular architecture [2–4], comprising several stages that process the input sequences and iteratively refine representations of the RNA.

The key idea for such types of models (i.e. for this specific task) is that we don't just keep/update a sequence/MSA representation (i.e. 1 representation vector per token/residue), but also a pair representation (i.e. 1 representation vector per pair of tokens/residues). This is needed as RNAs/proteins have very intricate structure and pairs of residues can form a base-pairs, i.e. two nucleotides that are connected by hydrogen bonds, forming part of the double-stranded or folded structure of the molecule.

We first discuss the module versions of the models our model is based on, and then we discuss how we've utilized them.

4.1 AlphaFold's Architecture

Throughout this paper, by AlphaFold [2–4] we refer to AlphaFold2, which is the second generation of the model and introduces numerous changes.

4.1.0.1 Alphafold (2018, CASP13)

- Relied on a pipeline combining multiple neural networks for distance predictions and angle restraints.

- Predictions were converted into protein structures using traditional energy minimization tools like gradient descent.

- Used ensembles of models and heavy reliance on external folding engines.

- **Performance:** It won CASP13 but was still less accurate for hard targets and required additional computational steps to assemble the structure from distance maps.

4.1.0.2 Alphafold2 (2020, CASP14)

- Fully end-to-end neural network from sequence to 3D coordinates.

- Used a novel recycling mechanism that refines the structure iteratively within the model.

- Entire folding process is learned by the model—no external energy minimization needed.

- **Performance:** Dramatically outperformed all other methods at CASP14; reached near-experimental accuracy for many proteins; considered a breakthrough for protein structure prediction.

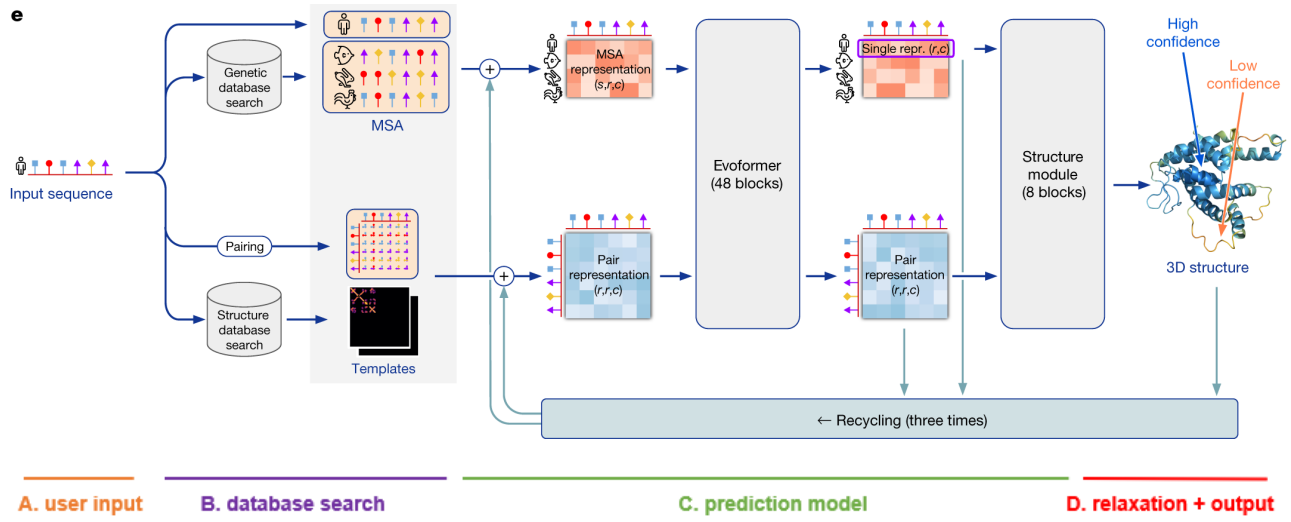


Figure 3: AlphaFold's main model architecture.

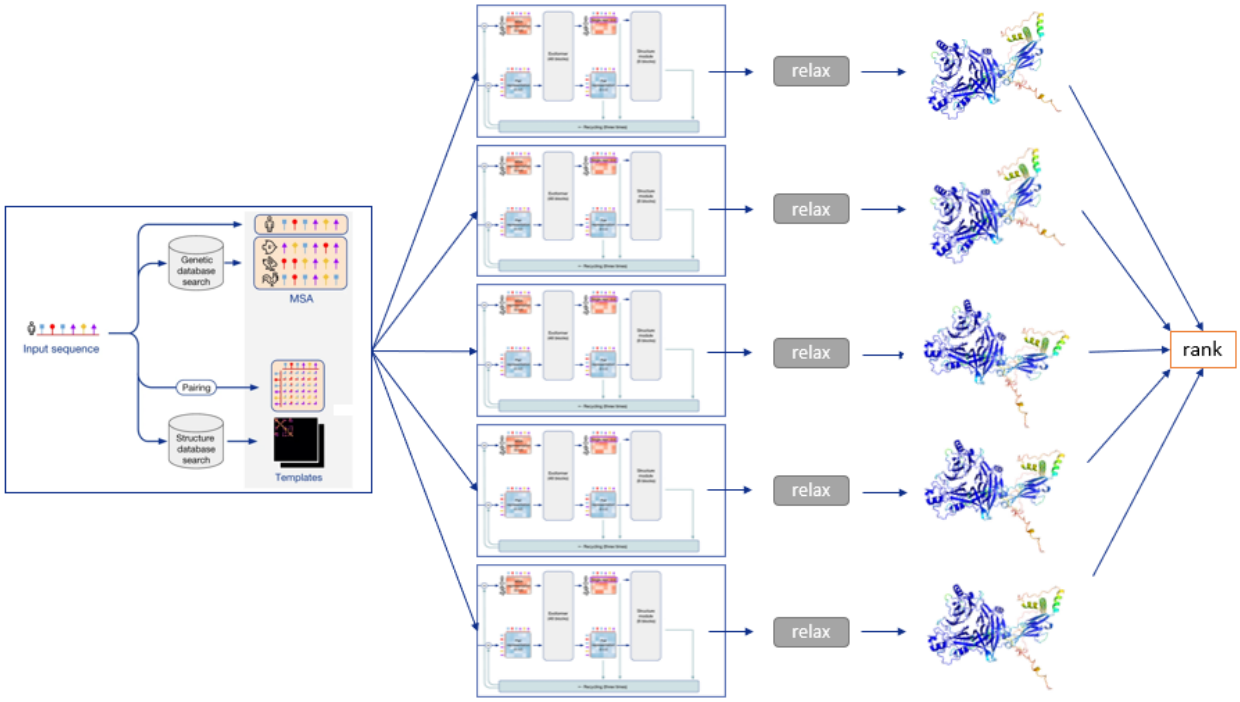


Figure 4: AlphaFold's full model architecture.

AlphaFold's architecture (Figures 3, 4 & 5) has 3 main parts:

- database search (Section 4.1.1)
- prediction model (Section 4.1.2)
- relaxation (Section 4.1.3)

4.1.1 Database search

For each sequence specified, a multiple sequence alignment (MSA) search is launched. This is done by first running JackHMMER on Mgnify (keeping the top 5,000 matches) and UniRef90 (keeping the top 10,000 matches), and then running HHBlits on UniClust30 + BFD (keeping all matches).

As AlphaFold bases itself largely on coevolutionary information, a qualitative and deep MSA is essential for good predictions. In their publication, DeepMind claims that they see a significant drop in accuracy for MSAs of less than 30 sequences. In case of protein complexes, MSAs are paired according to evolutionary distance (prokaryotes) or simply the ranking of the

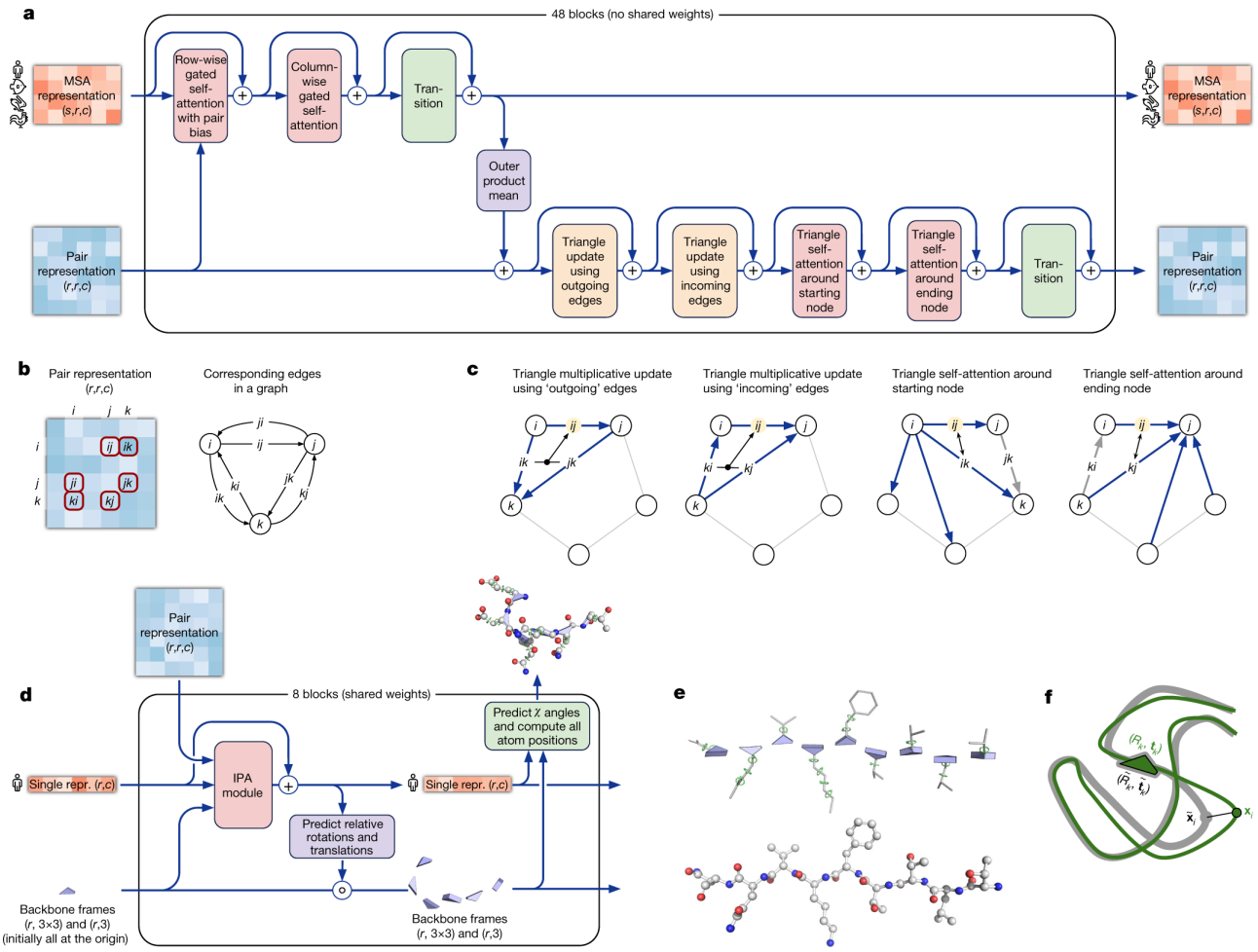


Figure 5: AlphaFold's Evoformer block architecture. **a:** Evoformer block. Arrows show the information flow. The shape of the arrays is shown in parentheses. **b:** The pair representation interpreted as directed edges in a graph. **c:** Triangle multiplicative update and triangle self-attention. The circles represent residues. Entries in the pair representation are illustrated as directed edges and in each diagram, the edge being updated is ij. **d:** Structure module including Invariant point attention (IPA) module. The single representation is a copy of the first row of the MSA representation. **e:** Residue gas: a representation of each residue as one free-floating rigid body for the backbone (blue triangles) and χ angles for the side chains (green circles). The corresponding atomic structure is shown below. **f:** Frame aligned point error (FAPE). Green: predicted structure; grey: true structure; (R_k, t_k): frames; x_i: atom positions.

matches (eukaryotes).

Finally, although less important than MSA depth, a template search is done. Using the MSA obtained from UniRef90, PDB70 is searched with HHSearch, only allowing templates before a specified date to be found. After discarding templates identical to (a subset of) the input sequence, the top 4 templates are chosen. These templates serve as a starting position for the prediction models.

4.1.2 Prediction model

The MSA and templates are given to five AlphaFold models. These all have the same network architecture, but different parameters following five independent training stages with different randomization seeds. Thus, they will predict slightly different 3-D structures.

The network architecture has two main parts. It consists of Evoformer blocks 4.1.4, which apply pairwise updates to numerical MSA representation and a 2-D pair representation, and Structure module blocks, which take care of the actual folding. These modules are repeated multiple times via a process called recycling, where the predicted 3-D structure is used as an input template for a new prediction iteration for further finetuning. By default, three recycling runs are done.

4.1.3 Relaxation

After each model predicts a 3-D structure, AMBER relaxation is done. Structures are ranked by the average predicted local distance difference test (pLDDT), found in the output of the prediction models. The pLDDT can be seen as a measure of local prediction confidence per position.

4.1.4 Evoformer block

As visualized in Figure 5, the sequence representation pass together through several blocks:

- Row-wise/Column-wise gated self-attention with pair bias (Section 4.1.5)
- Outer product mean (Section 4.1.6)
- Triangular Multiplicative Module for outgoing/ingoing edges (Section 4.1.7)
- Triangular self-attention(Section 4.1.11)
- Transitions (Section 4.1.9)

4.1.5 Row-wise/Column-wise gated self-attention with pair bias

Module architecture is shown on Figure 6.

MSA representations are processed with consecutive blocks of gated row-wise and column-wise self-attention blocks.

The row-wise version (Section 1.6.1 in [3]) builds attention weights for residue pairs and integrates the information from the pair representation as an additional bias term (line 3). This allows communication from the pair stack into the MSA stack to encourage consistency between them.

The column-wise version (Section 1.6.2 in [3]) lets the elements that belong to the same target residue exchange information.

4.1.6 Outer product mean

Module architecture is shown on Figure 7

The "Outer product mean" block transforms the MSA representation into an update for the pair representation (Section 1.6.4 in [3]). All MSA entries are linearly projected to a smaller dimension with two independent Linear transforms. The outer products of these vectors from two columns i and j are averaged over the sequences and projected to dimension c_z to obtain an update for entry ij in the pair representation. This is a memory intensive operation, as it requires constructing high-dimensional intermediate tensors.

4.1.7 Triangular Multiplicative Module for outgoing/ingoing edges

Module architecture is shown on Figure 8. Also, see Figure 5. Impleneted as in [19].

The triangular multiplicative update (Section 1.6.5 in [3]) updates the pair representation by combining information within each triangle of graph edges ij, ik, and jk. Each edge ij receives an update from the other two edges of all triangles, where it is involved. There are two symmetric versions, one for the "outgoing" edges and one for the "incoming" edges.

4.1.8 Triangular self-attention

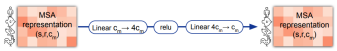
Module architecture is shown on Figure 9. Also, see Figure 5.

The triangular self-attention (Section 1.6.6 in [3]) updates the pair representation. The "starting node" version updates the edge ij with values from all edges that share the same starting node i, (i.e. all edges ik). The decision whether edge ij will receive an update from edge ik is not only determined by their query-key similarity (as in standard attention [95]), but also modulated by the information b_{jk} derived from the third edge jk of this triangle. Furthermore we extend the update with an additional gating g_{ij} derived from edge ij. The symmetric pair of this module operates on the edges around the ending node.

The Triangular Self-attention around the starting & ending node is very similar to the Row-wise & Column-wise gated self-attention. The main difference is that we pass the pair representation for both inputs. However, for the "ending node" version we still use attention bias, as opposed to the Column-wise gated self-attention. Another difference is the in the causal/padding masking: for the "ending node" version we use the same mask as for the "starting node" version, while we cannot do that with the Gated self-attention, as the input is doesn't have equal dimensions.

4.1.9 Transitions

There are 2 transitions on the end. Bot of them are fully connected layers with activations. The MSA representation's transition (Section 1.6.3 & 1.6.7 in [3]):

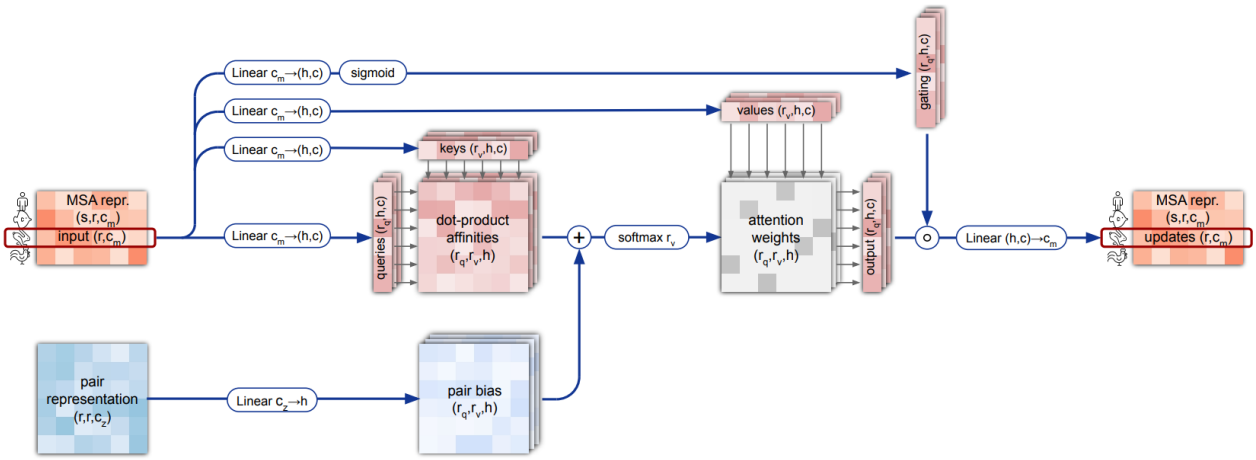


Supplementary Figure 4 | MSA transition layer. Dimensions: s: sequences, r: residues, c: channels.

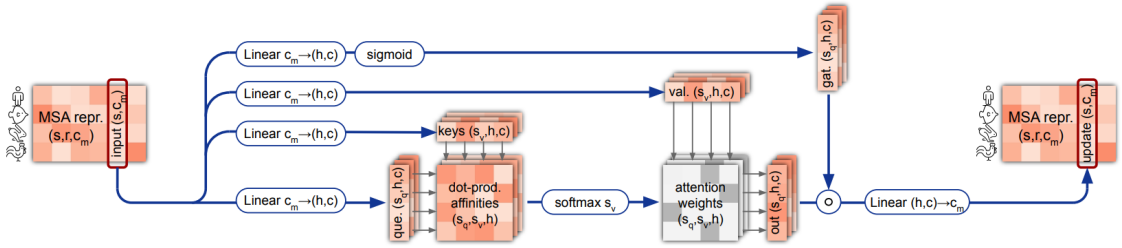
and the pair representation's transition is analogous.

4.1.10 Relative positional Encoding

To provide the network with information about the positions of residues in the chain, we also encode relative positional features (Section 1.5 in [3]) into the initial pair representations. Specifically, for a residue

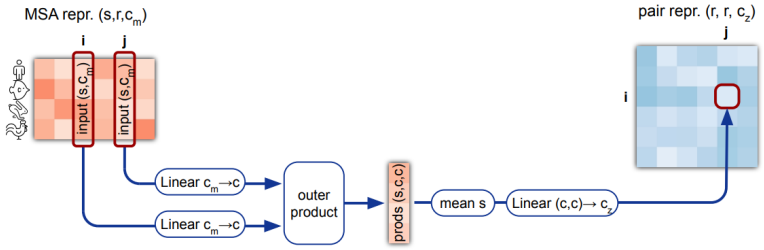


Supplementary Figure 2 | MSA row-wise gated self-attention with pair bias. Dimensions: s: sequences, r: residues, c: channels, h: heads.



Supplementary Figure 3 | MSA column-wise gated self-attention. Dimensions: s: sequences, r: residues, c: channels, h: heads.

Figure 6: Gated self-attention. (a) Row-wise with pair bias. (b) Column-wise.



Supplementary Figure 5 | Outer product mean. Dimensions: s: sequences, r: residues, c: channels.

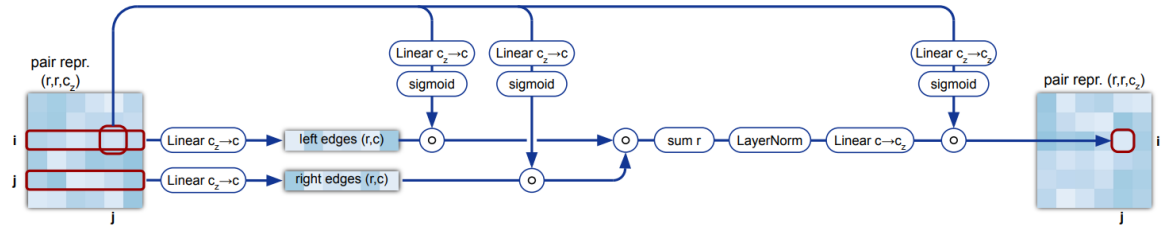
Figure 7: Outer product mean.

pair $i, j \in \{1...N_{res}\}$ we compute the clipped relative distance within a chain, encode it as a one-hot vector, and add this vector's linear projection to z_{ij} . Since we are clipping by the maximum value 32, any larger distances within the residue chain will not be distinguished by this feature. 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.

4.1.11 Invariant Point Attention (IPA)

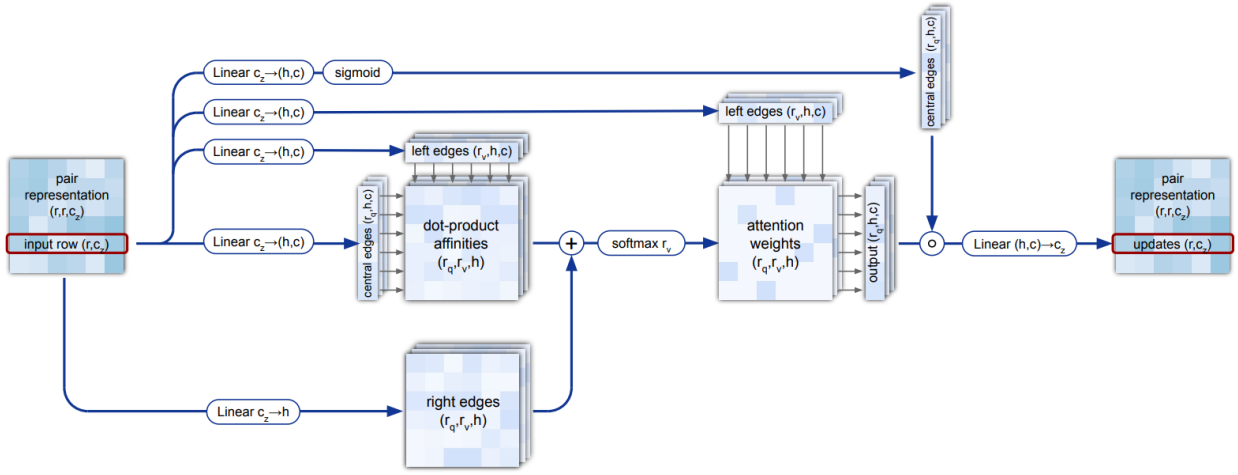
Module architecture is shown on Figure 10. Also, see Figure 5. This block is used in the Structure module.

Invariant Point Attention (IPA) (Section 1.8.2 in [3]) is a form of attention that acts on a set of frames



Supplementary Figure 6 | Triangular multiplicative update using "outgoing" edges. Dimensions: r: residues, c: channels.

Figure 8: Triangular multiplicative module for "outgoing" edges.



Supplementary Figure 7 | Triangular self-attention around starting node. Dimensions: r: residues, c: channels, h: heads

Figure 9: Triangular Self-attention around the starting node.

(parametrized as Euclidean transforms T_i) and is invariant under global Euclidean transformations T_{global} on said frames. We represent all the coordinates within IPA in nanometers; the choice of units affects the scaling of the point component of the attention affinities.

4.2 RibonanzaNet

RibonazaNet [8–10] is heavily based on AlphaFold [2]. It uses the modules introduced in AlphaFold. There are several versions of the Evoformer block for RibonanzaNet, called either Evoformer, Transformer Encoder, or ConvTransformer (See Figure 11).

The module architecture is shown on Figure 12.

4.3 RibonanzaNet3D

We introduce our version of both of those models called "RibonanzaNet3D". The architecture is very similar, as one can see on Figures 13 & 14.

We outline the major components here. An overview of the data flow is as follows (see Figure 13 for a schematic representation of the model architecture):

- **Input Embedding (Section 4.3.1):** The main RNA sequence and associated sequences are first transformed into initial embeddings. We use learnable positional encodings and per-nucleotide embeddings for this step.
- **Sequence extractor (Section 4.3.2):** Associated sequences (the MSA & MSA Profiles inputs) are embedded then concatenated and transitioned to d_{model} space.
- **Pair Representation Initialization:** We initialize a pairwise representation for each pair of positions in the main RNA (an $L \times L$ matrix of features) using the Outer Product Mean module, which combines information from the MSA representation per pair of residues.

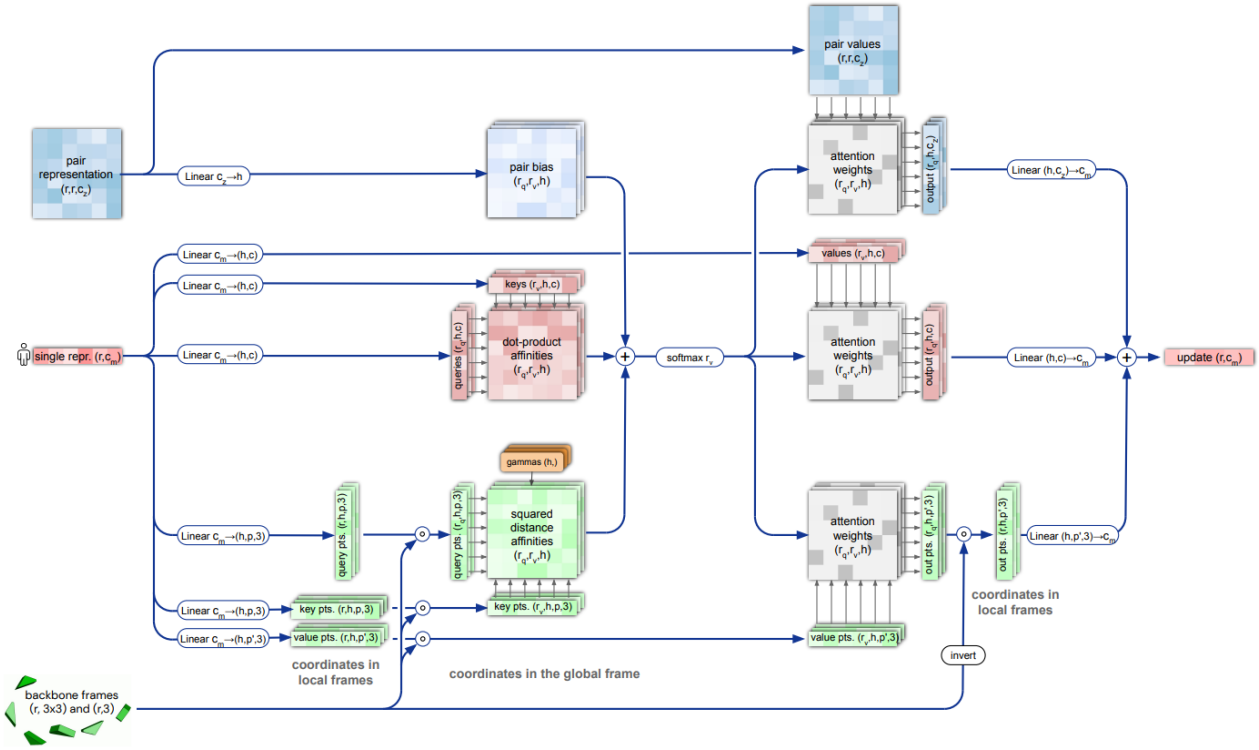


Figure 10: Invariant Point Attention (IPA).

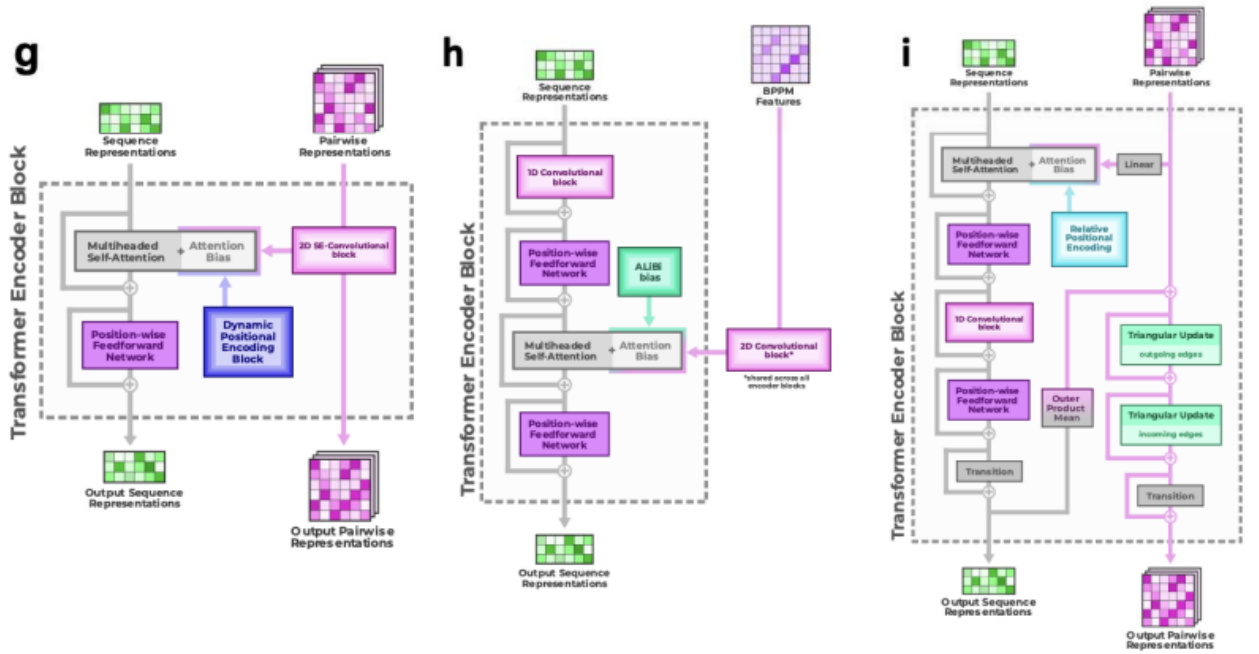


Figure 11: Different versions of the Transformer Block for RibonanzaNet.

- **Representation Update Stack:** We then apply multiple blocks of Representation updates. In each block, the main sequence representation, the pair representation, and the Product Sequences representation are updated. The block contains:

  1. Conv Transformer Block (Figure 14 **(b)**, (Section 4.3.3). It updates both the MSA & Pair representation.
  2. BiLSTM block for updating the Product Sequences representation.

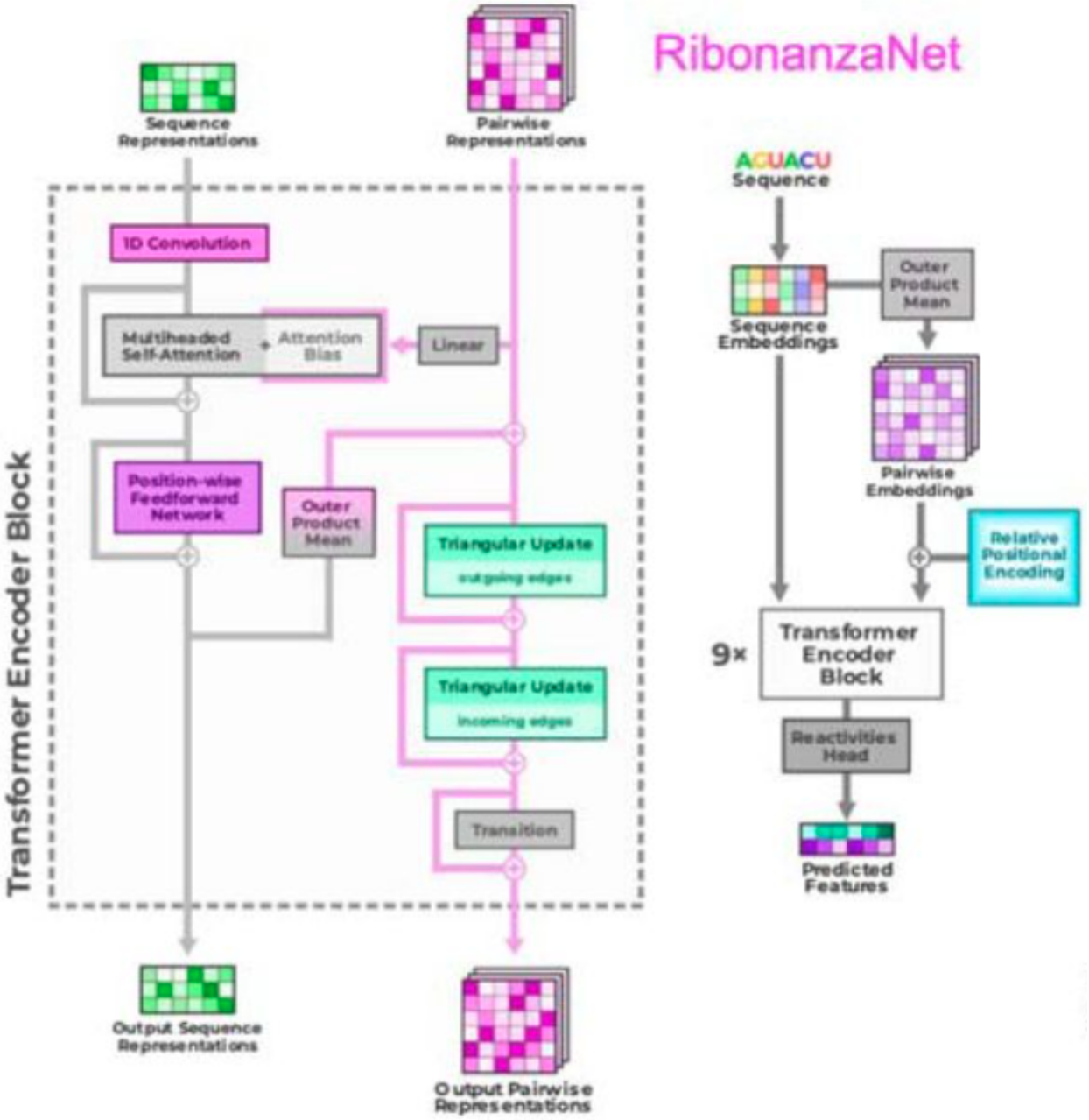


Figure 12: RibonanzaNet architecture.

3. MultiHead cross-attention that uses the Product Sequences representation (extracting the last vector in the sequences) to update the MSA representation.

We repeat such blocks N times (with separate weights) to iteratively refine the representations.

- **Regression Heads (Section 4.3.4):** Finally, we apply transitions on the MSA representation to predict both 3D coordinates of each residue and Confidence probability for each residue.

We note that our architecture uses fewer parameters than AlphaFold [2] and forgoes some complex elements such as explicit prediction of torsion angles or iterative "structure module" with inverted frames. Instead, it focuses on learning a mapping from sequence (and sequence correlations) to a coarse 3D structure.

Throughout the model, we use dropout and layer normalization to help generalize given the small dataset. The model is implemented in an object-oriented fashion: each logical component (embedding layer, sequence extractor, outer product module, triangular update, etc.) is a separate class or function. Also, we've updated the implementations of all blocks from AlphaFold to support multiple batching dimen-

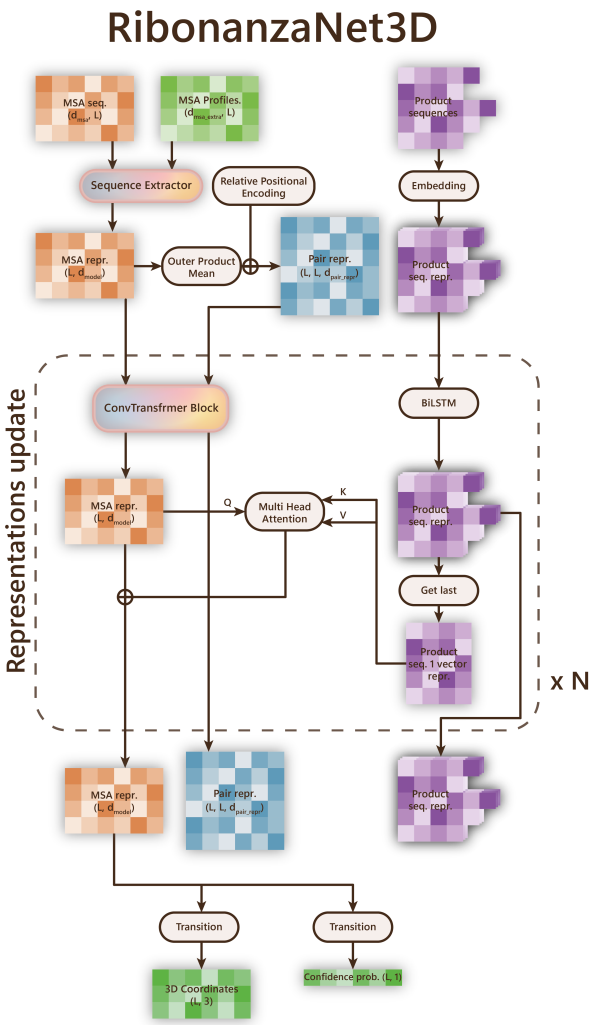


Figure 13: RibonanzaNet3D architecture

sions, as we believe this is the correct way to implement blocks. This modular design was influenced by the original code structure and greatly aided in testing and debugging.

Next, we describe key components of the architecture in detail.

4.3.1 Input Embeddings and Positional Encoding

We embed the main sequence of length L using a trainable embedding matrix of size $4 \times d_{model}$ (since there are 4 nucleotides A,C,G,U). Each nucleotide index is mapped to a d-dimensional vector. Similarly, for the N_{assoc} associated sequences (if $N_{\text{assoc}} > 0$), we embed each nucleotide in those sequences with the same embedding matrix.

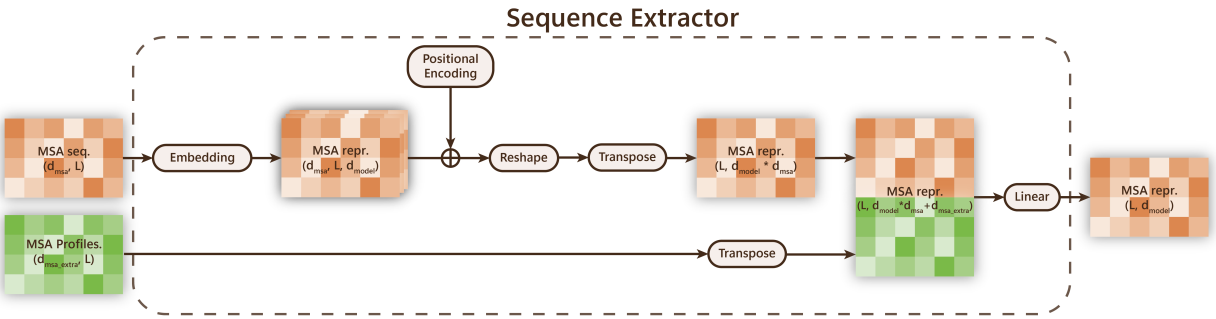
Positional encoding is crucial for transformers to be aware of sequence order. We add a sinusoidal positional encoding to the main sequence embeddings. For the product sequences, we embed their residues, but don't apply positional encoding, as they are processed by an LSTM, so their position is implied by the execution order.

4.3.2 Sequence extractor

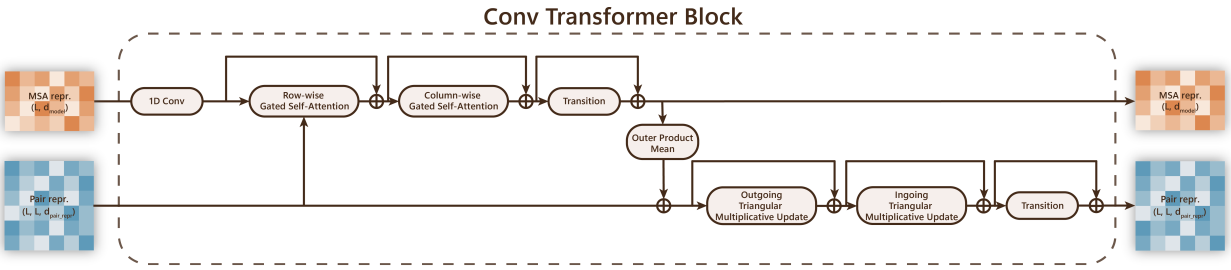
The sequence extractor (Figure 14 **(a)**) encodes the MSA sequences and the MSA profiles. A key difference from the previous architectures [2–4, 8–10] is that we don't store 1 vector per residue per MSA sequence in the MSA representation because of memory constraints (Section 5.3).

4.3.3 Conv Transformer

The module (Figure 14 **(b)**) updates both the MSA & Pair representation. It is really similar to AlphaFold's Evoformer(Section 4.1.4), but we add a convolution in the beginning, similar to RibonanzaNet



(a) Sequence Representation Extractor



(b) Conv Transformer Block

Figure 14: **(a)** Sequence Representation Extractor **(b)** Conv Transformer Block

(Section 12. Furthermore, we skip the Triangular Self-attention on the Pair representations, although we've implemented & incorporated it in our code base.

4.3.4 Output Regression Heads

The final part of the model is taking the processed MSA representation and predicting coordinates & confidence scores. We have a final sequence representation vector h_i^{final} for each position i. We map this to a 3D coordinate $y_i = (x_i, y_i, z_i)$ through a small network: specifically, a couple linear layers from d_{model} to 3. Also, we map the vector to 1 logit (also through a small fully connected network) and use it to create a confidence probability in our prediction.

5 Training Procedure

We trained our model using a supervised learning with backward propagation. For training, we used a local NVIDIA GeForce RTX 3070 Laptop GPU with 8GB VRAM.

5.1 Loss Function

5.1.1 TM Score

Although the metric in the Kaggle competition [1] is TM-score [20–22], this is not a differentiable metric and we cannot use it as loss function. We wanted to use it as validation metric, but unfortunately, the implementation was a bit tricky, so we've left the integration of a third party library for utilizing this metric for a future experiment.

5.1.2 Topological Loss

We call our loss Topological Loss, as it aims to measure the difference in the topological structure of the predicted 3D coordinates & the ground truth.

First, we align the prediction with the ground truth using the Kabsch-Umeyama algorithm [11, 12] (Section 5.1.3 & 5.1.4) to remove arbitrary rotations/translations/scaling. Throughout the computation, we mask out any padded positions (if the RNA is shorter than the max length) or missing ground truth positions. We used a multi-term loss:

- **RMSE loss:** We compute the Root Mean Square Error (RMSE) or the square root of the L2 loss between each predicted and actual coordinate. We add

square root for stability of the loss. We calculate:

$$L_{rmse} = \sqrt{\frac{1}{N_{batch\_size}} \sum_{i=1}^{N_{batch\_size}} ||y_i - \hat{y}_i||^2}$$

- **Pair Distances loss:** We also compute an L2 loss on pairwise distances: the difference between distance of predicted coordinates of different residues $||y_i - y_j||$ and ground truth distance $||\hat{y}_i - \hat{y}_j||$ for all i, j. This encourages correct overall geometric structural consistence. We calculate:

$$L_{pair\_distance} = \frac{1}{N_{batch\_size}}.$$

$$\sum_{i=1}^{N_{batch\_size}} \sum_{\substack{j=1 \\ j \neq i}}^{N_{batch\_size}} \left(||y_i - y_j|| - ||\hat{y}_i - \hat{y}_j||\right)^2$$

- **Folding angles loss:** We calculate absolute difference between the cosine similarity of the vectors $(y_{i+1} - y_i)$ and $y_{i-1} - y_i$ and the respective cosine similarity for $\hat{y}$. This encourages correct overall geometric structural consistence. We calculate:

$$C\left(i, y\right) = \frac{\langle y_{i+1} - y_i, y_{i-1} - y_i \rangle}{||y_{i+1} - y_i|| . ||y_{i-1} - y_i||}$$

$$L_{folding\_angles} = \frac{1}{N_{batch\_size}}.$$

$$\sum_{i=2}^{N_{batch\_size}-1} \left|C\left(i, y\right) - C\left(i, \hat{y}\right)\right|$$

- **Confidence loss:** We calculate the BinaryCrossentropy loss between the predicted confidence logits from the model and ebd $e^{-\frac{(y_{i+1}-y_i)^2}{Temp_{prob}}}$. This encourages The model to learn how well it has predicted the coordintes as well. We calculate:

$$L_{confidence} = \sum_{i=1}^{N_{batch\_size}} e^{-\frac{(y_{i+1}-y_i)^2}{Temp_{prob}}}$$

The weighted combination of these losses guides the model to learn both local and global structure. The combined loss looks like this:

$$L = \alpha_{rmse}.L_{rmse}$$

$$+\alpha_{pair\_distance}.L_{pair\_distance}$$

$$+\alpha_{folding\_angles}.L_{folding\_angles}$$

$$+\alpha_{confidence}.L_{confidence}$$

Also, not that there may be multiple ground truth targets for a data sample, so we calculate the loss for each one and take the minimum.

5.1.3 Kabsch Algorithm

The Kabsch algorithm [12] is a method used to compute the optimal rotation matrix that minimizes the root mean square deviation (RMSD) between two sets of paired points, typically used in structural biology to superimpose molecular structures. Given two sets of corresponding points, the algorithm first translates both point sets to a common centroid, then computes the covariance matrix of the centered coordinates. It performs a singular value decomposition (SVD) on this matrix to derive the rotation matrix that best aligns the two sets without introducing scaling or reflection, ensuring the result is a proper rotation. This makes the Kabsch algorithm a widely used and efficient solution for structural alignment problems.

5.1.4 Kabsch-Umeyama Algorithm

The Kabsch-Umeyama algorithm [11] extends the Kabsch algorithm by incorporating both rotation and uniform scaling to optimally align two sets of corresponding points. While the Kabsch algorithm finds the rotation that minimizes the root mean square deviation (RMSD) between the point sets, the Kabsch-Umeyama algorithm additionally computes a global scaling factor, making it suitable for cases where the point sets may differ in size due to uniform scaling. The algorithm follows similar steps: centering both point sets, computing the covariance matrix, and performing singular value decomposition (SVD), but it also calculates the optimal scale by comparing the variances of the two sets. This generalization makes the Kabsch-Umeyama algorithm applicable to a broader range of alignment problems, especially when both rotation and scaling need to be corrected simultaneously.

5.2 Optimizer

Ranger21 is a powerful optimizer built on AdamW (or optionally MadGrad), meticulously crafted by integrating multiple advanced techniques to enhance convergence, stability, and generalization. Its core features are:

- Adaptive Gradient Clipping (AGC)
- Gradient Centralization (GC)
- Positive–Negative Momentum
- Norm Loss
- Stable Weight Decay
- Linear Warm-up + Explore–Exploit Scheduling + Decay
- Lookahead
- Softplus Transformation

- Gradient Normalization

Adaptive Gradient Clipping (AGC): Dynamically clips gradients based on the ratio of gradient norm to parameter norm, stabilizing training, especially with small batches or high learning rates, without manual threshold tuning.

Gradient Centralization (GC): Subtracts the mean from gradients (per layer), acting as implicit regularization. This leads to smoother training curves and improved generalization.

Positive–Negative Momentum: Maintains separate momentum estimates for odd/even steps, then averages them to introduce parameter-dependent noise. This helps escape saddle points and converge to flatter minima.

Norm Loss: An alternative to standard weight decay, Norm Loss gently nudges weights toward unit-norm, offering more robust regularization and less sensitivity to hyper parameters.

Stable Weight Decay: Adjusts decay strength based on adaptive variance (the second moment), ensuring consistent regularization across training, avoiding overly aggressive decay in later stages.

Linear Warm-up + Explore–Exploit Scheduling + Decay: Starts with a linear ramp-up of the learning rate, transitions into an explore–exploit regime to find flat minima, then applies decay (e.g., Chebyshev or cosine) to finalize training.

Lookahead: Periodically updates a slower-moving shadow copy of the weights, smoothing the optimization trajectory and enhancing stability.

Softplus Transformation: Applies softplus to small adaptive variance terms to keep them significant, preventing numerical issues—this was added post-publication.

Gradient Normalization: Scales gradient norms for consistent step sizes across layers, further boosting stability. Also introduced after the original paper.

Together, these components collaboratively enhance AdamW's performance—delivering smoother loss curves, faster convergence, and stronger generalization across a broad range of challenging setups, including training BN-free ResNets. Furthermore, this is the optimizer usually used for training RibonanzaNet [8–10].

5.3 Memory Constraints & Optimizations

Initially we trained with the whole experimental dataset (Section 3.1 & Figures 1 & 2), which has some rather long sequences. This caused the model with the standard parameters from RibonanzaNet [10] not to fit on our GPU, so we needed to both decrease the number of parameters in the model, and utilize memory optimization techniques.

5.3.1 Activation checkpointing (Optimization)

Activation checkpointing — also known as gradient checkpointing — is a memory-saving technique used during neural network training wherein intermediate activations are selectively discarded during the forward pass and later recomputed on-the-fly during the backward pass, trading extra computation for reduced memory usage. Rather than storing all activations, only key inputs or outputs of defined segments are retained, and deeper activations are freed; during backpropagation, the forward pass for those segments is re-executed to regenerate necessary activations . This approach can drastically reduce peak GPU memory usage—often by 50–60%—at the expense of an approximate 15–33% increase in computation , enabling the training of larger models or batch sizes that otherwise wouldn't fit in memory.

We applied activation checkpoint on the most memory-intensive layers, like the Gated Self-attention, Multihead attention, etc.

5.3.2 Batch Size = 1 (Constraint)

To fit on the GPU we were forced to use batch size 1.

5.3.3 Gradient Accumulation (Optimization)

To remedy the smallest batch size we needed to use, we accumulate (avg) gradients across several batches, practically artificially increasing the batch size from the optimizer's point of view.

5.3.4 Mixed-Precision Training (Optimization)

To further reduce memory, we applied mixed-precision training.

Mixed-precision training accelerates deep learning by strategically using both 16-bit (FP16 or BF16) and 32-bit (FP32) floating-point formats—leveraging faster, lower-memory operations on modern GPUs while preserving accuracy using higher precision where necessary. During training, most computations (like forward/backward passes and tensor multiplications) are done in FP16 to reduce memory usage and improve

throughput, while critical components—such as model weights—are maintained in FP32, often alongside loss scaling techniques to prevent gradient underflow. Modern frameworks like PyTorch `torch.cuda.amp` or TensorFlow mixed-precision APIs delivering up to 3× speedups and 50% lower memory use, with accuracy comparable to full-precision models on supported hardware.

The only op that couldn't handle half precision was the SVD in the Kabsch algorithm.

5.3.5 Dataset filtration (Constraint)

To enable using a larger model, for the second training we filtered out data samples with main sequences with length $\geq$ 500. This reduced the dataset with 1628 data samples (i.e. with 30.27%). This is rather large percent but we increased the model's expressiveness, and sped up the training.

5.4 Training Implementation

We've implemented an end-to-end training, validation & testing pipeline. We use a Checkpoint Manager for saving last and best checkpoints during training and on error. Because we used stateful dataloader & batch-sampler, that allowed us to stop/continue training mid epoch, which was really important as 1 training epoch time was long (in order of hours).

Additionally, we tracked the training with Tensorboard logging (Figure 15).

6 Results and Analysis

We present here our preliminary results, including analysis of some qualitative evaluation of model predictions. All results are to be considered in the context of a prototype system; a thorough benchmark against existing methods is left for future work once the model is fully optimized and trained.

We have done 2 trainings (at least rather good trainings). The results are shown on Figure 17. We know that both models are far from being fully trained, but that is what we've achieved in the time we had.

The weights of the losses were picked in the following way: the bottom is the normalization factor, and the top is the actual weight. We decided to put almost all of the weight on the RMSE loss and the remaining is distributed on the others. This decision is based on early failed trainings.

Model Parameter Explanation:

- $\mathbf{d_{model}}$ - This is the dimensionality of the main sequence (MSA) embedding space. Every residue token—whether from the target sequence or its aligned homologs—is projected into a $\mathbf{d_{model}}$-dimensional vector before entering the transformer blocks. Larger $\mathbf{d_{model}}$ allows the model to capture richer features per residue, at the cost of more parameters and higher memory/computation requirements.

- $\mathbf{N_{heads}}$ - The number of attention heads in each multi-head self-attention layer (both row-wise and column-wise). Each head computes a separate projection of the $\mathbf{d_{model}}$-dimensional inputs into queries, keys, and values, enabling the model to attend to different aspects or "subspaces" of the representation in parallel. More heads can improve expressivity but increase computational overhead.

- $\mathbf{d_{pair\_repr}}$ - The channel size of the $\mathbf{L} \times \mathbf{L}$ pairwise representation, where $\mathbf{L}$ is sequence length. This matrix encodes relationships between every pair of residues (e.g. potential base-pairing, distance biases). A larger $\mathbf{d_{pair\_repr}}$ gives the model more capacity to represent complex inter-residue interactions, again at higher memory cost.

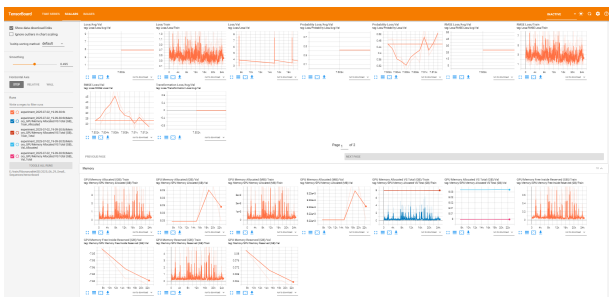
- $\mathbf{d_{lstm}}$ - The hidden-state dimensionality of each direction in the BiLSTM that processes the "product" sequences stream. After the Transformer-style blocks update MSA and pair features, these LSTM layers refine information coming from auxiliary experimental "product" sequences. Doubling $\mathbf{d_{lstm}}$ doubles the per-direction capacity for modeling sequence order in that stream.

- $\mathbf{N_{lstm\_layers}}$ - How many stacked BiLSTM layers are applied to the product-sequence embeddings. More layers deepen the recurrent processing of that auxiliary data, potentially capturing higher-order sequence patterns, but each layer adds parameters and latency.

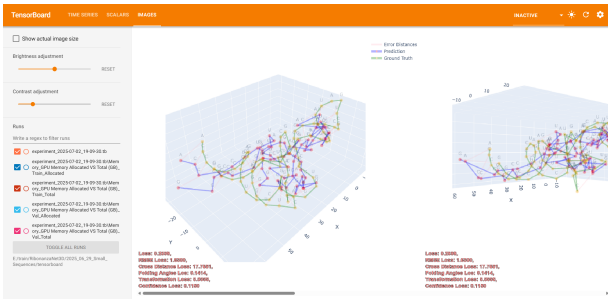
- $\mathbf{N_{blocks}}$ - The number of repeated Representation update blocks that alternately update the MSA, Pair, and Product sequence representations.

- $\mathbf{d_msa}$ - The number of representative sequences selected for MSA clustering. Also, used as internal dimension for the Outer product mean module.

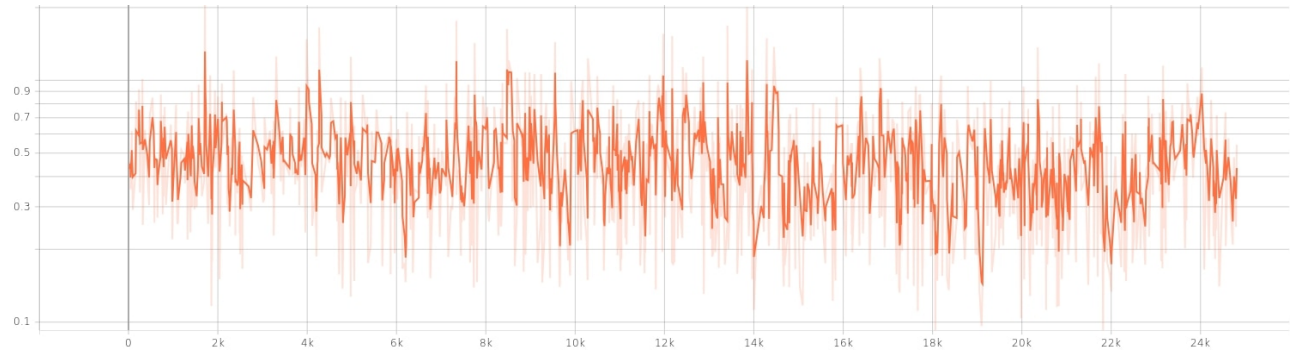
Impact of Model Capacity and Data Filtering: Training2 uses a larger model—doubling most embedding dimensions (e.g. from $d_{model} = 16$ to 64, d_{msa}=16 to 64), increasing blocks from 4 to 7, and adding an extra LSTM layer—while also filtering out sequences of length $\geq$ 500. This dramatically reduces the validation loss from 7.456 (Training1) to 4.729 (Training2), indicating the bigger model can fit the filtered data much more closely .



(a) Training Metrics



(b) Training Prediction Plot



(c) Train Loss

Figure 15: Tensorboard logging. **(a)** Training Metrics **(b)** Training Prediction Plot **(c)** Train Loss

Batching, Precision, and Optimization Settings: Both runs share small batch size (1 with gradient accumulation to 6 → 32), mixed precision, and the same learning rate schedule (lr=10^{-3} with warmup and decay). Yet Training2's complex architecture benefits greatly in fitting capacity — but its aggressive regularization (length filtering) and absence of stronger generalization techniques (e.g. heavier dropout or synthetic data augmentation) likely lead to the observed generalization gap. Future experiments should explore balancing model size with more diverse or augmented training data to reduce overfitting.

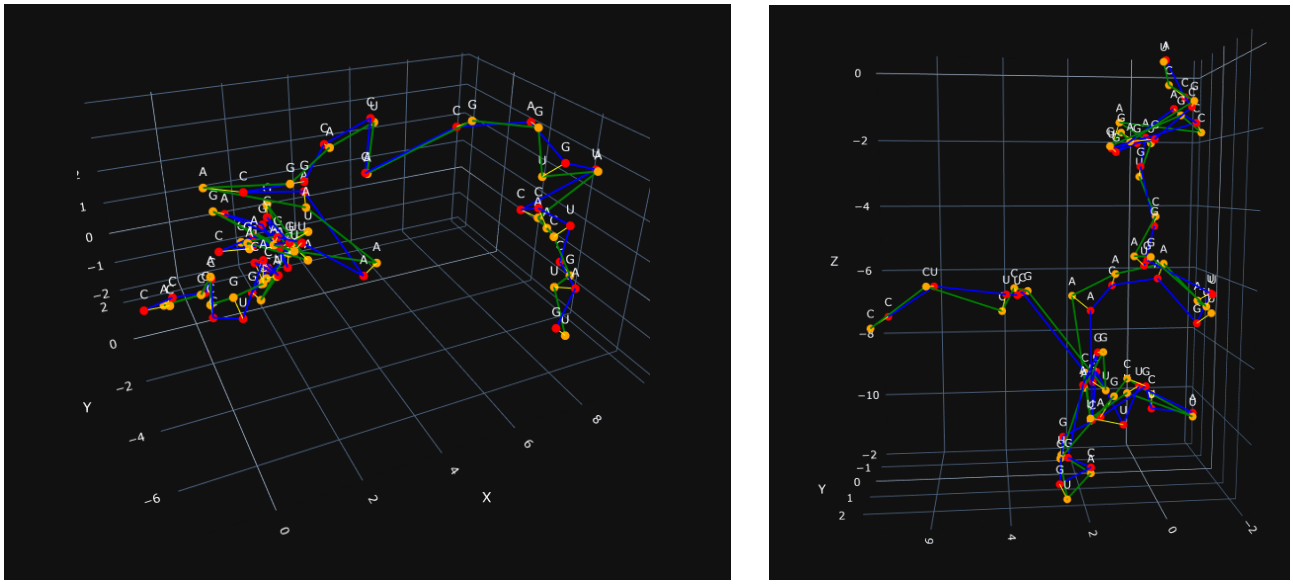
Example predictions: On Figure 16 you can see predictions from Training2.

7 Conclusion

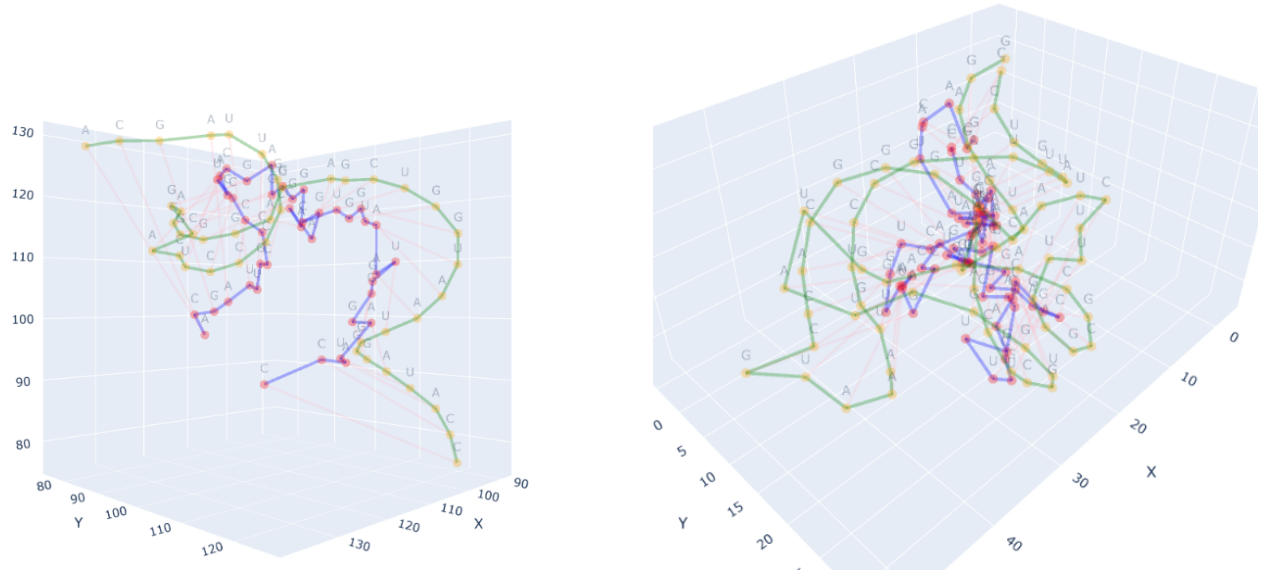
RibonanzaNet3D (Section 4.2) tackles the challenge of predicting RNA 3D residue-level structures directly from sequence by adapting the core ideas of AlphaFold [2–4] and RibonanzaNet [8–10] to the RNA domain. Unlike full atomic models, it predicts one coarse-grained coordinate per nucleotide, dramatically reducing output complexity. The model ingests a primary RNA sequence along with optional multiple-sequence alignments (MSAs) of homologs and "product" sequences from experimental assays, embedding each input stream via learnable residue embeddings and positional encodings before fusing them through specialized attention and outer-product modules inspired by AlphaFold's Evoformer, but simplified to accommodate scarce RNA data and limited GPU memory.

To overcome data scarcity, we assembled both an experimental dataset of 5K RNAs with known 3D coordinates and a larger synthetic set of 446K in silico–modeled structures, though current training uses only the experimental portion. We preprocess each example by encoding residues (including non-RNA "product" characters), clustering large MSAs to fixed-size representative sets plus frequency profiles, and applying augmentation such as random MSA block removals and residue mutations. During batching, a custom "proximity" sampler groups sequences of similar length to minimize padding, while an optional synthetic/experimental balancing sampler can control data mixture during training.

The core neural architecture — the "Representation Updates" stack—alternately refines a per-residue MSA representation and an L×L pairwise representation via gated self-attention, outer-product mean, and triangular update blocks, supplemented by BiLSTMs over product sequences and cross-attention between streams. After N such blocks, a final head projects each residue embedding to its 3D coordinate triplet and a confidence score. To train, we align predictions to ground truth via the Kabsch–Umeyama



(a) Good predictions from the model from Training2



(b) Bad predictions from the model from Training2

Figure 16: Example predictions from Training2. **(a)** Good predictions. **(b)** Bad predictions

algorithm, then optimize a weighted "topological" loss combining per-residue RMSE, all-pairs distance squared-error, and folding-angle deviations. Mixed-precision, activation checkpointing, gradient accumulation, and sequence-length filtering (length ≥ 500) are applied to fit the model on an 8GB GPU.

Preliminary results—evaluated on a small validation split—show that even this streamlined, coarse-grained approach can capture key aspects of RNA folding, such as secondary-structure formation and coarse tertiary contacts, albeit with room for quantitative improvement. We plan future work to incorporate richer geometric outputs, integrate thermodynamic priors, leverage synthetic and knowledge-distilled data more fully, and benchmark against state-of-the-art methods like RhoFold+ [6] and trRosettaRNA [7]. Our modular, object-oriented code base and data workflows provide a flexible foundation for these extensions.

Hyperparameter	Training 1	Training 2
# of Epochs	4	7
Batch Size	1	1
Acc. Grad. Batch Size	6	started 6, increased to 32 at 3^{rd} epoch
Is Mixed-Precision Training?	Yes	Yes
With filtered out sequences with length ≥ 500?	No	Yes
d_{model}	16	64
N_{heads}	8	8
d_{pair_repr}	4	16
d_{lstm}	4	16
N_{lstm_layers}	1	2
N_{blocks}	4	7
d_{msa}	16	64
Use Triangular self-attention?	No	No
Use Bidirectional LSTM?	Yes	Yes
Dropout	0.1	0.1
Rel. Pos. Encoding Clip value	16	16
α_{rmse}	$\frac{10.8}{65}$	$\frac{10.8}{65}$
$\alpha_{pair\_distances}$	$\frac{0.1}{4\times10^3}$	$\frac{0.1}{4\times10^3}$
$\alpha_{folding\_angles}$	$\frac{0.1}{1.1}$	$\frac{0.1}{1.1}$
$\alpha_{confidence}$	0.05	0.05
Probability temperature	10.0	10.0
	10^{-6}	10^{-6}
Learning rate	0.001	0.001
$\eta_{min}^{lr\_scheduler}$	10^{-6}	10^{-6}
$N_{warmup\_iterations}$	100	100
$Loss_{avg\_train}$	0.7791	0.4523
$Loss_{val}$	7.456	4.729

Figure 17: Training results

References

[1] Stanford University: "Stanford RNA 3D Folding", *Kaggle challenge*, 2025, [Link]

[2] J. Jumper, *et al.*: "Highly accurate protein structure prediction with AlphaFold", *Nature*, vol. 596, pp. 583–589, 2021, [Link]

[3] J. Jumper, *et al.*, "Highly accurate protein structure prediction with AlphaFold - Supplementary Information", 2022, [Link]

[4] Boris Burkov: "How does DeepMind AlphaFold2 work?", *Online Blog*, 2021, [Link]

[5] R. J. L. Townshend, *et al.*: "Geometric deep learning of RNA structure", *Science*, vol. 373, pp. 1047–1051, 2021, [Link]

[6] T. Shen, *et al.*: "Accurate RNA 3D structure prediction using a language model-based deep learning approach (RhoFold+)", *Nature Methods*, vol. 21, pp. 2287–2298, 2024, [Link]

[7] W. Wang, *et al.*: "trRosettaRNA: automated prediction of RNA 3D structure with transformer network", *Nature Communications*, vol. 14, Article 7266, 2023, [Link]

[8] Y. Li, *et al.*: "Ribonanza: deep learning of RNA structure through dual crowdsourcing," *bioRxiv*, (preprint), 2024, [Link]

[9] Moth: "Stanford RNA 3D — RibonanzaNet Explained", *Python Notebook*, 2025, [Link]

[10] Siddhantoon, *et al.*: "RibonanzaNet-2.0-DDPM - Explained", *Python Notebook*, 2025, [Link]

[11] Tuomas Siipola: "Aligning point patterns with Kabsch–Umeyama algorithm", *Online Blog*, 2021, [Link]

[12] Hunter Heidenreich: "Kabsch Algorithm: NumPy, PyTorch, TensorFlow, and JAX", *Online Blog*, [Link]

[13] @lessw2020: "Ranger21: integrating the latest deep learning components into a single optimizer", *Repository*, 2022, [Link]

[14] Walter Hamilton, Brookhaven National Laboratory *et al.*, "The Protein Data Bank", Online Database, since 1971, [Link]

[15] Andrew Favor, *et al.*: "UW Synthetic RNA Structures", *Synthetic Dataset*, 2025, [Link]

[16] T. Hamamsy, J.T. Morton, R. Blackwell, *et al.*: "Protein remote homology detection and structural alignment using deep learning.", *Nat Biotechnol* 42, 975–985, 2024, [Link]

[17] G. Hinton, O. Vinyals, & J. Dean: "Distilling the knowledge in a neural network", *NIPS Deep Learning and Representation Learning Workshop*, 2015, [Link]

[18] Aleksa Gordić: "ELI5: FlashAttention", *Online Blog*, 2023, [Link]

[19] @lucidrains: "Triangle Multiplicative Module - Pytorch Implementation", *Repository*, 2021, [Link]

[20] Yang Zhang, *et al.*: "Scoring function for automated assessment of protein structure template quality", 2004, [Link]

[21] Yang Zhang, *et al.*: "TM-align: a protein structure alignment algorithm based on the TM-score", *Nucleic Acids Research*, Volume 33, Issue 7, 1 April 2005, Pages 2302–2309, [Link]

[22] Chengxin Zhang, *et al.*: "US-align: universal structure alignments of proteins, nucleic acids, and macromolecular complexes", *Nat Methods* 19, 1109–1115, 2022, [Link]

[23] @pseeth: "AutoClip: Adaptive Gradient Clipping", *Repository*, 2020, [Link]

[24] A. M. Watkins, *et al.*: "FARFAR2: Improved De Novo Rosetta Prediction of Complex Global RNA Folds", Structure, Volume 28, Issue 8, 2020, Pages 963-976.e6, [Link]

[25] M. J. Boniecki, *et al.*: "SimRNA: a coarse-grained method for RNA folding simulations and 3D structure prediction", *Nucleic Acids Res*, 2016, [Link]

[26] M. Biesiada, *et al.*: "Automated RNA 3D Structure Prediction with RNAComposer", *Methods Mol Biol*, 2016, [Link]

[27] "Stateful DataLoader", *Pytorch documentation*, [Link]