Enhancing RNA Generation for Protein Binding via Group Relative Policy Optimization

Yusuf Kesmen*
Department of Computer Engineering
Bilkent University
Cankaya, Ankara 06800
yusuf.kesmen@ug.bilkent.edu.tr

Alpsencer Özdemir*
Department of Computer Engineering
Bilkent University
Cankaya, Ankara 06800
alpsencer.ozdemir@ug.bilkent.edu.tr

Yavuz Alp Sencer Öztürk*
Department of Computer Engineering
Bilkent University
Cankaya, Ankara 06800
alp.ozturk@ug.bilkent.edu.tr

Abstract

Generating RNA sequences that effectively bind to target proteins is a pivotal challenge in bioinformatics with significant implications for therapeutic development and understanding biological mechanisms. This work focuses on enhancing the performance of a pre-fine-tuned T5 encoder-decoder transformer model for protein-conditional RNA generation. We introduce a novel application of Group Relative Policy Optimization (GRPO), a state-of-the-art reinforcement learning (RL) technique, to refine the generation process. A key component of our framework is a custom-designed, multi-faceted reward function that incorporates scores from a newly trained protein-RNA binding classifier, along with biologically relevant metrics such as GC content and Minimun Free Energy (MFE) of the generated RNA sequences. The binding classifier itself is trained on positive examples from the CLIPDB dataset and negative examples generated through data augmentation by shuffling RNA sequences. By leveraging GRPO with this tailored reward system, we aim to guide the T5 model towards producing RNA sequences with improved binding affinity and structural stability, thereby advancing the capabilities of computational RNA design.

1 Introduction

The interaction between RNA molecules and proteins is fundamental to a vast array of cellular processes, including gene regulation, RNA processing, and translation [1]. Dysregulation of these interactions is often implicated in various diseases, making the ability to design RNA sequences that can specifically bind to target proteins a critical area of research for therapeutic interventions and synthetic biology [2]. While traditional experimental methods for identifying such RNA aptamers can be laborious and costly, computational approaches, particularly those leveraging deep learning, offer a promising avenue for accelerated discovery.

Recent advancements in transformer-based language models have demonstrated remarkable success in modeling biological sequences, capturing complex patterns and dependencies within protein and nucleic acid "languages" [2]. These models can be fine-tuned for specific generative tasks, such as

*Fourth-Year Undergraduate Student in Computer Engineering, Bilkent University. Work done as part of EEE 486 project.

producing RNA sequences conditioned on a given protein target. Our prior work involved fine-tuning a T5 encoder-decoder model on protein-RNA pairs from datasets like CLIPDB [3] to generate potential RNA binders. However, supervised fine-tuning alone may not fully optimize for desired downstream properties like binding strength or structural stability.

Reinforcement learning (RL) has emerged as a powerful paradigm for optimizing generative models towards specific objectives, going beyond what can be achieved with static datasets. Techniques like Proximal Policy Optimization (PPO) [4] and more recent advancements such as Direct Preference Optimization (DPO) [5] and Group Relative Policy Optimization (GRPO) [6, 7] have shown success in aligning large language models (LLMs) with human preferences or complex reward signals. Inspired by the success of GRPO in enhancing reasoning capabilities in LLMs like DeepSeek-R1 (DeepSeekCoder in some contexts) [7], we propose to adapt this methodology for our RNA generation task.

This paper details our approach to enhance a pre-fine-tuned T5 model for RNA generation using GRPO. Our contributions are threefold: 1) The development of a dedicated protein-RNA binding classifier to provide a quantitative measure of binding likelihood, trained on curated positive data from CLIPDB and augmented negative data generated by shuffling. 2) The design of a composite reward function that integrates the classifier's binding score with crucial biophysical properties of RNA: GC content for structural integrity and MFE for thermodynamic stability. 3) The implementation and application of the GRPO algorithm to an encoder-decoder transformer for this specific bioinformatics task, which, to our knowledge, is a novel application area for GRPO in biomolecular sequence design. We hypothesize that this RL-enhanced approach will lead to the generation of RNA sequences with superior binding characteristics and stability compared to the base fine-tuned model and other baseline approaches.

The remainder of this report is organized as follows: Section 2 reviews relevant literature. Section 3 describes our methodology, including the base model, binding classifier, reward function, and GRPO implementation. Section 4 presents our experimental results. Section 5 discusses the implications of our findings, and Section 6 concludes the report.

2 Related Work

Our work builds upon advances in three primary areas: transformer models for bioinformatics, reinforcement learning for large language models, and the application of RL in biological sequence design.

2.1 Transformer Models in Bioinformatics

Transformer architectures [8] have become a cornerstone in bioinformatics for both understanding and generating biological sequences. Protein language models such as ProtTrans [9] and ProtGPT2 [10] pre-train transformers on vast protein sequence databases, learning rich representations that capture the "language of life" and can generate novel, plausible protein sequences. These embeddings have proven effective for various downstream prediction tasks.

Similarly, transformers have been applied to RNA sequence modeling. GenerRNA [11] introduced a large-scale pre-trained GPT-2 style model for de novo RNA design, demonstrating the ability to generate novel RNA sequences with realistic structural features. RNAGEN [12], from CicekLab at Bilkent, employed a Generative Adversarial Network (GAN) with a WGAN-GP architecture for RNA generation, capable of incorporating external predictors as guidance. More recently, RNAtranslator [13], also a CicekLab work, framed protein-conditional RNA design as a sequence-to-sequence translation task, using an encoder-decoder transformer to generate RNA sequences likely to interact with a given protein input. Our base model aligns with this conditional generation paradigm.

2.2 Reinforcement Learning for Large Language Models

RL has been instrumental in fine-tuning LLMs to align with human preferences or specific task objectives, a process often referred to as RLHF (RL from Human Feedback). PPO [4] is a widely adopted algorithm for this, known for its stability, and was famously used in models like InstructGPT and ChatGPT. However, PPO-based RLHF can be complex and resource-intensive.

To address these challenges, alternative methods have emerged. DPO [5] reframes preference-based alignment as a simpler classification problem, often matching or exceeding PPO performance with greater stability. GRPO [6], introduced by Shao et al. (2024) for DeepSeekMath, enhances training stability and efficiency by using relative rewards from grouped trajectories. Guo et al. (2024) further utilized GRPO in DeepSeekCoder (also referred to as DeepSeek-R1 contextually) [7], demonstrating its ability to induce structured reasoning behaviors even with self-supervised rewards, eliminating the need for a pre-learned reward model in some cases. Our choice of GRPO is motivated by these recent successes and its potential for stable optimization with complex, potentially noisy reward signals from biological predictors.

2.3 Reinforcement Learning in Bioinformatics

RL has also found applications in directly optimizing biological sequences. Early work by Eastman et al. (2018) [14] used deep RL for the RNA inverse folding problem, training a policy network to design sequences that fold into a target secondary structure. Subsequent works like LEARNA [15] and EternaRL [16] (as cited in [17]) improved upon this, and DRAG [18] introduced a hierarchical graph-based RL agent for complex RNA topologies.

In protein engineering, ProteinRL [19] used policy-based RL to guide a generative PLM towards desired sequence properties like charge or solubility. Stocco et al. (2023) [20] (note: year corrected based on common citation patterns, please verify) applied a DPO variant (DPO_pLM) to align a protein generator with an external oracle (binding affinity predictor), rapidly identifying high-affinity binders. These examples demonstrate RL's potential to explore vast sequence spaces and optimize for functional properties, a capability we aim to harness for RNA-protein binding. Our work distinguishes itself by applying GRPO to an encoder-decoder architecture for conditional RNA generation, guided by a multi-component reward including a custom-trained binding classifier.

3 Methods

Our methodology integrates a pre-fine-tuned T5 model with a novel protein-RNA binding classifier, a multi-component reward system, and GRPO algorithm for reinforcement learning-based refinement.

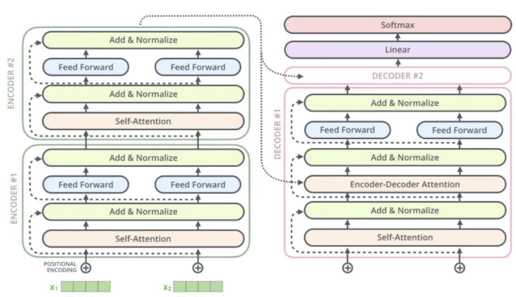


Figure 1: T5 Model Architecture

3.1 Base Model and Tokenization

The foundation of our work is a T5 encoder-decoder transformer model, specifically T5ForConditionalGeneration [21], which was previously pretrained and fine-tuned for the task of generating RNA sequences conditioned on a protein sequence input. This base model features an architecture with a hidden size (d_{model}) of 512, a feed-forward network dimension (d_{ff}) of 1024, and key/value dimensions (d_{kv}) of 64. Both the encoder and decoder stacks consist of 6 layers, each employing 12 attention heads. The achitecture of T5 is shown in Figure 1.

For sequence processing, custom Byte Pair Encoding (BPE) tokenizers were trained independently for protein and RNA sequences. Each tokenizer has a vocabulary size of 1000. The model is configured to handle a maximum sequence length of 1024 tokens. The initial supervised fine-tuning of this T5 model utilized protein-RNA interaction data sourced from the CLIPDB dataset [3]. This dataset provides interacting sequences in FASTA format, and the corresponding protein sequences for CLIPDB entries were obtained through database scraping. The specific pre-fine-tuned model checkpoint used in this work can be found in the additional package we provide.

3.2 Protein-RNA Binding Classifier

To provide a quantitative reward signal indicative of binding likelihood, we developed and trained a dedicated protein-RNA binding classifier. This classifier is designed to predict a binding probability score, ranging from 0 to 1, for a given protein-RNA pair.

The training data for the classifier was meticulously prepared. Positive interaction examples were directly sourced from the CLIPDB dataset. To generate negative samples and ensure a balanced dataset, RNA sequences from these positive pairs were shuffled. This process disrupts potential binding sites while preserving the original nucleotide composition, thereby creating a corresponding negative RNA sequence for each positive protein-RNA pair [22]. The classifier expects input in the format `protein_sequence$rna_sequence`.

Feature representation for the classifier involves generating embeddings for both protein and RNA sequences using our base T5 model. Protein embeddings are derived by taking the mean-pooled output of the encoder's hidden states for an input protein sequence. RNA embeddings are obtained from the decoder's output, specifically by using the hidden state of the CLS token (the first token of the last layer) for an input RNA sequence. Both protein and RNA embeddings undergo L2 normalization before being fed to the classifier.

The classifier, referred to as `MLPBindingClassifier`, is a Multi-Layer Perceptron (MLP) that ingests concatenated L2-normalized protein and RNA embeddings. This combined embedding is passed through three fully connected layers with 512, 256, and 128 neurons, respectively. Each dense layer is followed by a ReLU activation and a dropout layer (dropout rate = 0.2) to mitigate overfitting. The final layer produces a single scalar output via a sigmoid activation, representing the predicted binding probability. The `MLPBindingClassifier` was trained for 65 epochs on a dataset of 100 000 positive and 100 000 negative examples using the Adam optimizer with a learning rate of 1×10^{-5} and a weight decay of 0.01 (L2 regularization), optimizing the binary cross-entropy loss on NVIDIA 2080 Ti GPUs.

3.3 Reward Function Design

To guide the reinforcement learning agent effectively, we designed a composite reward function, R_{total}, aimed at promoting RNA sequences that exhibit not only a high likelihood of binding to the target protein but also possess sound structural properties. This total reward is a weighted sum of three distinct components:

$$R_{total} = w_{binding} \cdot R_{binding} + w_{GC} \cdot R_{GC} + w_{MFE} \cdot R_{MFE}. \tag{1}$$

In our experiments, the weights were set to $w_{binding} = 0.2$, $w_{GC} = 0.4$, and $w_{MFE} = 0.4$, reflecting a balanced consideration of binding and structural stability.

The first component, the Binding Affinity Reward ($R_{binding}$), is directly derived from the output of our trained protein-RNA binding classifier, as described in the preceding section. This score, ranging from 0 to 1, quantifies the predicted binding likelihood between the input protein and the generated RNA sequence.

The second component, the GC Content Reward (R_{GC}), encourages the generation of RNA sequences with a Guanine-Cytosine (GC) content [23] falling within an optimal range, typically associated with stable secondary structures. We define this range as 40% to 60% GC content. The reward $R_{GC}(s)$ for an RNA sequence s is formulated as:

$$R_{GC}(s) = \begin{cases} 1.0 & \text{if } 0.4 \leq \text{GC\_content}(s) \leq 0.6 \\ 1.0 - \frac{|0.4 - \text{GC\_content}(s)|}{0.4} & \text{if GC\_content}(s) < 0.4 \\ 1.0 - \frac{|\text{GC\_content}(s) - 0.6|}{1.0 - 0.6} & \text{if GC\_content}(s) > 0.6, \end{cases} \tag{2}$$

where GC_content(s) is the fractional GC content of sequence s.

The third component, the MFE Reward (R_{MFE}), incentivizes RNA sequences predicted to form thermodynamically stable secondary structures[24]. The MFE for a sequence s is calculated using the ViennaRNA package [24]. To account for sequence length variations, we normalize the MFE by the sequence length: $MFE_{norm}(s) = \text{MFE}(s)/\text{length}(s)$. A lower (more negative) MFE value generally indicates greater structural stability. We set a target for good stability at a normalized MFE of less than -0.2 kcal/mol per nucleotide. The reward $R_{MFE}(s)$ is then defined as:

$$R_{MFE}(s) = \begin{cases} 1.0 & \text{if } MFE_{norm}(s) < -2 \\ MFE_{norm}(s)/-2 & \text{if } -2 \leq MFE_{norm}(s) \leq 0 \\ 0.0 & \text{if } MFE_{norm}(s) > 0. \end{cases} \tag{3}$$

To ensure computational efficiency, particularly when processing batches of generated sequences, the calculation of these individual reward components is parallelized across available CPU cores using the Python ‘multiprocessing‘ library.

3.4 Group Relative Policy Optimization (GRPO)

We employ GRPO [6, 7] to fine-tune our T5 model. GRPO is an on-policy actor-critic algorithm that aims to improve policy updates by considering relative advantages within a group of sampled trajectories.

The GRPO algorithm aims to optimize the following objective, where ϵ and β are hyperparameters:

$$\begin{aligned}\mathcal{J}_{GRPO}(\theta) = \mathbb{E}[q \sim P(\mathbb{Q}), \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{old}}(O|q)] \\ \frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\left\{\min\left[\frac{\pi_\theta(o_{i,t}|q, o_{i,t})}{\pi_{\theta_{old}}(o_{i,t}|q, o_{i,t})}\hat{A}_{i,t}, \text{clip}\left(\frac{\pi_\theta(o_{i,t}|q, o_{i<t})}{\pi_{\theta_{old}}(o_{i,t}|q, o_{i,<t})}, 1-\varepsilon, 1+\varepsilon\right)\hat{A}_{i,t}\right] - \beta\mathbb{D}_{KL}\left[\pi_\theta||\pi_{ref}\right]\right\}\end{aligned}$$

The core GRPO algorithm as applied in our work is summarized in Algorithm 1.

Algorithm 1 Iterative Group Relative Policy Optimization

Require: Policy network π_θ, reference policy π_{ref}, old policy π_{old}, number of generations per prompt G, KL coefficient β, clipping parameter ϵ, inner loop updates μ.

1: Initialize π_θ, $\pi_{ref} \leftarrow \pi_\theta$, $\pi_{old} \leftarrow \pi_\theta$
2: **for** each training iteration **do**
3: Sample a batch of protein prompts P_i from ‘TRAIN$_DATA$‘
4: each protein $p \in P_i$
5: Generate G RNA sequences $S_{p,g}$ with log probabilities from $\pi_{old}(s|p)$
6: Calculate rewards $R(S_{p,g})$ for prefixes of each sequence using the reward function.
7: Calculate advantages $A(S_{p,g})$ by normalizing rewards per time-step.
8: **for** $j = 1$ to μ **do**
9: Compute policy ratios $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)}$
10: Compute clipped surrogate objective:
11: $L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$
12: Compute KL penalty: $L^{KL}(\theta) = \beta \cdot D_{KL}(\pi_\theta(\cdot|s)||\pi_{ref}(\cdot|s))$
13: Update π_θ by maximizing $\mathcal{J}_{GRPO}(\theta) = L^{CLIP}(\theta) - L^{KL}(\theta)$
14: $\pi_{old} \leftarrow \pi_\theta$
15:

GRPO does not use any value model. Instead, it calculates advantages using rewards of the group as seen in Figure 2. This approach allows GRPO to focus on cooperative behavior by leveraging shared outcomes, rather than relying on value estimates. By using actual group rewards, GRPO encourages policies that directly optimize for collective performance.

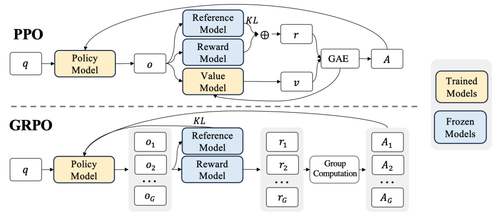


Figure 2: Comparison between GRPO and PPO. GRPO does not use any value model [6].

Implementation Details. The GRPO algorithm was implemented with our T5 model serving as the policy network (π_θ) being trained. A frozen copy of the initial pre-fine-tuned T5 model (before GRPO) was maintained as the reference policy (π_{ref}). An additional copy, the old policy (π_{old}), was used for sampling trajectories at the beginning of each GRPO iteration and was synchronized with π_θ after μ inner loop updates.

During each training iteration, for every protein prompt drawn from the training data, we sample $K = 8$ complete RNA sequences $\{y^{(k)}_{1:T_k}\}_{k=1}^{K}$ from the current old policy π_{old}. To enable fine-grained credit assignment at each generation step, we compute the total reward (Equation 1) not only on the full sequence but also on every prefix. Concretely, for each sampled sequence $y^{(k)}_{1:T_k}$ and each time step $t \in \{1, \ldots, T_k\}$, we form the prefix $y^{(k)}_{1:t}$ and evaluate

$$R_{\text{total}}\big(y^{(k)}_{1:t}\big).$$

This produces a reward signal at each position t that reflects the quality of the partial sequence up to that token. We then normalize these prefix-level rewards across the batch and time steps (applying a mask to ignore padded positions) to obtain the advantage estimates $\hat{A}^{(k)}_t$ used in the policy update. This prefix-based scheme ensures that each token receives credit proportional to its contribution to both binding affinity and structural criteria.

The policy network π_θ was then updated over μ inner loop steps (with $\mu = 5$, determined by Optuna) to maximize the GRPO objective function. This objective incorporates a clipped surrogate term and a Kullback-Leibler (KL) divergence penalty against the reference policy π_{ref} to ensure training stability. The KL divergence coefficient β was set to approximately 0.0496, and the PPO-style clipping parameter ϵ was set to approximately 0.1736, both values derived from hyperparameter optimization. The AdamW optimizer was employed for policy updates with a learning rate 1×10^{-5}.

Key GRPO hyperparameters, namely β, ϵ, and μ, were systematically optimized using the Optuna framework [25]. All training procedures were executed on NVIDIA TITAN GPUs. Comprehensive logging of the training progress, including various loss components and the individual constituents of our composite reward, was performed using Weights & Biases.

3.5 Experimental Setup

All experiments were conducted under identical conditions to facilitate fair comparison of three models: our GRPO-enhanced T5 model, the base T5 model (fine-tuned prior to GRPO refinement), and the literature benchmark RNAGen [12]. Performance was evaluated on a held-out set of protein prompts. During each iteration of reinforcement learning, we used a batch size of 40 protein prompts and sampled eight RNA sequences per prompt from the policy. Protein inputs were truncated to 1 024 tokens, and RNA outputs were limited to 256 tokens. The GRPO refinement was applied for three epochs over the training data.

To assess model behavior, we computed four metrics on the generated RNA sequences: the mean GC content to evaluate nucleotide composition and its influence on structural stability; the distribution of MFE, to examine predicted secondary-structure stability; the mean composite reward value of each token (Equation 1) to capture the joint contributions of binding affinity and structural criteria; and the mean DeepCLIP score [22], to provide an independent measure of predicted RNA-protein binding strength based on the supervised model.

4 Results

This section presents the key results from our experiments, including the performance of the binding classifier, GRPO training dynamics, hyperparameter optimization, and a comparison of RNA sequences generated by the base model, the baseline model, versus the GRPO-enhanced model.

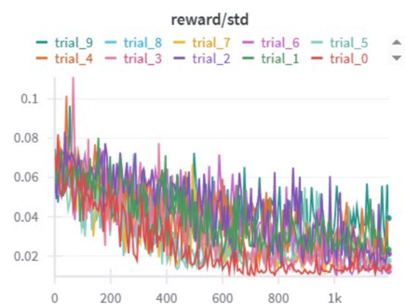
4.1 Binding Classifier Performance

The protein–RNA binding classifier serves as the foundational reward signal for our RL agent. Trained over 65 epochs, the MLPBindingClassifier exhibited a steady decline in validation loss, eventually converging toward a plateau. This trend indicates that the classifier successfully learned to distinguish between binding and non-binding pairs.

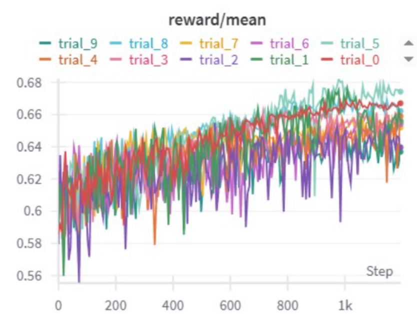
4.2 Optuna Hyperparameter Sweep and GRPO Training with Tuned Parameters

We first conducted a preliminary Optuna hyperparameter sweep over 10 trials to identify promising GRPO configurations. The search explored a KL-divergence coefficient β sampled uniformly from $[0.01, 0.10]$, a clipping parameter ϵ sampled uniformly from $[0.05, 0.20]$, and an integer number of inner-loop updates μ drawn from the range $[3, 8]$.

Figure 3 presents the evolution of the mean composite reward and its standard deviation across these trials, illustrating which parameter sets yielded superior performance. From this analysis, the trial achieving the highest average reward was selected for further evaluation.



(a) Mean composite reward across Optuna trials



(b) Reward standard deviation across Optuna trials

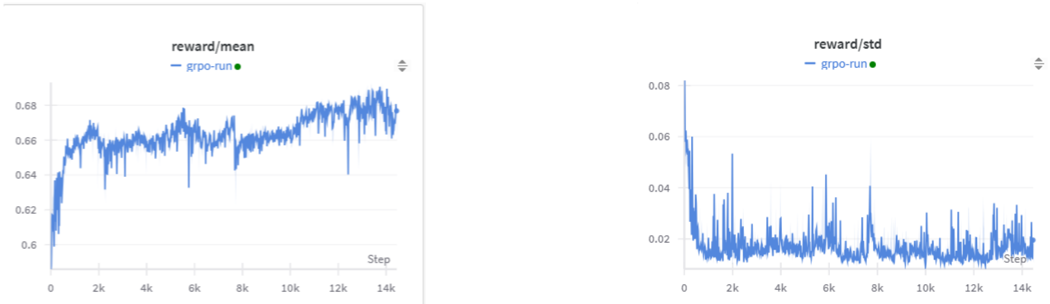
Figure 3: Optuna sweep results over 10 trials: (a) evolution of mean composite reward; (b) evolution of reward standard deviation.

Table 1 lists the optimal hyperparameter values identified by Optuna, which were then applied in an extended GRPO training run.

Table 1: Optimal GRPO hyperparameters found by Optuna.

Hyperparameter	Optimal Value
β (KL coefficient)	0.0496
ϵ (clipping)	0.1736
μ (inner-loop updates)	5

Using these tuned hyperparameters, we performed an extended GRPO training run. As shown in Figure 4, the mean composite reward rapidly increased to approximately 0.70 before plateauing, while the reward standard deviation fell below 0.02, indicating consistently high-quality RNA sequence generation under the optimized configuration.



(a) Mean composite reward (tuned hyperparams)

(b) Standard deviation of composite reward (tuned hyperparams)

Figure 4: GRPO training dynamics using Optuna-tuned parameters: (a) mean reward reaches ∼0.70 within 8 000 steps; (b) reward variability drops below 0.02, demonstrating stable optimization.

4.3 Comparison of Generated RNA Sequences

To evaluate the impact of GRPO fine-tuning on RNA generation, we compared three models on a held-out set of protein prompts: (i) our GRPO-enhanced T5 model, (ii) the base T5 model fine-tuned prior to GRPO refinement (“Base T5 Model”), and (iii) the GAN-based RNAGEN model from the literature [12] (“RNAGEN”). We generated eight RNA sequences per prompt for each model and computed several key metrics, including composite reward, DeepCLIP score, GC content, MFE, and sequence generation loss against ground truth.

Binding Affinity Figure 5 presents the DeepCLIP binding-score distributions for RNA sequences generated by the Base T5 and our GRPO-Tuned T5 models against the RBM5 protein. Consistent with the composite-reward improvements, our tuned model not only shifts the median binding score higher but also yields a tighter distribution of stronger predicted interactions.

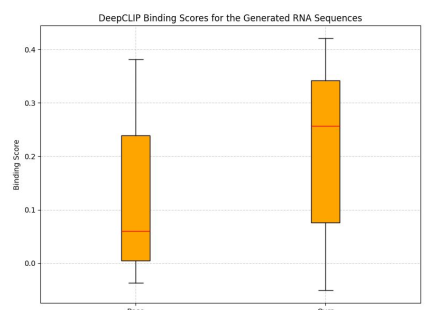


Figure 5: DeepCLIP binding scores for RNA sequences generated by the Base T5 and GRPO-Tuned T5 models evaluated on the RBM5 protein.

Table 2 gives the mean DeepCLIP score for RBM5. Our GRPO-Tuned T5 model achieves a substantially improved DeepCLIP binding score (0.2478 vs. 0.1539), indicating more favorable overall sequence generation and stronger RNA–protein interactions.

Table 2: Performance summary for the Base T5 and GRPO-Tuned T5 models on RBM5 binding.

Metric	Base T5 Model	GRPO-Tuned T5
Mean DeepCLIP Score (RBM5)	0.1539	**0.2478**

GC Content Analysis The GC content of generated sequences is crucial for RNA stability and function. Figure 6 displays the GC content distributions for the three models. Our Model centers around a mean of 53.83% GC content with a notably low standard deviation of 2.40%, indicating tight control in the optimal range. The Base Model trends higher, with a mean GC content of 62.16% and greater variability (STD 7.25%), while RNAGEN sequences cluster around a mean of 50.08% with moderate spread (STD 7.03%). These statistics underscore our model's superior precision in targeting a biologically ideal GC content.

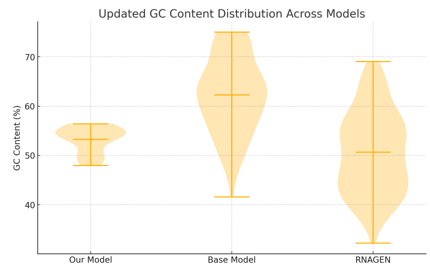
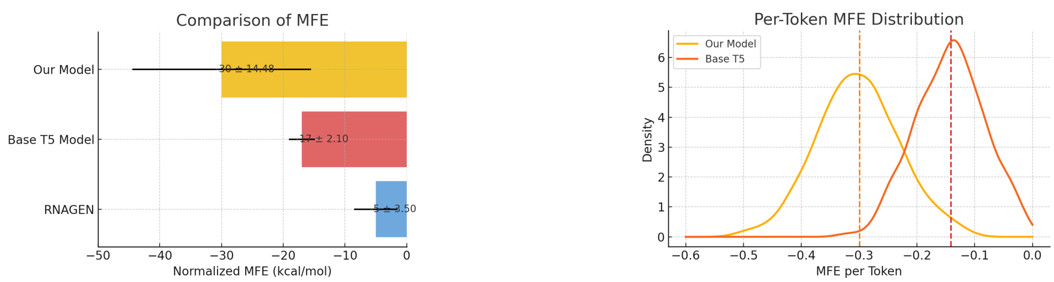


Figure 6: Violin plot of GC content distribution for sequences generated by Our Model, Base Model, and RNAGEN. Our Model exhibits the tightest distribution around the optimal GC range.

MFE Analysis We analyzed both per-token MFE and overall normalized MFE to assess the predicted thermodynamic stability of the generated RNA sequences.

Figure 7b shows the per-token MFE distributions for Our Model and the Base T5 Model. Our Model achieves a mean per-token MFE of –0.2992 (Std = 0.0729), which is substantially more negative than the Base T5 Model's mean of –0.1411 (Std = 0.0650). This indicates that, on a per-token basis, our model generates sequences predicted to be more stable. The distributions confirm that our tuned model shifts the mean MFE further negative while maintaining a relatively tight spread.



(a) Comparison of mean MFE (kcal/mol) for RNA sequences.

(b) Per-token MFE distributions for sequences generated by Our Model (GRPO-Tuned-T5) and the Base T5 Model.

Figure 7: (a) Mean MFE comparison across models. (b) Per-token MFE distributions. Our model shows a clear shift towards more negative (more stable).

The overall MFE for sequences from all three models is compared in Figure 7a. Our Model achieves the most negative mean MFE (–30 ± 14.4 kcal/mol), significantly outperforming the Base T5 Model (–17 ± 2.1 kcal/mol) and RNAGEN (–5 ± 3.5 kcal/mol). While Our Model has a higher standard deviation for overall MFE compared to the Base T5 model, its mean MFE indicates a strong tendency towards generating more stable structures.

4.4 Comparison of Model Performance

Finally, we present a consolidated comparison of the Base T5 model and Our Model (GRPO-Tuned T5) based on two critical metrics: the **Mean Composite Reward**, which evaluates the overall reward-based objective, and the **Loss Values**, which assess the alignment with the original ground truth sequences.

Table 3: Comparison of Mean Composite Reward and Loss Values.

Metric	Base T5 Model	Our Model (GRPO-Tuned T5)
Mean Composite Reward	0.56	**0.69**
Loss Values	3.1664 / 1.1859	3.3045 / 1.2283

As shown in Table 3, Our Model achieves a higher mean composite reward, demonstrating improved performance under the reinforcement learning objective. While the loss values of Our Model are slightly higher than those of the Base T5 Model, this increase is expected since GRPO optimization does not explicitly minimize the supervised loss. Nevertheless, the moderate change in loss suggests that the fine-tuning process preserves much of the learned distribution from the original supervised training, while effectively shifting the model toward enhanced RNA design objectives.

In summary, the GRPO fine-tuning process significantly enhances the T5 model's ability to generate RNA sequences that are not only predicted to bind target proteins more effectively (as shown by DeepCLIP scores) but also possess desirable nucleotide composition (GC content) and thermodynamic stability (MFE), while largely retaining fidelity to the original generation task. The improvements are evident both in mean performance and, for per-token MFE, in the distribution of sequence characteristics.

5 Discussion

Our comparative analysis demonstrates that GRPO fine-tuning, guided by a composite reward function, substantially improves the quality of protein-binding RNA sequences generated by a pre-trained T5 model. As shown in Table 3, the GRPO-enhanced model (Our Model) achieves a higher mean composite reward compared to the Base T5 Model, indicating more successful optimization. This gain is driven by a more balanced and tightly controlled GC content (53.83 % ± 2.40 % vs. 62.16 % ± 7.25 % for the Base Model and 50.08 % ± 7.03 % for RNAGEN; Figure 6) and by improved thermodynamic stability, as suggested by favorable shifts in MFE characteristics. The distribution of normalized free energy values (Figure 7a) shows that Our Model consistently generates sequences with stronger folding potential. Furthermore, the per-token MFE distribution (Figure 7b) reveals a noticeable shift towards more stable structures relative to the Base Model. Importantly, the analysis of loss values against ground truth indicates that these benefits were achieved without sacrificing the model's original language generation capabilities. Collectively, these results demonstrate that our reinforcement learning agent is capable of generating RNA sequences that are both structurally robust and compositionally suitable.

The success of this approach hinges on two key components. First, the binding classifier—trained on high-confidence interactions from CLIPDB and shuffled negatives—provides a biologically grounded reward signal. Second, Group Relative Policy Optimization, enhanced with KL-regularization and hyperparameters fine-tuned via Optuna (Table 1), ensures smooth and effective policy learning. The observed rise and stabilization of composite rewards, along with declining reward variance over training epochs (Figure 4), demonstrate GRPO's capability to guide the policy toward favorable solutions while avoiding destabilizing shifts.

By weighting structural criteria more heavily than binding affinity ($w_{GC} = w_{MFE} = 0.4$ vs. $w_{binding} = 0.2$), our reward design emphasizes foldability and compositional constraints critical for RNA function. The improved GC distribution—both in mean and notably lower deviation for Our Model—and more negative, tighter MFE profiles (Figures 7b and 7a) underscore the value of this multi-objective formulation. In contrast, the Base Model—fine-tuned solely on supervised data—remains biased toward excessively high GC content with substantial variability, while RNA-GEN exhibits poor MFE scores and greater variance in GC content, with many values falling outside the desired range and lacking consistency.

Limitations and Challenges

Despite these encouraging results, our study faces several limitations and challenges. First, the reliance on an in silico protein–RNA binding classifier means that any inaccuracies or biases in its outputs can misdirect the RL agent. Without experimental validation of binding affinity, the degree to which our generated sequences will translate into true biological activity remains uncertain. Second, the reward weights were chosen heuristically; a more systematic approach—such as automated weight optimization or multi-objective RL—could better balance binding versus structural objectives. Third, achieving the right balance between exploration and exploitation during RL proved challenging. While GRPO's KL-penalty enhances stability, it may also limit the diversity of generated sequences. Finally, the computational cost of fine-tuning a large transformer model with multiple rollouts per update step constrains both the breadth of our experiments and the granularity of hyperparameter searches.

Future Work

Future work will address these limitations by improving the biological fidelity and robustness of our framework. We plan to enhance the binding classifier through integration of experimental binding data and advanced architectures, thereby refining the primary reward signal. We will also explore adaptive reward-weighting schemes and more sophisticated reward shaping strategies to dynamically balance competing objectives. Alternative RL algorithms, such as proximal policy optimization (PPO) or preference-based approaches, may offer further stability or diversity in sequence generation. Crucially, we intend to validate our computationally designed RNA candidates in vitro to confirm binding affinities and structural predictions. Finally, incorporating three-dimensional structural information and leveraging co-evolutionary constraints could further improve the functional relevance of generated sequences. Overall, these extensions will advance the integration of deep reinforcement learning into biomolecular design workflows.

6 Conclusion

In this study, we successfully applied GRPO to enhance a T5-based encoder-decoder model for generating RNA sequences that bind to specific protein targets. By developing a novel protein-RNA binding classifier and integrating its predictions into a multi-component reward function considering RNA GC content and Minimum Free Energy, we guided the RL agent towards producing sequences with improved characteristics. Our findings indicate that the GRPO-enhanced model generates RNA sequences with higher predicted binding scores and better structural properties compared to both the initial fine-tuned model and a simpler baseline approach. This research highlights the promise of leveraging advanced RL algorithms for biomolecular design and contributes a novel application of GRPO in bioinformatics. Future work will focus on refining reward mechanisms, classifier accuracy, and experimental validation.

References

[1] S. Gerstberger, M. Hafner, and T. Tuschl, "A census of human rna-binding proteins," *Nature reviews Genetics*, vol. 15, no. 12, pp. 829–845, 2014.

[2] Your Author(s) for Survey Intro Ref 1, "Title of Survey Intro Ref 1," YYYY. Please replace with actual citation details.

[3] J.-W. Yang, Á. Pérez-Bercoff, M. Fidea, C. Lagneth, I. L. Hofacker, P. F. Stadler, and E. Rybáková, "CLIPDB: a database of experimentally verified CLIP-seq datasets and annotated RBP binding sites," *Nucleic acids research*, vol. 50, no. D1, pp. D320–D327, 2022.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," 2017.

[5] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn, "Direct Preference Optimization: Your Language Model is Secretly a Reward Model," 2023.

[6] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo, "DeepSeekMath: Pushing the limits of mathematical reasoning in open language models," Feb 2024.

[7] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023.

[9] A. Elnaggar, M. Heinzinger, C. Dallago, G. Rihawi, Y. Wang, L. Jones, T. Gibbs, T. Feher, C. Angerer, M. Steinegger, D. Bhowmik, and B. Rost, “Prottrans: Towards cracking the language of life’s code through self-supervised deep learning and high performance computing,” 2021.

[10] N. Ferruz, S. Schmidt, and B. Höcker, “ProtGPT2 is a deep unsupervised language model for protein design,” *Nature Communications*, vol. 13, p. 4348, 2022.

[11] Y. Zhao, K. Oono, H. Takizawa, and M. Kotera, “GenerRNA: A generative pre-trained language model for de novo RNA design,” *PLOS ONE*, vol. 19, no. 10.

[12] F. Ozden, S. Barazandeh, D. Akboga, S. S. S. Tabrizi, U. O. S. Seker, and A. E. Cicek, “RNAGEN: A generative adversarial network-based model to generate synthetic RNA sequences to target proteins,” *bioRxiv*, Jul 2023. Preprint.

[13] S. S. S. Tabrizi, S. Barazandeh, H. H. Aghdam, and A. E. Çiçek, “RNAtranslator: Modeling protein-conditional RNA design as sequence-to-sequence natural language translation,” *bioRxiv*, 2025.

[14] P. Eastman, J. Shi, B. Ramsundar, and V. S. Pande, “Solving the RNA design problem with reinforcement learning,” *PLoS Computational Biology*, vol. 14, p. e1006176, Jun 2018.

[15] F. Runge, D. Stoll, S. Falkner, and F. Hutter, “Learning to design rna,” 2019.

[16] I. Kauvar, E. B. Richman, and W. E. Allen, “Eterna-rl : Using reinforcement learning to design rna secondary structures,” 2017.

[17] A. Whatley, Z. Luo, and X. Tang, “Improving RNA Secondary Structure Design using Deep Reinforcement Learning,” Nov 2021.

[18] Y. Li, X. Pan, H. Shen, and Y. Yang, “Drag: design rnas as hierarchical graphs with reinforcement learning,” *Briefings in bioinformatics*, vol. 26, 03 2025.

[19] M. Sternke and J. Karpiak, “ProteinRL: Reinforcement learning with generative protein language models for property-directed sequence design,” in *NeurIPS 2023 Generative AI and Biology (GenBio) Workshop*, 2023.

[20] F. Stocco, M. Artigues-Lleixa, A. Hunklinger, T. Widatalla, M. Guell, and N. Ferruz, “Guiding generative protein language models with reinforcement learning,” 12 2024.

[21] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[22] A. Grønning, T. K. Doktor, S. Larsen, U. Petersen, L. L. Holm, G. Bruun, M. B. Hansen, A.-M. Hartung, J. Baumbach, and B. S. Andresen, “Deepclip: predicting the effect of mutations on protein–rna binding with deep learning,” *Nucleic Acids Research*, vol. 48, pp. 7099–7118, 06 2020.

[23] D. Risso, K. Schwartz, G. Sherlock, and S. Dudoit, “Gc-content normalization for rna-seq data,” *BMC Bioinformatics*, vol. 12, no. 1, p. 480, 2011.

[24] R. Lorenz, S. H. Bernhart, C. Höner Zu Siederdissen, H. Tafer, C. Flamm, P. F. Stadler, and I. L. Hofacker, “ViennaRNA Package 2.0,” *Algorithms for Molecular Biology*, vol. 6, no. 1, p. 26, 2011.

[25] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631, 2019.

A Appendix A: Member Contributions

Each group member's contributions to the project are detailed below:

- **Yusuf Kesmen**: Led the implementation and adaptation of the GRPO algorithm for the T5 model. Conducted hyperparameter tuning using Optuna and analyzed RL training dynamics. Primary contributor to the technical sections of the manuscript (Methods, Results) and overall technical lead.
- **Alpsencer Özdemir**: Developed and trained the protein-RNA binding classifier, including data preparation (positive/negative samples, embedding generation). Designed and implemented the individual components of the reward function (GC content, MFE). Developed and evaluated the baseline model. Contributed to the Introduction and Related Work sections of the manuscript.
- **Yavuz Alp Sencer Öztürk**: Managed the initial fine-tuning of the base T5 model and the creation/management of custom BPE tokenizers. Integrated various model components and set up the computational environment and data pipelines. Coordinated the report and presentation preparation, primary contributor to the Discussion, Conclusion, and Appendix sections of the manuscript.

All members contributed to the conceptualization of the project, literature review, debugging, and final report preparation.

B Appendix B: Code Overview

*

File: inference.py

Listing 1: Code Snippet

```
import os
import torch
import RNA
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from transformers import T5ForConditionalGeneration
from custom_tokenizer import get_tokenizer

#            Paths to your three checkpoints and their short names
CKPT_DIRS = [
    # "/data6/alpsencer/reinforce_rna/grpo-project/results/epoch_01_step_000050",
    # "/data6/alpsencer/reinforce_rna/grpo-project/results/epoch_03_step_000150",
    # "/data6/sobhan/rllm/results/train/t5/run3_20240822-152114/checkpoints/checkpoint-3
    "/data6/alpsencer/reinforce_rna/grpo-project/results/epoch_03_step_000900"
]
MODEL_NAMES = ["step900"]

#            Tokenizer files
PROTEIN_TOKENIZER_PATH =
    "/data6/alpsencer/tokenizers/bpe_protein_1000_1024.json"
RNA_TOKENIZER_PATH     =
    "/data6/alpsencer/tokenizers/bpe_rna_1000_1024.json"

#            Validation data
VALIDATION_PATH = "/data6/alpsencer/reinforce_rna/validation_rl.txt"

```

```python
#            Generation parameters
VOCAB_SIZE           = 1000
SEQ_SIZE             = 1024
MAX_PROMPT_LENGTH    = 1024
MAX_COMPLETION_LENGTH= 1024
NUM_RETURN_SEQUENCES = 1
TOP_P                = 1.0
TEMPERATURE          = 0.7
BEAM_SIZE            = 1

#            Helper functions
def map_to_bjuz(rna_acgu_sequence):
    return (rna_acgu_sequence.replace("A","b")
                             .replace("C","j")
                             .replace("U","u")
                             .replace("G","z"))

def map_to_acgu(rna_bjuz_sequence):
    s = rna_bjuz_sequence.replace(" ", "")
    return (s.replace("b","A")
             .replace("j","C")
             .replace("u","U")
             .replace("z","G"))

def load_validation_data(filepath):
    proteins, rnas = [], []
    with open(filepath) as f:
        for line in f:
            line = line.strip()
            if not line: continue
            prot, rna = line.split(",",1)
            proteins.append(prot)
            rnas.append(rna)
    return proteins, rnas

def calculate_gc_content(seq):
    seq = seq.upper()
    gc = seq.count("G") + seq.count("C")
    return (gc / len(seq)) * 100 if seq else 0

def calculate_mfe(seq):
    try:
        _, mfe = RNA.fold(seq)
        return mfe
    except:
        return None

def calculate_mfe_per_token(seq):
    """Calculate MFE normalized by sequence length"""
    mfe = calculate_mfe(seq)
    if mfe is not None and seq:
        return mfe / len(seq)
    return None

#            Main evaluation loop
def main():
    os.makedirs("analysis_outputs_2", exist_ok=True)
    proteins, ground_truths = load_validation_data(VALIDATION_PATH)

    results = {}
    for ckpt, name in zip(CKPT_DIRS, MODEL_NAMES):
        print(f"\nLoading model '{name}' from {ckpt}")
        device = "cuda" if torch.cuda.is_available() else "cpu"
        model =
            T5ForConditionalGeneration.from_pretrained(ckpt).to(device)
```

```python
        model.eval()

        # tokenizers
        tok_p = get_tokenizer("bpe", VOCAB_SIZE, SEQ_SIZE,
                              tokenizer_path=PROTEIN_TOKENIZER_PATH)
        tok_r = get_tokenizer("bpe", VOCAB_SIZE, SEQ_SIZE,
                              tokenizer_path=RNA_TOKENIZER_PATH)
        tok_p.pad_token_id = tok_r.pad_token_id = 0

        gc_vals, mfe_vals, loss_vals = [], [], []
        mfe_per_token_vals = []  # Add list for MFE per token values

        for prot, gt_acgu in tqdm(zip(proteins[:10], ground_truths[:10]),
                                  total=min(10, len(proteins)),
                                  desc=name):
            # --- tokenize protein input ---
            enc = tok_p.tokenize(prot)
            ids  = enc.ids[:MAX_PROMPT_LENGTH]
            mask = enc.attention_mask[:MAX_PROMPT_LENGTH]
            pad  = MAX_PROMPT_LENGTH - len(ids)
            ids  += [tok_p.pad_token_id]*pad
            mask += [0]*pad
            inp = {
                "input_ids":      torch.tensor([ids],dtype=torch.long).to(device),
                "attention_mask": torch.tensor([mask],dtype=torch.long).to(device)
            }

            # --- generate RNA ---
            gen_kwargs = {
                "max_new_tokens": MAX_COMPLETION_LENGTH,
                "num_return_sequences": NUM_RETURN_SEQUENCES,
                "do_sample": BEAM_SIZE==1,
                "top_p": TOP_P,
                "temperature": TEMPERATURE,
                "num_beams": BEAM_SIZE,
                "early_stopping": BEAM_SIZE>1
            }
            with torch.no_grad():
                out_ids = model.generate(**inp, **gen_kwargs)
            bjuz = tok_r.decode(out_ids)[0] if isinstance(tok_r.decode(out_ids), list) else tok_r.decode(out_ids)
            gen_acgu = map_to_acgu(bjuz)

            # --- GC & MFE ---
            gc = calculate_gc_content(gen_acgu)
            mfe = calculate_mfe(gen_acgu)
            mfe_per_token = calculate_mfe_per_token(gen_acgu)  # Calculate MFE per token
            gc_vals.append(gc)
            mfe_vals.append(mfe if mfe is not None else np.nan)
            mfe_per_token_vals.append(mfe_per_token if mfe_per_token is not None else np.nan)  # Add MFE per token

            # --- loss on ground truth ---
            gt_bjuz = map_to_bjuz(gt_acgu)
            lbl = tok_r.tokenize(gt_bjuz).ids[:MAX_COMPLETION_LENGTH]
            pad2 = MAX_COMPLETION_LENGTH - len(lbl)
            lbl += [tok_r.pad_token_id]*pad2
            labels = torch.tensor([lbl],dtype=torch.long).to(device)
            labels[labels==tok_r.pad_token_id] = -100
            with torch.no_grad():
```

```
                loss = model(**inp, labels=labels).loss.item()
            loss_vals.append(loss)

        # store
        results[name] = {
            "gc": gc_vals,
            "mfe": mfe_vals,
            "mfe_per_token": mfe_per_token_vals,  # Add MFE per token to results
            "loss": loss_vals
        }

        #     save raw data lists to txt
        np.savetxt(f"analysis_outputs_2/gc_values_{name}.txt", gc_vals,  fmt="%.4f")
        np.savetxt(f"analysis_outputs_2/mfe_values_{name}.txt", mfe_vals, fmt="%.4f")
        np.savetxt(f"analysis_outputs_2/mfe_per_token_values_{name}.txt", mfe_per_token_vals, fmt="%.6f")  # Save MFE per token values
        np.savetxt(f"analysis_outputs_2/loss_values_{name}.txt",loss_vals,fmt="%.6f")

    #           summary table
    with open("analysis_outputs_2/summary_metrics.txt","w") as f:
        f.write("Model\tGC_mean\tGC_std\tMFE_mean\tMFE_std\tMFE_per_token_mean\tMFE_per_to
        for name, m in results.items():
            g, mf, mfpt, lo = np.array(m["gc"]), np.array(m["mfe"]), np.array(m["mfe_per_token"]), np.array(m["loss"])
            f.write(f"{name}\t"
                    f"{g.mean():.2f}\t{g.std():.2f}\t"
                    f"{mf.mean():.2f}\t{mf.std():.2f}\t"
                    f"{mfpt.mean():.6f}\t{mfpt.std():.6f}\t"
                    f"{lo.mean():.4f}\t{lo.std():.4f}\n")

    #           plots
    # 1) Violin plot of GC content
    plt.figure(figsize=(8,5))
    parts = plt.violinplot(
        [results[n]["gc"] for n in MODEL_NAMES],
        showmeans=True, showmedians=True)
    plt.xticks([1,2,3], MODEL_NAMES)
    plt.ylabel("GC content (%)")
    plt.title("GC content by model")
    plt.grid(alpha=0.3)
    plt.savefig("analysis_outputs_2/gc_violin.png")
    plt.close()

    # 2) MFE distribution
    plt.figure(figsize=(8,5))
    for name in MODEL_NAMES:
        plt.hist(results[name]["mfe"], bins=30, alpha=0.5,
                 density=True, label=name)
    plt.xlabel("MFE (kcal/mol)")
    plt.ylabel("Density")
    plt.legend()
    plt.title("MFE distributions")
    plt.grid(alpha=0.3)
    plt.savefig("analysis_outputs_2/mfe_dist.png")
    plt.close()

    # 3) Loss distribution
    plt.figure(figsize=(8,5))
    for name in MODEL_NAMES:
        plt.hist(results[name]["loss"], bins=30, alpha=0.5,
                 density=True, label=name)
```

```
    plt.xlabel("Loss (NLL)")
    plt.ylabel("Density")
    plt.legend()
    plt.title(" G r o u n d truth  loss distributions")
    plt.grid(alpha=0.3)
    plt.savefig("analysis_outputs_2/loss_dist.png")
    plt.close()

    # 4) MFE per token distribution
    plt.figure(figsize=(8,5))
    for name in MODEL_NAMES:
        plt.hist(results[name]["mfe_per_token"], bins=30, alpha=0.5,
                 density=True, label=name)
    plt.xlabel("MFE per token (kcal/mol per nucleotide)")
    plt.ylabel("Density")
    plt.legend()
    plt.title("MFE per token distributions")
    plt.grid(alpha=0.3)
    plt.savefig("analysis_outputs_2/mfe_per_token_dist.png")
    plt.close()

    print("All metrics, plots and data have been saved under
        ./analysis_outputs_2/")

if __name__ == "__main__":
    main()
```

*

Directory: classifier/

File: classifier/binding_embedding_creator.py

Listing 2: Code Snippet

```
"""
Embedding generator for protein and RNAs
Last modified: 13.04.2025
Modified by: Alpsencer & Yusuf
TODO: organize the code if necessary
"""

import torch
from transformers import T5ForConditionalGeneration
from pathlib import Path
from tqdm import tqdm
from util.tokenizer import get_tokenizer  # Update the path
import numpy as np  # Add numpy for random split

from datasets import load_dataset
from copy import deepcopy
import pickle # Import pickle for incremental saving

MODEL_PATH =
    "/data6/sobhan/rllm/results/train/t5/run3_20240822-152114/checkpoints/checkpoint-34980
SOURCE_TOKENIZER =
    "/data6/alpsencer/tokenizers/bpe_protein_1000_1024.json"
RNA_TOKENIZER = "/data6/alpsencer/tokenizers/bpe_rna_1000_1024.json"
MAX_LEN = 32
# TRAIN_DATA = "/data6/sobhan/RLLM/dataset/rph/train_rp.txt"
TRAIN_DATA = "/data6/alpsencer/reinforce_rna/sample_100K.txt"
EVAL_DATA = "/data6/sobhan/RLLM/dataset/rph/eval_rp.txt"
PAIR_SAVE_DIR = "/data6/alpsencer/data/rllm_classifier_pairs"
```

```python
DATASET_NAME = "pairs_100K_eos"
BATCH_SIZE = 16
TRAIN_OUTPUT_FILENAME = f"{DATASET_NAME}_train.pkl" # Define output filename with .pkl extension
TEST_OUTPUT_FILENAME = f"{DATASET_NAME}_test.pkl" # Define output filename with .pkl extension
TRAIN_TEST_SPLIT = 0.95

def tokenize_dataset(sample, source_tokenizer, rna_tokenizer):
    text = sample["text"]
    # Basic error handling for malformed lines
    try:
        source, rna, pair = text.strip().split("$")
    except ValueError:
        # Return None or handle appropriately if lines might be malformed
        # For now, assume data is clean as per original code's lack of handling
        # If errors occur, add more robust handling here.
        # Example: return None and filter later
         print(f"Warning: Skipping malformed line: {text.strip()}") # Keep minimal warning if needed, or remove
         return None


    rna = rna.replace("A", "b").replace("C", "j").replace("U", "u").replace("G", "z")
    source = source.lower()

    source_tokenized = source_tokenizer.tokenize(source)
    rna_tokenized = rna_tokenizer.tokenize(rna)

    # replace the first padding token with eos token


    # need to set these to -100 to calculate the loss properly
    # rna_labels = [-100 if i == 0 else i for i in rna_tokenized.ids]

    # return {
    #     "input_ids": source_tokenized.ids,
    #     "attention_mask": source_tokenized.attention_mask,
    #     "labels": rna_labels,
    # }
    return {
        "source_tokenized": source_tokenized,
        "rna_tokenized": rna_tokenized,
        "label": 1 if pair == "+" else 0,
    }


def get_datasets(args, source_tokenizer, rna_tokenizer, iterable=True):

    # Get the dataset
    dataset = load_dataset(
        "text",
        data_files=args.train_data,
        split="train",
        cache_dir="/data6/alpsencer/cache",
    )

    if iterable:
        train_dataset = dataset.to_iterable_dataset()
    else:
        train_dataset = dataset
```

```python

    # # Filter dataset
    filter_source_tokenizer = deepcopy(source_tokenizer)
    filter_source_tokenizer.tokenizer.no_truncation()
    filter_rna_tokenizer = deepcopy(rna_tokenizer)
    filter_rna_tokenizer.tokenizer.no_truncation()

    def _is_valid_sample(sample):
        # Check for malformed lines first if tokenize_dataset can return None
        if sample is None or "text" not in sample:
            return False
        try:
            source, rna, _ = sample["text"].strip().split("$")
            return (
                len(filter_source_tokenizer.tokenize(source).ids) <= 1024
                and len(filter_rna_tokenizer.tokenize(rna).ids) <= 1024
            )
        except ValueError:
            return False # Filter out malformed lines

    filtered = train_dataset.filter(_is_valid_sample)

    # # # Shuffle dataset
    if iterable:
        shuffled = filtered.shuffle(buffer_size=10000)
    else:
        shuffled = filtered.shuffle()

    # Tokenize dataset
    tokenized = shuffled.map(
        lambda sample: tokenize_dataset(sample, source_tokenizer, rna_tokenizer)
    )
    # Add filtering step if tokenize_dataset returns None for errors
    tokenized = tokenized.filter(lambda x: x is not None)


    return tokenized


def count_file_lines(file_path):
    try:
        with open(file_path, "r", encoding="utf-8", errors="ignore") as file:
            return sum(1 for _ in file)
    except FileNotFoundError:
        return 0 # Return 0 if file not found, avoids crash


# Import The encoder-decoder model
device = "cuda" if torch.cuda.is_available() else "cpu"
model = T5ForConditionalGeneration.from_pretrained(MODEL_PATH)  # Update the model path
model.to(device)
model.eval() # Set model to evaluation mode

# Import the tokenizers
source_tokenizer = get_tokenizer(
    tokenizer_name="bpe",
    vocab_size=1000,
    seq_size=1024,
    tokenizer_path=SOURCE_TOKENIZER,
```

```python
)
rna_tokenizer = get_tokenizer(
    tokenizer_name="bpe", vocab_size=1000, seq_size=1024, tokenizer_path=RNA_TOKENIZER
)


# Import the dataset, tokenize the data
class Args:
    def __init__(self, train_data, eval_data):
        self.train_data = train_data
        self.eval_data = eval_data


dataset_args = Args(TRAIN_DATA, EVAL_DATA)
train_dataset = get_datasets(
    dataset_args, source_tokenizer=source_tokenizer, rna_tokenizer=rna_tokenizer
)

"""
return {
    "input_ids": source_tokenized.ids,
    "attention_mask": source_tokenized.attention_mask,
    "labels": rna_labels,
}
"""

print(model.config) # Original print statement

save_dir = Path(PAIR_SAVE_DIR)
# --- Ensure save directory exists ---
save_dir.mkdir(parents=True, exist_ok=True)
train_output_path = save_dir / TRAIN_OUTPUT_FILENAME
test_output_path = save_dir / TEST_OUTPUT_FILENAME

# Generate train/test indices before processing
total_samples = count_file_lines(TRAIN_DATA)
indices = np.arange(total_samples)
np.random.seed(42)  # Set seed for reproducibility
np.random.shuffle(indices)
train_size = int(TRAIN_TEST_SPLIT * total_samples)
train_indices = set(indices[:train_size])

# pair_list = [] # Removed: No longer accumulating in memory
print(type(train_dataset)) # Original print statement
batched_dataset = train_dataset.iter(batch_size=BATCH_SIZE)

file_length = count_file_lines(TRAIN_DATA)
# Use file_length for tqdm if reliable, otherwise might need adjustment or removal
tqdm_total = (file_length // BATCH_SIZE + 1) if file_length > 0 else None

current_index = 0  # Keep track of global index

# --- Open output file before the loop ---
with open(train_output_path, "wb") as train_outfile:
    with open(test_output_path, "wb") as test_outfile:
        # Freeze the model gradients and disable dropout etc.
        with torch.inference_mode():
            for tokenized_batch in tqdm(batched_dataset, total=tqdm_total): # Use calculated total if available

                protein_ids = [
```

```
    tokenized.ids for tokenized in tokenized_batch["source_tokenized"]
]
protein_attention = [
    tokenized.attention_mask
    for tokenized in tokenized_batch["source_tokenized"]
]

rna_ids = [tokenized.ids for tokenized in tokenized_batch["rna_tokenized"]]
rna_attention = [
    tokenized.attention_mask for tokenized in tokenized_batch["rna_tokenized"]
]

# --- Convert lists to tensors ---
protein_ids_tensor = torch.tensor(protein_ids, dtype=torch.long).to(device)
protein_attention_tensor = torch.tensor(protein_attention, dtype=torch.long).to(device)
rna_ids_tensor = torch.tensor(rna_ids, dtype=torch.long).to(device)
rna_attention_tensor = torch.tensor(rna_attention, dtype=torch.long).to(device)

# Extract protein & rna embeddings

# Forward pass through the encoder
encoder_outputs = model.encoder(
    input_ids=protein_ids_tensor,
    attention_mask=protein_attention_tensor,
)
protein_hidden_states = encoder_outputs.last_hidden_state  # Encoder embeddings

# Forward pass through the decoder
decoder_outputs = model.decoder(
    input_ids=rna_ids_tensor,
    attention_mask=rna_attention_tensor,
    encoder_hidden_states=protein_hidden_states,
    encoder_attention_mask=protein_attention_tensor,  # Pass encoder mask
    output_hidden_states=True,
)
rna_hidden_states = decoder_outputs.last_hidden_state

# Protein embedding (Mean Pooling)
# Use attention mask directly for masking and length calculation
protein_masks = protein_attention_tensor.to(device) # Already on device
protein_hidden_masked = protein_hidden_states * protein_masks.unsqueeze(-1)
protein_masked_sum = torch.sum(protein_hidden_masked, dim=1)
protein_sequence_lengths = torch.sum(protein_masks, dim=1, keepdim=True).clamp(min=1) # Avoid div by zero
protein_embeddings = protein_masked_sum / protein_sequence_lengths


```

```
                # RNA embedding (Last non-padding token / EOS-like)
                # Find the index of the last non-padding token using the attention mask
                rna_sequence_lengths_indices = rna_attention_tensor.sum(dim=1) - 1 # Get last index (0-based)
                rna_sequence_lengths_indices = rna_sequence_lengths_indices.clamp(min=0) # Ensure index is not negative

                # Gather the hidden state at the last token index for each sequence in the batch
                rna_embeddings = rna_hidden_states[
                    torch.arange(rna_hidden_states.size(0), device=device), rna_sequence_lengths_indices
                ]

                # Average embedding
                #rna_masks = torch.stack(
                #    [
                #        (torch.tensor(tokenized.ids) != 0)
                #        for tokenized in tokenized_batch["rna_tokenized"]
                #    ]
                #).to(device)
                #rna_hidden_masked = rna_hidden_states * rna_masks.unsqueeze(dim=2)
                #print(protein_hidden_masked[0])
                #rna_masked_sum = torch.sum(rna_hidden_masked, dim=1).to(device)
                #rna_sequence_lengths = torch.sum(rna_masks, dim=1).to(device)
                #rna_embeddings = rna_masked_sum / rna_sequence_lengths.unsqueeze(dim=1)


                # --- Create list of pairs *for this batch* and write incrementally ---
                current_batch_labels = tokenized_batch["label"]
                # Move tensors to CPU before pickling
                protein_embeddings_cpu = protein_embeddings.cpu()
                rna_embeddings_cpu = rna_embeddings.cpu()

                # Process each sample in the batch
                for i in range(len(current_batch_labels)):
                    pair_data = {
                        "protein": protein_embeddings_cpu[i],
                        "rna": rna_embeddings_cpu[i],
                        "label": current_batch_labels[i],
                    }
                    # Write to appropriate file based on pre-computed split
                    if current_index in train_indices:
                        pickle.dump(pair_data, train_outfile)
                    else:
                        pickle.dump(pair_data, test_outfile)
                    current_index += 1
```

File: classifier/classifier.py

Listing 3: Code Snippet

```
"""
Binding classifiers for protein and RNA embeddings
```

```python
Last modified: 24.02.2025
Modified by: Alpsencer & Yusuf
"""

import torch
import torch.nn as nn
import torch.nn.init as init


# Define the MLP Classifier
# TODO: hyperparameters from yaml
# TODO: regressor or classifier
class MLPBindingClassifier(nn.Module):
    def __init__(self, protein_dim, rna_dim, hidden_dims=[512, 256,
        128], dropout_rate=0.2):
        super(MLPBindingClassifier, self).__init__()

        # self.protein_mean = protein_mean
        # self.protein_std = protein_std

        # self.rna_mean = rna_mean
        # self.rna_std = rna_std

        # Input dimension is the concatenated protein and RNA
            embeddings
        input_dim = protein_dim + rna_dim

        # Build the MLP layers
        layers = []
        # layers.append(nn.BatchNorm1d(num_features=input_dim))
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(p=dropout_rate))
            prev_dim = hidden_dim

        # Output layer (binary classification)
        layers.append(nn.Linear(prev_dim, 1))
        layers.append(nn.Sigmoid())

        self.model = nn.Sequential(*layers)

    def forward(self, protein_embedding, rna_embedding):
        # Concatenate the embeddings

        # protein_normalized = ( protein_embedding -
            self.protein_mean ) / self.protein_std

        # rna_normalized = ( rna_embedding - self.rna_mean) /
            self.rna_std

        combined = torch.cat((protein_embedding, rna_embedding),
            dim=1)

        return self.model(combined)


class ContrastiveBindingClassifier(nn.Module):
    pass
```

*

File: classifier/embedding_loader.py

Listing 4: Code Snippet

```
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F
import pickle


class ProteinRNAEmbeddingDataset(Dataset):
    def __init__(self, dataset_path):

        # self.data = torch.load(dataset_path)

        # Load the dataset with pickle
        self.data = []
        with open(dataset_path, "rb") as f:
            while True:
                try:
                    self.data.append(pickle.load(f))
                except EOFError:
                    break

        proteins = torch.stack([x["protein"] for x in self.data])
        rnas = torch.stack([x["rna"] for x in self.data])
        labels = torch.tensor([x["label"] for x in self.data])

        self.proteins_normalized = F.normalize(proteins, p=2, dim=-1)
        self.rnas_normalized = F.normalize(rnas, p=2, dim=-1)
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        item = self.data[idx]
        # protein = torch.tensor(item["protein"], dtype=torch.float32)
        # rna = torch.tensor(item["rna"], dtype=torch.float32)
        protein =
            item["protein"].clone().detach().requires_grad_(True).float()
        rna =
            item["rna"].clone().detach().requires_grad_(True).float()
        label = torch.tensor(item["label"], dtype=torch.long)

        return protein, rna, label
```

*

File: classifier/train_classifier.py

Listing 5: Code Snippet

```
"""
Train a classifier for binding score of protein and rna embeddings
Last modified: 13.04.2025
Modified by: Alpsencer & Yusuf
"""

import torch
from torch.utils.data import Dataset, DataLoader
from torch.optim import Adam
from torch.nn import BCELoss
```

```python
from torch.optim.lr_scheduler import LinearLR
from tqdm import tqdm
import wandb

from classifier import MLPBindingClassifier
from embedding_loader import ProteinRNAEmbeddingDataset

# Constants
TRAIN_DATA = "/data6/alpsencer/data/rllm_classifier_pairs/pairs_100K_eos_train.pkl"
VALIDATION_DATA = "/data6/alpsencer/data/rllm_classifier_pairs/pairs_100K_eos_test.pkl"
BATCH_SIZE = 64
LEARNING_RATE = 0.00001
WEIGHT_DECAY = 0.01  # L2 regularization parameter
DROPOUT_RATE = 0.0  # Dropout rate for regularization
NUM_EPOCHS = 1000
VALIDATION_EPOCHS = 5
WARMUP_EPOCHS = 10  # Number of epochs for warmup
WARMUP_START_FACTOR = 0.1  # Starting learning rate factor

# Initialize wandb
wandb.init(
    project="rllm-binding-classifier",
    config={
        "learning_rate": LEARNING_RATE,
        "weight_decay": WEIGHT_DECAY,
        "dropout_rate": DROPOUT_RATE,
        "batch_size": BATCH_SIZE,
        "epochs": NUM_EPOCHS,
        "validation_epochs": VALIDATION_EPOCHS,
        "warmup_epochs": WARMUP_EPOCHS,
        "warmup_start_factor": WARMUP_START_FACTOR,
    }
)

# Load dataset and create DataLoader
train_dataset = ProteinRNAEmbeddingDataset(TRAIN_DATA)
validation_dataset = ProteinRNAEmbeddingDataset(VALIDATION_DATA)

train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
validation_dataloader = DataLoader(validation_dataset, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)

# Find embedding sizes
proteins, rnas, labels = next(iter(train_dataloader))
protein_dim = proteins.shape[1]
rna_dim = rnas.shape[1]

# Initialize the model, optimizer, and loss function
device = "cuda" if torch.cuda.is_available() else "cpu"

model = MLPBindingClassifier(
    protein_dim=protein_dim, rna_dim=rna_dim, hidden_dims=[256, 64], dropout_rate=DROPOUT_RATE
)

model.to(device)

optimizer = Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=WEIGHT_DECAY)
criterion = BCELoss()  # Binary Cross-Entropy Loss for binary classification

```

```python
# Create learning rate scheduler with linear warmup
total_steps = len(train_dataloader) * NUM_EPOCHS
warmup_steps = len(train_dataloader) * WARMUP_EPOCHS
scheduler = LinearLR(
    optimizer,
    start_factor=WARMUP_START_FACTOR,
    end_factor=1.0,
    total_iters=warmup_steps
)

# normalized_embedding = (embedding - stats["mean"]) / stats["std"]

# Training loop
for epoch in range(NUM_EPOCHS):
    print(f"Epoch: {epoch+1}")

    model.train()  # Set the model to training mode
    running_loss = 0.0

    for batch in tqdm(train_dataloader):
        proteins, rnas, labels = batch

        labels.unsqueeze_(1)  # unsqueeze to make labels 2D tensor
        proteins, rnas, labels = proteins.to(device),
            rnas.to(device), labels.to(device)

        # Forward pass
        outputs = model(proteins, rnas)
        loss = criterion(outputs, labels.float())

        # Zero the gradients
        optimizer.zero_grad()

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        # Step the scheduler
        if epoch < WARMUP_EPOCHS:
            scheduler.step()

        running_loss += loss.item()

    # Log training loss and learning rate per epoch
    epoch_loss = running_loss / len(train_dataloader)
    current_lr = optimizer.param_groups[0]['lr']
    wandb.log({
        "train_loss": epoch_loss,
        "learning_rate": current_lr,
        "epoch": epoch + 1
    })
    print(f"Epoch [{epoch+1}/{NUM_EPOCHS}], Loss: {epoch_loss:.4f},
       LR: {current_lr:.6f}")

    if (epoch + 1) % VALIDATION_EPOCHS == 0:
        model.eval()
        validation_loss = 0.0
        total_predictions = 0
        true_positives = 0
        predicted_positives = 0
        total_positives = 0

        with torch.inference_mode():
            for batch in tqdm(validation_dataloader):
                proteins, rnas, labels = batch
```

```
                labels.unsqueeze_(1)
                proteins, rnas, labels = proteins.to(device), rnas.to(device), labels.to(device)

                outputs = model(proteins, rnas)
                loss = criterion(outputs, labels.float())
                validation_loss += loss.item()

                # Convert outputs to binary predictions
                predictions = (outputs > 0.5).float()

                # Update metrics
                total_predictions += len(predictions)
                true_positives += ((predictions == 1) & (labels == 1)).sum().item()
                predicted_positives += (predictions == 1).sum().item()
                total_positives += (labels == 1).sum().item()

            # Calculate final metrics
            accuracy = true_positives / total_predictions if total_predictions > 0 else 0
            precision = true_positives / (predicted_positives + 1e-8) if predicted_positives > 0 else 0
            recall = true_positives / (total_positives + 1e-8) if total_positives > 0 else 0
            f1_score = 2 * (precision * recall) / (precision + recall + 1e-8) if (precision + recall) > 0 else 0

            # Log validation metrics
            val_loss = validation_loss / len(validation_dataloader) if len(validation_dataloader) > 0 else 0
            wandb.log({
                "val_loss": val_loss,
                "val_accuracy": accuracy,
                "val_precision": precision,
                "val_recall": recall,
                "val_f1": f1_score,
                "epoch": epoch + 1
            })

            print(f"Validation Loss: {val_loss:.4f}")
            print(f"Accuracy: {accuracy:.4f}, Precision: {precision:.4f}, Recall: {recall:.4f}, F1 Score: {f1_score:.4f}")

print("Training complete!")
wandb.finish()
```

*

Directory: classifier/util/

*

File: classifier/util/generate_background_seq.py

Listing 6: Code Snippet

```
import csv
import random
from tqdm import tqdm

# ------------------------------------------------------------------------------
```

```
# 1) Define file paths
# -----------------------------------------------------------------------------
INPUT_CSV = "/data6/alpsencer/reinforce_rna/output.csv"
OUTPUT_CSV = "/data6/alpsencer/reinforce_rna/background_sequences.csv"

# -----------------------------------------------------------------------------
# 2) Function to generate a background sequence by shuffling the RNA sequence
# -----------------------------------------------------------------------------
def generate_background(rna_seq):
    rna_list = list(rna_seq)
    random.shuffle(rna_list)
    return "".join(rna_list)

# -----------------------------------------------------------------------------
# 3) Iteratively process the CSV file row by row
# -----------------------------------------------------------------------------
with open(INPUT_CSV, "r", newline="") as infile, open(OUTPUT_CSV, "w", newline="") as outfile:
    reader = csv.DictReader(infile)
    writer = csv.DictWriter(outfile, fieldnames=reader.fieldnames)
    writer.writeheader()

    # Process each row iteratively with a progress bar
    for row in tqdm(reader, desc="Processing rows"):
        original_rna = row["rna_sequence"]
        bg_rna = generate_background(original_rna)
        row["rna_sequence"] = bg_rna
        row["pair"] = "-"  # mark as background sequence
        writer.writerow(row)

print("Background sequences saved to:", OUTPUT_CSV)
```

*

File: classifier/util/generate_protein_input.py

Listing 7: Code Snippet

```
# This will converr the protein_seqs csv file in the form that grpo trainer can get properly (different protein inputs)
# and save it as protein_input.txt
import pandas as pd

# 1) Read your source CSV
df = pd.read_csv("protein_seqs.csv")

# 2) Build each line as "sequence$prot_name"
lines = df.apply(lambda row: f"{row['seq']}${row['prot_name']}", axis=1)

# 3) Write to a .txt file
with open("protein_input.txt", "w") as f:
    for line in lines:
        f.write(line + "\n")

print("Wrote", len(lines), "entries to protein_input.txt")
```

*

File: classifier/util/sample_generate.py

Listing 8: Code Snippet

```python
import csv
import random
from tqdm import tqdm

# -----------------------------------------------------------------------------
# 1) Define file paths and sample sizes
# -----------------------------------------------------------------------------
POSITIVE_CSV   = "/data6/alpsencer/reinforce_rna/positive_sequences.csv"
BACKGROUND_CSV = "/data6/alpsencer/reinforce_rna/background_sequences.csv"
MERGED_TXT     = "/data6/alpsencer/reinforce_rna/sample_100K.txt"
NUM_SAMPLES_PER_FILE = 100000  # 50K rows per file

# -----------------------------------------------------------------------------
# 2) Generator function to stream rows with strand '+' from a file
# -----------------------------------------------------------------------------
def positive_rows_generator(file_path, limit):
    """
    Yields rows from file_path where the strand is '+' until the limit is reached.
    """
    count = 0
    with open(file_path, "r", newline="") as infile:
        reader = csv.DictReader(infile, delimiter=",")
        for row in reader:
            if row["strand"] == "+":
                yield row
                count += 1
                if count >= limit:
                    break

# -----------------------------------------------------------------------------
# 3) Collect rows from both CSV files without reading them entirely into memory
# -----------------------------------------------------------------------------
merged_rows = []

# Process the positive CSV
for row in tqdm(positive_rows_generator(POSITIVE_CSV, NUM_SAMPLES_PER_FILE),
                desc="Processing positive CSV"):
    merged_rows.append({
        "protein_seq": row["protein_seq"],
        "rna_seq": row["rna_sequence"],
        "pair": row["pair"]
    })

# Process the background CSV
for row in tqdm(positive_rows_generator(BACKGROUND_CSV, NUM_SAMPLES_PER_FILE),
                desc="Processing background CSV"):
    merged_rows.append({
        "protein_seq": row["protein_seq"],
```

```python
        "rna_seq": row["rna_sequence"],
        "pair": row["pair"]
    })

# ------------------------------------------------------------------------------
# 4) Shuffle the merged rows randomly
# ------------------------------------------------------------------------------
random.shuffle(merged_rows)

# ------------------------------------------------------------------------------
# 5) Write the shuffled rows to a TXT file using '$' as the delimiter
# ------------------------------------------------------------------------------
with open(MERGED_TXT, "w", newline="") as outfile:
    fieldnames = ["protein_seq", "rna_seq", "pair"]
    writer = csv.DictWriter(outfile, fieldnames=fieldnames, delimiter="$")
    writer.writeheader()
    for row in merged_rows:
        writer.writerow(row)

print(f"Merged and shuffled sample TXT created with {NUM_SAMPLES_PER_FILE * 2} rows using '$' as delimiter:", MERGED_TXT)
```

*

File: classifier/util/tokenizer.py

Listing 9: Code Snippet

```python
import json
import os
import numpy as np


from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import ByteLevel
from tokenizers.processors import TemplateProcessing
from tokenizers.normalizers import Sequence, Lowercase

import torch


class BpeTokenizer:
    def __init__(self, seq_size, vocab_size):
        self.tokenizer = Tokenizer(BPE())
        self.tokenizer.normalizer = Sequence([Lowercase()])
        self.tokenizer.pre_tokenizer = ByteLevel(add_prefix_space=False)

        self.tokenizer.enable_padding(max_length=seq_size, direction='right')
        self.tokenizer.enable_truncation(max_length=seq_size)

        self.special_tokens = {
            "pad": {"id": 0, "token": "<pad>"},
            "eos": {"id": 1, "token": "</s>"},
            "unk": {"id": 2, "token": "<unk>"},
```

```python
        }


        self.special_tokens_list = [None] * len(self.special_tokens)
        for token_dict in self.special_tokens.values():
            self.special_tokens_list[token_dict["id"]] = token_dict["token"]

        self.tokenizer.post_processor = TemplateProcessing(
            single=f"$A {self.special_tokens['eos']['token']}",
            special_tokens=[
                (self.special_tokens["eos"]["token"], self.special_tokens["eos"]["id"]),
            ],
        )

        self.trainer = BpeTrainer(
            vocab_size=vocab_size,
            special_tokens=self.special_tokens_list,
            show_progress=True
        )

    def train_tokenizer(self, train_data, which=True):
        def iterator(data, which):
            for sequence_item in data: # Renamed sequence to sequence_item
                text = sequence_item['text'] # Renamed sequence to sequence_item
                mol, rna  = text.strip().split('$')
                if which:
                    yield mol
                else:
                    yield rna
        self.tokenizer.train_from_iterator(iterator(data=train_data, which=which), trainer=self.trainer)
        self.add_unk_id()

    def train_from_files(self, data_files: str):
        self.tokenizer.train(files=[data_files], trainer=self.trainer)
        self.add_unk_id()

    def add_unk_id(self):
        tokenizer_json = json.loads(self.tokenizer.to_str())
        tokenizer_json["model"]["unk_id"] = self.special_tokens["unk"]["id"]
        self.tokenizer = Tokenizer.from_str(json.dumps(tokenizer_json))

    def save(self, path: str, name: str):
        if not os.path.exists(path):
            os.makedirs(path)
        self.tokenizer.save(os.path.join(path, f"{name}.json"))

    def load(self, path: str):
        self.tokenizer = Tokenizer.from_file(path)

    def tokenize(self, sequence_to_tokenize): # Renamed sequence to sequence_to_tokenize
        # print(sequence_to_tokenize)
        return self.tokenizer.encode(sequence_to_tokenize)

    def decode(self, sequence_to_decode): # Renamed sequence to sequence_to_decode
        # print(sequence_to_decode)
        if isinstance(sequence_to_decode, torch.Tensor):
```

```
            sequence_to_decode = sequence_to_decode.detach().cpu().numpy().astype(dtype=np.int64)
        # decode method of huggingface tokenizers usually decodes a single sequence of ids,
        # or a batch if batch_decode is used. The original code here implies it might get a list of lists.
        # However, self.tokenizer.decode expects a list of ints (single sequence) or a string.
        # If sequence_to_decode is already a list of token IDs (e.g., from model.generate), this is correct.
        # If it's a list of sequences (list of list of IDs), it would need a loop or batch_decode.
        # The custom_tokenizer.py in grpo-trainer has a different decode for list of sequences.
        # Assuming this one is for a single sequence.
        return self.tokenizer.decode(sequence_to_decode)

    def encode(self, tokenized_list): # Renamed tokenized to tokenized_list
        return [self.tokenizer.encode(seq) for seq in tokenized_list]


def get_tokenizer(tokenizer_name:str, vocab_size:int, seq_size:int, tokenizer_path:str=None):
    # Choose tokenizer
    if tokenizer_name=="bpe":
        my_tokenizer = BpeTokenizer(vocab_size=vocab_size, seq_size=seq_size)
    else:
        raise NotImplementedError

    # Load pre-trained tokenizer or train tokenizer
    if tokenizer_path:
        my_tokenizer.load(tokenizer_path)

    if vocab_size != my_tokenizer.tokenizer.get_vocab_size():
        assert "There is a conflict Tokenizer's vocab size and arguments'"

    return my_tokenizer
```

Directory: grpo-trainer/

File: grpo-trainer/config.py

Listing 10: Code Snippet

```
MODEL_PATH = "/data6/sobhan/rllm/results/train/t5/run3_20240822-152114/checkpoints/checkpoint-34980
PROTEIN_TOKENIZER = "/data6/alpsencer/tokenizers/bpe_protein_1000_1024.json"
RNA_TOKENIZER = "/data6/alpsencer/tokenizers/bpe_rna_1000_1024.json"

TRAIN_DATA = "/data6/alpsencer/reinforce_rna/protein_input.txt"
EVAL_DATA = "/data6/alpsencer/reinforce_rna/validation_rl.txt"

MAX_LENGTH = 1024
VOCAB_SIZE = 1000
BATCH_SIZE = 16
LEARNING_RATE = 1e-5
NUM_TRAIN_EPOCHS = 3
GRADIENT_ACCUMULATION_STEPS = 4
```

```
# FREEZE_ENCODER = True  # Whether to freeze the encoder during training

GC_CONTENT_MIN = 0.4  # Minimum acceptable GC content
GC_CONTENT_MAX = 0.6  # Maximum acceptable GC content
BINDING_AFFINITY_WEIGHT = 0.2  # Weight for binding affinity reward
GC_CONTENT_WEIGHT = 0.4  # Weight for GC content reward
MFE_WEIGHT = 0.4  # Weight for MFE reward

NUM_GENERATIONS = 8  # Number of generations per prompt
MAX_PROMPT_LENGTH = 1024  # Maximum length of the prompt
MAX_COMPLETION_LENGTH = 256  # Maximum length of the generated completion

BETA = 0.04  # KL coefficient
EPSILON = 0.1  # Epsilon value for clipping
MU = 5

WANDB_PROJECT = "rna-protein-grpo"
```

File: grpo-trainer/custom_tokenizer.py

Listing 11: Code Snippet

```
import json
import os
import numpy as np


from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import ByteLevel
from tokenizers.processors import TemplateProcessing
from tokenizers.normalizers import Sequence, Lowercase

import torch


class BpeTokenizer:
    def __init__(self, seq_size, vocab_size):
        self.tokenizer = Tokenizer(BPE())
        self.tokenizer.normalizer = Sequence([Lowercase()])
        self.tokenizer.pre_tokenizer = ByteLevel(add_prefix_space=False)

        self.tokenizer.enable_padding(max_length=seq_size, direction='right')
        self.tokenizer.enable_truncation(max_length=seq_size)

        self.special_tokens = {
            "pad": {"id": 0, "token": "<pad>"},
            "eos": {"id": 1, "token": "</s>"},
            "unk": {"id": 2, "token": "<unk>"},
        }


        self.special_tokens_list = [None] * len(self.special_tokens)
        for token_dict in self.special_tokens.values():
            self.special_tokens_list[token_dict["id"]] = token_dict["token"]

        self.tokenizer.post_processor = TemplateProcessing(
            single=f"$A {self.special_tokens['eos']['token']}",
```

```
            special_tokens=[
                (self.special_tokens["eos"]["token"], self.special_tokens["eos"]["id"]),
            ],
        )

        self.trainer = BpeTrainer(
            vocab_size=vocab_size,
            special_tokens=self.special_tokens_list,
            show_progress=True
        )

    def train_tokenizer(self, train_data, which=True):
        def iterator(data, which_mol): # Renamed which to which_mol
            for sequence_item in data: # Renamed sequence to sequence_item
                text = sequence_item['text'] # Renamed sequence to sequence_item
                mol, rna  = text.strip().split('$')
                if which_mol: # Use which_mol
                    yield mol
                else:
                    yield rna
        self.tokenizer.train_from_iterator(iterator(data=train_data, which_mol=which), trainer=self.trainer) # Pass which as which_mol
        self.add_unk_id()

    def train_from_files(self, data_files: str):
        self.tokenizer.train(files=[data_files], trainer=self.trainer)
        self.add_unk_id()

    def add_unk_id(self):
        tokenizer_json = json.loads(self.tokenizer.to_str())
        tokenizer_json["model"]["unk_id"] = self.special_tokens["unk"]["id"]
        self.tokenizer = Tokenizer.from_str(json.dumps(tokenizer_json))

    def save(self, path: str, name: str):
        if not os.path.exists(path):
            os.makedirs(path)
        self.tokenizer.save(os.path.join(path, f"{name}.json"))

    def load(self, path: str):
        self.tokenizer = Tokenizer.from_file(path)

    def tokenize(self, sequence_to_tokenize): # Renamed sequence to sequence_to_tokenize
        # print(sequence_to_tokenize)
        return self.tokenizer.encode(sequence_to_tokenize)

    def decode(self, sequences_to_decode): # Renamed sequences to sequences_to_decode
        # print(sequences_to_decode)
        if isinstance(sequences_to_decode, torch.Tensor):
            sequences_to_decode = sequences_to_decode.detach().cpu().numpy().astype(dtype=np.int64)
        # This decode is meant to handle a batch of sequences (list of list of IDs or tensor)
        # and return a list of decoded strings.
        return [self.tokenizer.decode(seq) for seq in sequences_to_decode]

```

```python
    def encode(self, tokenized_list): # Renamed tokenized to tokenized_list
        return [self.tokenizer.encode(seq) for seq in tokenized_list]


def get_tokenizer(tokenizer_name:str, vocab_size:int, seq_size:int, tokenizer_path:str=None):
    # Choose tokenizer
    if tokenizer_name=="bpe":
        my_tokenizer = BpeTokenizer(vocab_size=vocab_size, seq_size=seq_size)
    else:
        raise NotImplementedError

    # Load pre-trained tokenizer or train tokenizer
    if tokenizer_path:
        my_tokenizer.load(tokenizer_path)

    if vocab_size != my_tokenizer.tokenizer.get_vocab_size():
        assert "There is a conflict Tokenizer's vocab size and arguments'"

    return my_tokenizer
```

File: grpo-trainer/hyperparameter_tuning.py

Listing 12: Code Snippet

```python
import optuna
import wandb
import sys
import copy
import torch
from pathlib import Path

# Add the project root to Python path to import from sibling directories
project_root = Path(__file__).resolve().parent # Use resolve() for robustness
sys.path.append(str(project_root))

# from config import (
#     LEARNING_RATE, BETA, EPSILON, MU,
#     BINDING_AFFINITY_WEIGHT, GC_CONTENT_WEIGHT, MFE_WEIGHT,
#     NUM_GENERATIONS, TOP_P, TEMPERATURE
# )
from main import ProteinRNAGRPOTainer # Assuming main.py is in the same dir (project_root)

# def run_hyperparameter_tuning(model, tok_p, tok_r, train_envs, eval_envs, config_obj, n_trials=20): # Renamed config to config_obj
def run_hyperparameter_tuning(model, tok_p, tok_r, train_envs, eval_envs, tuning_config, n_trials=20): # Renamed config_obj to tuning_config
    # Save the initial state of the model to reset before each trial
    base_model_state = copy.deepcopy(model.state_dict())

    def objective(trial):
        try:
            # Reset the model to the base state before each trial
            model.load_state_dict(base_model_state)

```

```python
        # Reset optimizer state by creating a new optimizer instance
        # The optimizer is created within ProteinRNAGRPOTainer, so this might not be strictly necessary here
        # if the trainer re-initializes its optimizer or if model reset is enough.
        # However, explicit re-creation ensures no stale optimizer state.
        # optimizer = torch.optim.AdamW(model.parameters(), lr=tuning_config.learning_rate) # Use tuning_config

        # Define hyperparameter search space for only mu, beta, and epsilon
        beta_val = trial.suggest_float('beta', 0.01, 0.1) # Renamed beta to beta_val
        epsilon_val = trial.suggest_float('epsilon', 0.05, 0.2) # Renamed epsilon to epsilon_val
        mu_val = trial.suggest_int('mu', 3, 8) # Renamed mu to mu_val

        # Initialize wandb run with the trial parameters
        current_run = wandb.init( # Renamed run to current_run
            project="rna-protein-grpo", # Or use tuning_config.WANDB_PROJECT if defined
            name=f"trial_{trial.number}",
            config={
                'beta': beta_val,
                'epsilon': epsilon_val,
                'mu': mu_val,
            },
            reinit=True  # Allow multiple runs in the same process
        )

        # Update config parameters FOR THIS TRIAL
        # The ProteinRNAGRPOTainer __init__ takes a config object.
        # We need to pass these tuned values to it.
        # One way is to create a new config object for each trial or modify a copy.

        trial_config = copy.deepcopy(tuning_config) # Work on a copy
        # These attributes (BETA, EPSILON, MU) need to be accessible by ProteinRNAGRPOTainer
        # If ProteinRNAGRPOTainer reads them from the global 'config' module, then we'd set config.BETA
        # If ProteinRNAGRPOTainer reads them from its self.config_obj, then we set them on trial_config.
        # Assuming ProteinRNAGRPOTainer uses its self.config_obj.BETA etc.
        trial_config.BETA = beta_val
        trial_config.EPSILON = epsilon_val
        trial_config.MU = mu_val

        # Run training
        trainer = ProteinRNAGRPOTainer(model, tok_p, tok_r, train_envs, eval_envs, trial_config) # Pass trial_config
        trainer.train(epochs=1)  # Run for 1 epoch during tuning

        # Get the final reward as the objective value
        # Check if summary is available and contains the key
        final_reward = 0
        if current_run and current_run.summary:
             final_reward = current_run.summary.get('reward/mean', 0)
```

```python
            # End wandb run properly
            if wandb.run is not None: # Check if a run is active
                wandb.finish()

            return final_reward

        except Exception as e:
            print(f"Trial failed with error: {str(e)}")
            # Make sure to finish the wandb run even if there's an error
            if wandb.run is not None:
                wandb.finish()
            # Return a very low reward to indicate failure
            return float('-inf')

    study = optuna.create_study(direction='maximize')
    study.optimize(objective, n_trials=n_trials)

    print("Best trial:")
    best_trial_obj = study.best_trial # Renamed trial to best_trial_obj

    print("  Value: ", best_trial_obj.value)
    print("  Params: ")
    for key, value in best_trial_obj.params.items():
        print(f"    {key}: {value}")

    # Restore the model to the initial base state (not necessarily best params state)
    # Or, if you want to set model to best params, you'd need to re-run with best_params.
    model.load_state_dict(base_model_state)

    return study.best_params
```

File: grpo-trainer/main.py

Listing 13: Code Snippet

```python
import argparse
import os
import random
import torch
import numpy as np
from transformers import T5ForConditionalGeneration, PreTrainedTokenizerFast
# from trl import GRPOConfig # GRPOConfig from trl is for a different GRPO setup. We use a custom one.
from reward_functions import rewarder
import config as global_config_module # Use an alias for the global config module
from custom_tokenizer import get_tokenizer
# from torch.nn.functional import log_softmax # Not directly used, F.log_softmax is used

import copy # Already imported
import torch.nn.functional as F


# from config import EPSILON, MU, BETA # These will be taken from global_config_module or trainer's config object


```

```python
import wandb
import multiprocessing as mp
from functools import partial

# wandb.init(project="protein-rna-grpo", name="grpo-run") # Init should be done once, ideally in main or controlled per run

# --- Define a simple config class for our GRPO Trainer ---
# This replaces the direct use of trl.GRPOConfig for a custom setup
class CustomGRPOConfig:
    def __init__(self, **kwargs):
        self.output_dir = "./results"
        self.learning_rate = 1e-5
        self.max_prompt_length = 1024
        self.max_completion_length = 256
        self.num_generations = 8 # Number of generations per prompt
        self.top_p = 1.0
        self.temperature = 1.0
        self.max_grad_norm = 1.0

        # GRPO specific, potentially tuned
        self.BETA = global_config_module.BETA # Default from global config
        self.EPSILON = global_config_module.EPSILON # Default from global config
        self.MU = global_config_module.MU # Default from global config

        # Update with any provided kwargs
        for key, value in kwargs.items():
            setattr(self, key, value)

# --- End simple config class ---


def generate_with_log_probs(
    model_obj: T5ForConditionalGeneration, # Renamed model to model_obj
    input_ids: torch.Tensor,
    attention_mask: torch.Tensor,
    num_return_sequences: int,
    max_new_tokens: int,
    top_p: float,
    temperature: float,
) -> tuple[torch.Tensor, torch.Tensor]:
    """
    1) Sample N sequences with model_obj.generate (still under no-grad)
    2) Re-run the model_obj forward pass on those sequences to obtain
       token-wise log-probs **with gradient**.
    Returns:
        sequences      (N, T) generated token ids  (without the start token)
        log_probs     (N, T) log-probs that require grad
    """
    device = input_ids.device

    # ---- 1) sampling (no grad) ------------------------------------------
    with torch.no_grad():
        gen_out = model_obj.generate(
            input_ids=input_ids,
            attention_mask=attention_mask,
            max_new_tokens=max_new_tokens,
            do_sample=True,
            top_p=top_p,
```

```python
            temperature=temperature,
            num_return_sequences=num_return_sequences,
            return_dict_in_generate=True,
        )
    # gen_out.sequences shape: (N, 1 + T)   (extra decoder-start token)
    full_sequences = gen_out.sequences

    sequences = full_sequences[:, 1:]            # drop start token (N, T)
    N, T = sequences.shape

    # ---- 2) compute log-probs with grad -----------------------------------
    # repeat the encoder prompt N times (because batch == 1)
    enc_input_ids      = input_ids.repeat_interleave(N, dim=0)
    enc_attention_mask = attention_mask.repeat_interleave(N, dim=0)

    # decoder_input_ids are everything *before* each position
    # prepend the decoder-start token id first
    start_tok = model_obj.config.decoder_start_token_id
    start_col = torch.full((N, 1), start_tok, dtype=torch.long, device=device)
    decoder_input_ids = torch.cat([start_col, sequences[:, :-1]], dim=1)  # (N, T)

    outputs = model_obj(
        input_ids=enc_input_ids,
        attention_mask=enc_attention_mask,
        decoder_input_ids=decoder_input_ids,
    )
    logits = outputs.logits                            # (N, T, V)

    log_probs_tensor = torch.log_softmax(logits, dim=-1)   # Renamed log_probs to log_probs_tensor (N, T, V)

    # pick the prob of the *actual* generated token at each step
    tok_ids = sequences.unsqueeze(-1)                  # (N, T, 1)
    log_probs_tensor = log_probs_tensor.gather(-1, tok_ids).squeeze(-1)  # (N, T)     requires grad

    return sequences, log_probs_tensor


def process_tokens_args_tuple(args_tuple): # Renamed args to args_tuple to avoid conflict
    """
    Helper function to process tokens for a single sequence, takes a tuple for pool.map
    """
    i, t, rna_ids, rna_str_list, protein_str_val, pad_token_id, eos_token_id = args_tuple # Unpack tuple, rna -> rna_str_list, protein -> protein_str_val
    if (rna_ids[i, t] == pad_token_id or rna_ids[i, t] == eos_token_id):
        return t, 0.0

    # Build the text up to this token
    # rna_str_list[i] is a list of characters/tokens for the i-th sequence
    text = "".join(rna_str_list[i][:t+1])
    reward_val = rewarder(text, protein_str_val) # Renamed reward to reward_val
    return t, reward_val

```

```python
def calculate_reward(
    rna_ids: torch.Tensor,
    rna_decoded_str_list: list[list[str]], # rna is now list of list of chars/tokens for each seq in batch
    protein_ids: torch.Tensor, # protein_ids not directly used here but good for context
    protein_str_val: str, # protein is now a single string for the batch
    pad_token_id: int = 0,
    eos_token_id: int = 1
) -> torch.Tensor:
    """
    sequences: (N, T) token IDs for generated RNAs
    rna_decoded_str_list: (N, list_of_chars_at_t) decoded RNA strings (ACGU format)
    protein_str_val: the source protein string (same for all N if batch_size=1)
    returns:   (N, T) rewards for each prefix ending at t,
               where rewarder(text, protein_str_val) is called.
    """
    device = rna_ids.device # Keep rewards on the original device if possible, or move later
    # rna_ids_cpu = rna_ids.to("cpu") # Move to CPU if multiprocessing requires it (often does)
    # protein_ids_cpu = protein_ids.to("cpu") # Not used in process_tokens_args_tuple directly

    N, T = rna_ids.size()
    rewards_tensor = torch.zeros(N, T, device=device) # Renamed rewards to rewards_tensor

    # Create a pool of workers
    # Using try-finally to ensure pool is closed
    pool = None
    try:
        # num_workers = mp.cpu_count() # Can be too many, limit if needed
        num_workers = min(mp.cpu_count(), 4) # Example limit
        pool = mp.Pool(processes=num_workers)

        for i in range(N):
            # Prepare arguments for each token position in this sequence
            # Pass rna_ids (on CPU if needed by pool), rna_decoded_str_list (Python list), protein_str_val (Python str)
            token_args_list = [(i, t, rna_ids, rna_decoded_str_list, protein_str_val, pad_token_id, eos_token_id)
                               for t in range(T)]

            # Process tokens in parallel
            results = pool.map(process_tokens_args_tuple, token_args_list)

            # Combine results for this sequence
            for t_res, reward_val_res in results: # Renamed t, reward to t_res, reward_val_res
                rewards_tensor[i, t_res] = reward_val_res
    finally:
        if pool:
            pool.close()
            pool.join()

    return rewards_tensor

```

```python

def calculate_advantage(rewards: torch.Tensor, eps: float = 1e-8) -> torch.Tensor:
    """
    Normalize advantages per time-step (column), ignoring padded values.
    rewards: (N, T) tensor of rewards, where padded values are 0
    returns: (N, T) tensor of normalized advantages
    """
    # Create a mask for non-padded values (assuming padding is 0)
    mask = (rewards != 0).float()

    # Calculate mean only over non-padded values
    sum_rewards = (rewards * mask).sum(dim=0, keepdim=True)  # (1, T)
    count = mask.sum(dim=0, keepdim=True).clamp(min=eps)  # Avoid division by zero if count is 0
    mean = sum_rewards / count  # (1, T)

    # Center the rewards
    centered = rewards - mean  # (N, T)

    # Calculate std only over non-padded values
    squared_diff = (centered * mask) ** 2  # (N, T)
    sum_squared_diff = squared_diff.sum(dim=0, keepdim=True)  # (1, T)
    std = torch.sqrt(sum_squared_diff / count ) # (1, T) # count already has eps

    # Normalize and apply mask to ensure padded values remain 0
    advantages = (centered / (std + eps)) * mask # Add eps to std for stability
    return advantages


class ProteinRNAEnvironment:
    """
    Environment wrapper for ProteinRNA, using tokenizer.tokenize()
    (returns .ids and .attention_mask).
    """
    def __init__(self, protein: str, rna: str,
                 protein_tokenizer, rna_tokenizer,
                 max_input_length: int, max_output_length: int):
        self.protein_str = protein # Renamed protein to protein_str
        self.rna_str     = rna # Renamed rna to rna_str
        self.p_tok   = protein_tokenizer
        self.r_tok   = rna_tokenizer
        self.max_in  = max_input_length
        self.max_out = max_output_length

        # Store pad token id for convenience
        self.p_tok_pad_id = self.p_tok.tokenizer.pad_token_id if hasattr(self.p_tok, 'tokenizer') and hasattr(self.p_tok.tokenizer, 'pad_token_id') else 0
        self.r_tok_pad_id = self.r_tok.tokenizer.pad_token_id if hasattr(self.r_tok, 'tokenizer') and hasattr(self.r_tok.tokenizer, 'pad_token_id') else 0


    def get_prompt_inputs(self):
        # 1) Tokenize protein prompt
        tok = self.p_tok.tokenize(self.protein_str)
        ids  = tok.ids[:self.max_in]
        mask = tok.attention_mask[:self.max_in]

        # 2) Pad out to max_in
        pad_len = self.max_in - len(ids)
```

```python
        if pad_len > 0:
            ids  = ids  + [self.p_tok_pad_id] * pad_len # Use stored pad_id
            mask = mask + [0]                      * pad_len

        # 3) Wrap in batch dim
        input_ids      = torch.tensor([ids],  dtype=torch.long)
        attention_mask = torch.tensor([mask], dtype=torch.long)
        return input_ids, attention_mask

    def get_labels(self):
        # 1) Tokenize target RNA (convert to bjuz first)
        rna_bjuz = self.rna_str.replace("A", "b").replace("C", "j").replace("U", "u").replace("G", "z").replace(" ", "")
        tok = self.r_tok.tokenize(rna_bjuz)
        ids  = tok.ids[:self.max_out]

        # Labels for T5 are usually shifted, and padded tokens are -100
        # The T5 model itself handles shifting if decoder_input_ids are not provided with labels.
        # If we are providing labels for loss computation, they should be the target tokens.
        # Padded tokens in labels should be -100.

        # Pad labels to max_out length
        pad_len = self.max_out - len(ids)
        if pad_len > 0:
            ids = ids + [self.r_tok_pad_id] * pad_len # Pad with actual pad token id first

        labels_tensor = torch.tensor([ids], dtype=torch.long) # Renamed labels to labels_tensor
        labels_tensor[labels_tensor == self.r_tok_pad_id] = -100 # Replace padding with -100 for loss calculation

        return labels_tensor


class ProteinRNAGRPOTainer:
    def __init__(self, model_obj, tokenizer_p, tokenizer_r, train_envs_list, eval_envs_list, config_obj_trainer): # Renamed params
        self.model       = model_obj # Renamed
        self.tok_p       = tokenizer_p
        self.tok_r       = tokenizer_r
        self.train_envs  = train_envs_list # Renamed
        self.eval_envs   = eval_envs_list # Renamed
        self.config      = config_obj_trainer # Renamed


        # make a frozen reference policy (  _ref  )
        self.ref_model = copy.deepcopy(self.model).eval()
        for p in self.ref_model.parameters():
            p.requires_grad = False

        # the "old" policy   _old  , initially equals   _ref
        self.old_model = copy.deepcopy(self.ref_model)


        self.optimizer   = torch.optim.AdamW(self.model.parameters(), lr=self.config.learning_rate)



```

```
    def evaluate(self, step=0):
        """
        Calculate validation loss and rewards on the evaluation dataset
        """
        device = next(self.model.parameters()).device
        self.model.eval()
        total_loss = 0
        total_reward = 0
        num_batches = 0

        if not self.eval_envs: # Handle empty eval_envs
            print("No evaluation environments provided. Skipping evaluation.")
            self.model.train()
            return 0,0

        with torch.no_grad():
            for env in self.eval_envs:
                # Get inputs and labels
                input_ids, attention_mask = env.get_prompt_inputs()
                labels = env.get_labels()

                # Move to device
                input_ids = input_ids.to(device)
                attention_mask = attention_mask.to(device)
                labels = labels.to(device)

                # Forward pass for loss
                outputs = self.model(
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    labels=labels
                )

                loss = outputs.loss
                total_loss += loss.item()

                # Generate RNA sequence for reward calculation
                gen_output = self.model.generate( # Renamed gen to gen_output
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    max_new_tokens=self.config.max_completion_length,
                    num_return_sequences=1,
                    # Add other generation params if needed, e.g., top_p, temperature
                    do_sample=True, # Assuming sampling for reward evaluation
                    top_p=self.config.top_p,
                    temperature=self.config.temperature,
                )
                # decode expects list of IDs, gen_output[0] is a sequence of IDs (1D tensor)
                generated_rna_bjuz_list = self.tok_r.decode(gen_output) # decode from custom_tokenizer returns a list of strings
                generated_rna_bjuz = generated_rna_bjuz_list[0] if generated_rna_bjuz_list else ""

                # Convert bjuz to ACGU for rewarder
                generated_rna_acgu = generated_rna_bjuz.replace("b","A").replace("j","C").replace("u","U").
                    ","")

```

```python
                total_reward += rewarder(generated_rna_acgu, env.protein_str) # Use env.protein_str

                num_batches += 1

        avg_loss = total_loss / num_batches if num_batches > 0 else 0
        avg_reward = total_reward / num_batches if num_batches > 0 else 0

        # Log to wandb
        if wandb.run: # Check if wandb run is active
            wandb.log({
                "validation/loss": avg_loss,
                "validation/reward": avg_reward,
                "train/step": step # Use the global step passed to evaluate
            })

        self.model.train()
        return avg_loss, avg_reward

    def train(self, epochs=1):
        device    = next(self.model.parameters()).device
        num_gen   = self.config.num_generations
        max_new   = self.config.max_completion_length
        eps_clip  = self.config.EPSILON # Use from trainer's config object
        beta_val  = self.config.BETA    # Use from trainer's config object
        mu_val    = self.config.MU      # Use from trainer's config object

        # best_val_loss = float('inf') # Not used currently
        validation_interval = 100  # Validate every 100 steps
        global_step_counter = 0 # For periodic validation and logging

        for epoch in range(epochs):
            random.shuffle(self.train_envs)
            # Consider limiting train_envs for quicker epochs if dataset is large: self.train_envs[:200]
            for step_idx, env in enumerate(self.train_envs):
                # 0) prepare inputs for one protein (batch_size=1)
                input_ids, attention_mask = env.get_prompt_inputs()
                input_ids      = input_ids.to(device)
                attention_mask = attention_mask.to(device)
                protein_str_val = env.protein_str # Use env.protein_str

                # 1) Sample once from  _old  (no grad)
                # print(">>> Generating theta_old outputs for a protein")
                with torch.no_grad():
                    seq_old, logp_old = generate_with_log_probs(
                        model_obj=self.old_model, # Pass model_obj
                        input_ids=input_ids,
                        attention_mask=attention_mask,
                        num_return_sequences=num_gen,
                        max_new_tokens=max_new,
                        top_p=self.config.top_p,
                        temperature=self.config.temperature,
                    )
                # print(">>> theta_old output generation is completed")
                N, T_len = seq_old.shape # Renamed T to T_len to avoid conflict
```

```

            # 3.1) build decoder_input_ids for teacher forcing
            start_tok = self.model.config.decoder_start_token_id
            start_col = torch.full((N,1), start_tok,
                                   device=device, dtype=torch.long)
            dec_inp    = torch.cat([start_col, seq_old[:, :-1]], dim=1)  # (N, T_len)

            # 3.2) Prepare encoder inputs for batch N
            enc_ids_repeated = input_ids.repeat_interleave(N, dim=0) # Renamed enc_ids
            enc_mask_repeated= attention_mask.repeat_interleave(N, dim=0) # Renamed enc_mask

            # print(">>> Generating ref outputs for a protein")
            with torch.no_grad(): # ref_model does not require grad
                out_ref    = self.ref_model(
                    input_ids=enc_ids_repeated,
                    attention_mask=enc_mask_repeated,
                    decoder_input_ids=dec_inp,
                )
            logits_ref= out_ref.logits                              # (N, T_len, V)
            logp_ref   = F.log_softmax(logits_ref, -1)          # (N, T_len, V)
            logp_ref   = logp_ref.gather(-1, seq_old.unsqueeze(-1))\
                                .squeeze(-1).detach()          # (N, T_len), ensure detached
            # print(">>> ref output generation is completed")

            # 2) Compute rewards & advantages (detach)
            # print(">>> Started calculating rewards and advantages")
            # tok_r.decode expects a batch of sequences (list of list of IDs or Tensor)
            # seq_old is already a batch of sequences (Tensor)
            decoded_seq_old_bjuz_list = self.tok_r.decode(seq_old) # Returns list of bjuz strings
            # Convert bjuz to ACGU for rewarder; rewarder expects list of char lists or list of strings
            decoded_seq_old_acgu_char_lists = [[char for char in seq.replace("b", "A").replace("j", "C").replace("u", "U").replace("z", "G").replace(" ", "")] for seq in decoded_seq_old_bjuz_list]

            # Pass seq_old (token_ids on device), decoded_seq_old_acgu_char_lists (Python list of lists of chars), protein_str_val
            rewards_tensor = calculate_reward(seq_old, decoded_seq_old_acgu_char_lists, input_ids, protein_str_val, pad_token_id=self.tok_r.tokenizer.pad_token_id, eos_token_id=self.tok_r.tokenizer.eos_token_id)
            advantages = calculate_advantage(rewards_tensor).detach()

            # Log reward stats to wandb
            non_zero_rewards = rewards_tensor[rewards_tensor != 0].flatten()
```

```
if wandb.run: # Check if wandb run is active
    if non_zero_rewards.numel() > 0:
        wandb.log({
            "reward/mean": non_zero_rewards.mean().item(),
            "reward/std": non_zero_rewards.std().item(),
            "reward/max": non_zero_rewards.max().item(),
            "reward/min": non_zero_rewards.min().item(),
            "train/step": global_step_counter
        })
    else:
         wandb.log({
            "reward/mean": 0.0, "reward/std": 0.0, "reward/max": 0.0, "reward/min": 0.0,
            "train/step": global_step_counter
        })
# print(">>> Ended calculating rewards and advantages")

# 3) Inner    GRPO updates on   _
for inner_loop_idx in range(mu_val): # Renamed inner to inner_loop_idx
    out_new = self.model( # Renamed out to out_new
        input_ids=enc_ids_repeated,
        attention_mask=enc_mask_repeated,
        decoder_input_ids=dec_inp,
    )
    logits_new = out_new.logits                                  # (N, T_len, V)
    logp_new   = F.log_softmax(logits_new, -1)                   # (N, T_len, V)
    logp_new   = logp_new.gather(-1, seq_old.unsqueeze(-1))\
                                .squeeze(-1)                     # (N, T_len)

    # 3.3) clipped surrogate
    ratio   = torch.exp(logp_new - logp_old.detach()) # Detach logp_old, it's from old_model
    clipped = ratio.clamp(1 - eps_clip, 1 + eps_clip)
    loss_pg = - torch.mean(
        torch.minimum(ratio * advantages,
                      clipped * advantages)
    )

    # 3.4)  K L penalty  using f(u)=u - log u - 1
    if beta_val > 0:
        u         = torch.exp(logp_ref - logp_new) # logp_ref is already detached
        kl_term   = torch.mean(u - (logp_ref - logp_new) - 1)
        loss_total = loss_pg + beta_val * kl_term # Renamed loss to loss_total
    else:
        kl_term = torch.tensor(0.0, device=device)
        loss_total = loss_pg

    # 3.5) debug prints (optional, can be verbose)
    # print(f"Epoch {epoch+1} Step {step_idx} Inner {inner_loop_idx} | "
    #       f"pg_loss={loss_pg.item():.4f} kl={kl_term.item():.4f}")
```

```python

                    if wandb.run: # Check if wandb run is active
                        wandb.log({
                            "loss/pg_loss": loss_pg.item(),
                            "loss/kl_term": kl_term.item(),
                            "loss/total": loss_total.item(),
                            "train/step": global_step_counter # Log with global step
                        })

                    # 3.6) backprop & step
                    self.optimizer.zero_grad()
                    loss_total.backward()
                    torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=self.config.max_grad_norm)
                    self.optimizer.step()

                global_step_counter += 1 # Increment global step

                # Run validation periodically
                if self.eval_envs and global_step_counter % validation_interval == 0:
                    val_loss, val_reward = self.evaluate(step=global_step_counter) # Pass global_step_counter
                    print(f"Global Step {global_step_counter} Validation: loss={val_loss:.4f} reward={val_reward:.4f}")

                # 4) sync  _old  _
                self.old_model.load_state_dict(self.model.state_dict())

            print(f"--- End Epoch {epoch+1} ---")

        # if wandb.run: # Check if wandb run is active # Moved finish to main
        #     wandb.finish()


def load_data(path):
    prots, rnas = [], []
    with open(path) as f:
        for line in f:
            line = line.strip()
            if '$' not in line: continue
            try:
                p, r = line.split('$',1) # Split only on the first '$'
                prots.append(p); rnas.append(r)
            except ValueError:
                print(f"Skipping malformed line in data file: {line}") # Handle lines without '$' if strip didn't catch it
    return prots, rnas

if __name__ == '__main__':
    # Initialize wandb once for the entire script run
    # Hyperparameter tuning might reinitialize runs if configured to do so.
    wandb.init(project=global_config_module.WANDB_PROJECT, name="grpo-main-run", reinit=True) # Allow reinit for tuning

    parser = argparse.ArgumentParser()
    # Arguments from global_config_module as defaults
    parser.add_argument('--train_data', default=global_config_module.TRAIN_DATA)
```

```
    parser.add_argument('--eval_data', default=global_config_module.EVAL_DATA)
    parser.add_argument('--model_path', default=global_config_module.MODEL_PATH)
    parser.add_argument('--protein_tokenizer', default=global_config_module.PROTEIN_TOKENIZER)
    parser.add_argument('--rna_tokenizer', default=global_config_module.RNA_TOKENIZER)
    parser.add_argument('--output_dir', default='./results_grpo') # Different default from CustomGRPOConfig
    parser.add_argument('--epochs', type=int, default=global_config_module.NUM_TRAIN_EPOCHS)
    parser.add_argument('--learning_rate', type=float, default=global_config_module.LEARNING_RATE)
    parser.add_argument('--max_input_length', type=int, default=global_config_module.MAX_PROMPT_LENGTH)
    parser.add_argument('--max_completion_length', type=int, default=global_config_module.MAX_COMPLETION_LENGTH)
    parser.add_argument('--num_generations', type=int, default=global_config_module.NUM_GENERATIONS)
    # top_p, temperature, max_grad_norm are not in global_config_module, use CustomGRPOConfig defaults or add to global_config_module
    parser.add_argument('--top_p', type=float, default=1.0)
    parser.add_argument('--temperature', type=float, default=1.0)
    parser.add_argument('--max_grad_norm', type=float, default=1.0)

    # GRPO specific params from global_config_module
    parser.add_argument('--beta', type=float, default=global_config_module.BETA)
    parser.add_argument('--epsilon', type=float, default=global_config_module.EPSILON)
    parser.add_argument('--mu', type=int, default=global_config_module.MU)

    parser.add_argument('--tune_hyperparameters', action='store_true', help='Run hyperparameter tuning with Optuna')
    parser.add_argument('--n_trials', type=int, default=20, help='Number of trials for hyperparameter tuning')
    args = parser.parse_args()

    # Setup
    random.seed(42); np.random.seed(42); torch.manual_seed(42)
    if torch.cuda.is_available(): torch.cuda.manual_seed_all(42) # For GPU reproducibility

    # Tokenizers & Model
    tok_p = get_tokenizer("bpe", global_config_module.VOCAB_SIZE, global_config_module.MAX_LENGTH, tokenizer_path=args.protein_tokenizer)
    tok_r = get_tokenizer("bpe", global_config_module.VOCAB_SIZE, global_config_module.MAX_LENGTH, tokenizer_path=args.rna_tokenizer)

    # Ensure pad token IDs are set if BpeTokenizer class doesn't handle it automatically after load
    # The BpeTokenizer in `classifier/util/tokenizer.py` and `grpo-trainer/custom_tokenizer.py`
    # sets special tokens using a dict. Let's assume pad_token_id is accessible.
    # The ProteinRNAEnvironment needs these.
    # tok_p.pad_token_id = tok_p.tokenizer.token_to_id("<pad>") # Or however it's stored
    # tok_r.pad_token_id = tok_r.tokenizer.token_to_id("<pad>")
```

```python

    model = T5ForConditionalGeneration.from_pretrained(args.model_path).to('cuda' if torch.cuda.is_available() else 'cpu')

    # Data Envs
    prots_train, rnas_train = load_data(args.train_data)
    train_envs = [ProteinRNAEnvironment(p, r, tok_p, tok_r, args.max_input_length, args.max_completion_length) for p, r in zip(prots_train, rnas_train)]
    eval_envs = []
    if args.eval_data and os.path.exists(args.eval_data):
        prots_eval, rnas_eval = load_data(args.eval_data)
        eval_envs = [ProteinRNAEnvironment(p, r, tok_p, tok_r, args.max_input_length, args.max_completion_length) for p, r in zip(prots_eval, rnas_eval)]
    print(f"Created {len(train_envs)} training environments and {len(eval_envs)} evaluation environments.")

    # Config for GRPO trainer
    # Pass CLI args to override defaults in CustomGRPOConfig
    grpo_trainer_config = CustomGRPOConfig(
        output_dir=args.output_dir,
        learning_rate=args.learning_rate,
        max_prompt_length=args.max_input_length,
        max_completion_length=args.max_completion_length,
        num_generations=args.num_generations,
        top_p=args.top_p,
        temperature=args.temperature,
        max_grad_norm=args.max_grad_norm,
        BETA=args.beta,
        EPSILON=args.epsilon,
        MU=args.mu
    )

    if args.tune_hyperparameters:
        print("Starting hyperparameter tuning...")
        from hyperparameter_tuning import run_hyperparameter_tuning # Assuming it's in the same directory

        # The hyperparameter_tuning script will modify the BETA, EPSILON, MU attributes
        # of the config object passed to it.
        best_params = run_hyperparameter_tuning(
            model=model,
            tok_p=tok_p,
            tok_r=tok_r,
            train_envs=train_envs,
            eval_envs=eval_envs,
            tuning_config=grpo_trainer_config, # Pass the trainer's config object
            n_trials=args.n_trials
        )

        # Update grpo_trainer_config with best parameters found by Optuna study
        # The run_hyperparameter_tuning function already modifies the tuning_config object's attributes
        # if designed to do so, or returns best_params.
        # If it returns params, apply them:
        if best_params:
             grpo_trainer_config.BETA = best_params.get('beta', grpo_trainer_config.BETA)
             grpo_trainer_config.EPSILON = best_params.get('epsilon', grpo_trainer_config.EPSILON)
```

```
            grpo_trainer_config.MU = best_params.get('mu', grpo_trainer_config.MU)

        print("Best hyperparameters from tuning:")
        print(f"BETA: {grpo_trainer_config.BETA}, EPSILON: {grpo_trainer_config.EPSILON}, MU: {grpo_trainer_config.MU}")
        wandb.config.update(best_params, allow_val_change=True) # Log best params to current wandb run

    # Trainer
    print("Starting training with config:")
    print(vars(grpo_trainer_config)) # Print the config being used

    trainer = ProteinRNAGRPOTainer(model, tok_p, tok_r, train_envs, eval_envs, grpo_trainer_config)
    trainer.train(epochs=args.epochs)

    if eval_envs: # Final evaluation after training
       print("Performing final evaluation...")
       final_val_loss, final_val_reward = trainer.evaluate(step=trainer.config.num_train_epochs * len(train_envs)) # Approx global step
       print(f"Final Validation: loss={final_val_loss:.4f} reward={final_val_reward:.4f}")

    if wandb.run: # Ensure wandb run is finished
        wandb.finish()
```

File: grpo-trainer/reward_functions.py

Listing 14: Code Snippet

```
import numpy as np
from typing import List, Tuple, Dict, Any, Optional, Union
import RNA  # ViennaRNA package for MFE calculation
import config as global_config_module # Use an alias
import get_binding_score

def calculate_gc_content(sequence: str) -> float:
    if not sequence: # Check for empty sequence
        return 0.0
    gc_count = sequence.upper().count('G') + sequence.upper().count('C') # Ensure uppercase
    total_count = len(sequence)
    return gc_count / total_count if total_count > 0 else 0.0

def calculate_gc_content_reward(sequence: str) -> float:
    """
    Calculate reward based on GC content.
    Args: sequence: RNA sequence (ACGU)
    Returns: Reward value between 0 and 1
    """
    if not sequence: return 0.0
    gc_content = calculate_gc_content(sequence)

    gc_min = global_config_module.GC_CONTENT_MIN
    gc_max = global_config_module.GC_CONTENT_MAX

    if gc_min <= gc_content <= gc_max:
        return 1.0
    else:
        if gc_content < gc_min:
            distance = gc_min - gc_content
```

```python
            max_distance_possible = gc_min # from 0 to gc_min
            # Reward decreases linearly from 1 (at gc_min) to 0 (at 0 GC content)
            # Ensure max_distance_possible is not zero
            return max(0.0, 1.0 - (distance / max_distance_possible if max_distance_possible > 1e-6 else 1.0))
        else:  # gc_content > gc_max
            distance = gc_content - gc_max
            max_distance_possible = 1.0 - gc_max # from gc_max to 1.0
            # Reward decreases linearly from 1 (at gc_max) to 0 (at 100% GC content)
            return max(0.0, 1.0 - (distance / max_distance_possible if max_distance_possible > 1e-6 else 1.0))


def calculate_mfe(sequence: str) -> float:
    """
    Calculate Minimum Free Energy for an RNA sequence using ViennaRNA.
    Args: sequence: RNA sequence (ACGU)
    Returns: MFE value (lower is better for stability)
    """
    if not sequence or not all(c in "ACGU" for c in sequence.upper()): # Check for valid RNA sequence
        # print(f"Warning: Invalid RNA sequence for MFE: {sequence}")
        return 0.0 # Or handle error appropriately, MFE is usually negative

    try:
        # ViennaRNA might be sensitive to non-ACGU characters or very short sequences
        if len(sequence) < 4: # Arbitrary minimum length for folding, adjust as needed
             return 0.0
        _, mfe_val = RNA.fold(sequence.upper()) # Use a different name, mfe_val, ensure uppercase
        return mfe_val
    except Exception as e:
        # print(f"Error calculating MFE for '{sequence}': {e}")
        return 0.0 # Return a neutral value or error indicator

def calculate_mfe_reward(sequence: str) -> float:
    """
    Calculate reward based on MFE value. Normalized by length.
    Lower MFE means more stable structure, which is better.
    Args: sequence: RNA sequence (ACGU)
    Returns: Reward value between 0 and 1
    """
    if not sequence:
        return 0.0

    mfe_val = calculate_mfe(sequence) # Renamed mfe to mfe_val

    if len(sequence) == 0: # Should have been caught by calculate_mfe, but double check
        return 0.0

    length_normalized_mfe = mfe_val / len(sequence)

    # Target is highly negative MFE (e.g., -0.5 kcal/mol/nt is good)
    # Let's define a target normalized MFE, e.g., -0.5
    # and a worst MFE, e.g., 0 or slightly positive.
    # Reward can be 1 if length_normalized_mfe <= target_norm_mfe
    # Reward can be 0 if length_normalized_mfe >= worst_norm_mfe
    # Linear interpolation in between.

```

```python
    target_norm_mfe = -0.5 # Example: very stable
    worst_norm_mfe  =  0.0 # Example: unstable or unfolded

    if length_normalized_mfe <= target_norm_mfe:
        return 1.0
    elif length_normalized_mfe >= worst_norm_mfe:
        return 0.0
    else: #  target_norm_mfe < length_normalized_mfe < worst_norm_mfe
        # Linearly map from [worst_norm_mfe, target_norm_mfe] to [0, 1]
        # Since target is more negative, (val - worst) / (target - worst)
        return (length_normalized_mfe - worst_norm_mfe) / (target_norm_mfe - worst_norm_mfe)


def calculate_binding_affinity(protein_sequence: str, rna_sequence: str) -> float:
    """
    Placeholder for binding affinity calculation.
    Args: protein_sequence, rna_sequence (ACGU)
    Returns: Binding affinity score between 0 and 1
    """
    # This should be replaced with a real model prediction
    if not protein_sequence or not rna_sequence:
        return 0.0

    return get_binding_score(protein_sequence, rna_sequence)

def calculate_token_level_reward(
    protein_sequence: str,
    partial_rna_sequence: str, # ACGU format
    position: int, # Current length of partial_rna_sequence
    max_length: int # Max possible length of RNA
) -> float:
    """
    Calculate combined reward for a partial RNA sequence at a specific position.
    """
    # Ensure partial_rna_sequence is not empty for component reward functions
    if not partial_rna_sequence:
        return 0.0

    gc_reward_val = calculate_gc_content_reward(partial_rna_sequence) # Renamed gc_reward
    mfe_reward_val = calculate_mfe_reward(partial_rna_sequence) # Renamed mfe_reward
    binding_reward_val = calculate_binding_affinity(protein_sequence, partial_rna_sequence) # Renamed binding_reward

    # Weighting rewards (can be dynamic based on progress if desired)
    # progress = position / max_length if max_length > 0 else 0
    # Example: fixed weights from global config
    combined_reward_val = ( # Renamed combined_reward
        global_config_module.BINDING_AFFINITY_WEIGHT * binding_reward_val +
        global_config_module.GC_CONTENT_WEIGHT * gc_reward_val +
        global_config_module.MFE_WEIGHT * mfe_reward_val
    )

    # Normalize combined reward to be between 0 and 1 if sum of weights can exceed 1
```

```python
    # sum_weights = global_config_module.BINDING_AFFINITY_WEIGHT +
        global_config_module.GC_CONTENT_WEIGHT +
        global_config_module.MFE_WEIGHT
    # if sum_weights > 0:
    #     combined_reward_val = combined_reward_val / sum_weights
    # else:
    #     combined_reward_val = 0.0

    return max(0.0, min(1.0, combined_reward_val)) # Ensure it's
        within [0,1]


def rewarder(partial_rna_acgu: str, protein_sequence_str: str,
    protein_ids=None, rna_ids=None) -> float: # Renamed params
    """
    Compute the GRPO token-level reward for a partial RNA sequence
        (ACGU)
    given its protein prompt.
    """
    current_pos = len(partial_rna_acgu) # Renamed position to
        current_pos
    max_len_val = global_config_module.MAX_COMPLETION_LENGTH #
        Renamed max_len

    # Compute the combined token level reward
    reward_value = calculate_token_level_reward( # Renamed reward to
        reward_value
        protein_sequence=protein_sequence_str, # Pass renamed param
        partial_rna_sequence=partial_rna_acgu, # Pass renamed param
            (ACGU)
        position=current_pos,
        max_length=max_len_val
    )
    return float(reward_value)
```

File: grpo-trainer/.DS_Store

This is a system file (.DS_Store) generated by macOS. It typically contains custom attributes of its containing folder, such as the position of icons or the choice of a background image. Its content is not relevant as source code and is therefore omitted.

Directory: grpo-trainer/util/

File: grpo-trainer/util/create_val.py

Listing 15: Code Snippet

```python
import pandas as pd
import random

# Set random seed for reproducibility
random.seed(42)

# Read the CSV file
# Ensure 'positive_sequences.csv' is in the correct path or provide
    full path
try:
    df = pd.read_csv('positive_sequences.csv')
except FileNotFoundError:
    print("Error: 'positive_sequences.csv' not found. Please check
        the path.")
    exit()

```

```python

# Remove rows with missing RNA or protein sequences
df = df.dropna(subset=['protein_seq', 'rna_sequence'])

# Keep only protein_seq and rna_seq columns
df_selected = df[['protein_seq', 'rna_sequence']].copy() # Use
    .copy() to avoid SettingWithCopyWarning

# Find the longest RNA sequence (on the selected dataframe)
if not df_selected.empty:
    df_selected['rna_length'] =
        df_selected['rna_sequence'].astype(str).str.len() # Ensure
        rna_sequence is string
    max_rna_length = df_selected['rna_length'].max()
    # Get the first such longest sequence if multiple exist
    longest_rna_series = df_selected.loc[df_selected['rna_length'] ==
        max_rna_length, 'rna_sequence']
    longest_rna = longest_rna_series.iloc[0] if not
        longest_rna_series.empty else "N/A"
else:
    max_rna_length = 0
    longest_rna = "N/A"
    print("DataFrame is empty after dropping NA values.")


# Randomly select 100 rows for validation, if enough rows exist
num_samples = min(100, len(df_selected))
if num_samples > 0:
    validation_df = df_selected.sample(n=num_samples,
        random_state=42).drop(columns=['rna_length'], errors='ignore')
    # Save to validation_rl.txt (protein,rna format without header
        for some tools)
    # The original inference.py splits by ',', so CSV format is
        expected
    validation_df.to_csv('validation_rl.txt', index=False,
        header=False) # header=False if inference.py expects no header
    print(f"Successfully created validation_rl.txt with
        {len(validation_df)} sequences.")
else:
    print("Not enough data to create validation_rl.txt")


# Print summary
print(f"Original dataset had {len(df)} rows, {len(df_selected) if
    'df_selected' in locals() else 0} valid sequences after
    cleaning.")
print(f"Longest RNA sequence length: {max_rna_length}")
print(f"Example of longest RNA sequence: {str(longest_rna)[:50]}...")
```

File: grpo-trainer/util/create_validation.py

Listing 16: Code Snippet

```python
import pandas as pd
import random

# Set random seed for reproducibility
random.seed(42)

# Read the CSV file
# Ensure 'positive_sequences.csv' is in the correct path or provide
    full path
try:
    df = pd.read_csv('positive_sequences.csv')
```

```python
except FileNotFoundError:
    print("Error: 'positive_sequences.csv' not found. Please check the path.")
    exit()

# Remove rows with missing protein sequences (and rna_seq for safety, though not explicitly checked before)
df = df.dropna(subset=['protein_seq', 'rna_seq'])


# Keep only protein_seq and rna_seq columns
df_selected = df[['protein_seq', 'rna_seq']].copy() # Use .copy() to avoid SettingWithCopyWarning

# Find the longest RNA sequence
if not df_selected.empty:
    # Ensure rna_seq is treated as string for .str accessor
    df_selected['rna_length'] = df_selected['rna_seq'].astype(str).str.len()
    max_rna_length = df_selected['rna_length'].max()
    longest_rna_series = df_selected.loc[df_selected['rna_length'] == max_rna_length, 'rna_seq']
    longest_rna = longest_rna_series.iloc[0] if not longest_rna_series.empty else "N/A"
else:
    max_rna_length = 0
    longest_rna = "N/A"
    print("DataFrame is empty after dropping NA values.")


# Randomly select 100 rows, if enough rows exist
num_samples = min(100, len(df_selected))
if num_samples > 0:
    validation_df = df_selected.sample(n=num_samples, random_state=42).drop(columns=['rna_length'], errors='ignore')
    # Save to validation_rl.txt (protein,rna format, no header)
    # inference.py in the main directory splits by ',', so CSV format is fine.
    validation_df.to_csv('validation_rl.txt', index=False, header=False) # header=False if inference.py expects no header

    print(f"Successfully created validation_rl.txt with {len(validation_df)} sequences.")
else:
    print("Not enough data to create validation_rl.txt")


print(f"Original dataset had {len(df)} rows, {len(df_selected) if 'df_selected' in locals() else 0} valid sequences after cleaning.")
print(f"Longest RNA sequence length: {max_rna_length}")
print(f"Example of longest RNA sequence: {str(longest_rna)[:50]}...") # Print first 50 characters
```

File: grpo-trainer/util/generate_data.py

Listing 17: Code Snippet

```python
import os
import csv
from tqdm import tqdm  # progress bar library

# ------------------------------------------------------------------------------
```

```python
# 1) Define paths
# -----------------------------------------------------------------------------
FASTA_FOLDER = "/data6/helya/dataset/CLIPdb_cluster/cd_hit_results_RBPs/identity_90"
PROTEIN_CSV = "/data6/alpsencer/reinforce_rna/protein_seqs.csv"  # CSV with columns: prot_name, seq
OUTPUT_CSV  = "./output.csv" # Output in the current directory of this script

# -----------------------------------------------------------------------------
# 2) Read the protein sequences into a dictionary (case-insensitive)
# -----------------------------------------------------------------------------
protein_dict = {}
try:
    with open(PROTEIN_CSV, "r", newline="") as f_in:
        reader = csv.DictReader(f_in)
        if 'prot_name' not in reader.fieldnames or 'seq' not in reader.fieldnames:
            print(f"Error: PROTEIN_CSV '{PROTEIN_CSV}' must have 'prot_name' and 'seq' columns.")
            exit()
        for row in reader:
            prot_name = row["prot_name"].strip().lower()  # using lowercase keys
            protein_dict[prot_name] = row["seq"]
except FileNotFoundError:
    print(f"Error: Protein CSV file not found at '{PROTEIN_CSV}'")
    exit()

# -----------------------------------------------------------------------------
# 3) Process FASTA files and write to CSV
# -----------------------------------------------------------------------------
if not os.path.isdir(FASTA_FOLDER):
    print(f"Error: FASTA folder not found at '{FASTA_FOLDER}'")
    exit()

with open(OUTPUT_CSV, "w", newline="") as f_out:
    writer = csv.writer(f_out)
    writer.writerow(["protein_name", "protein_seq", "rna_sequence", "strand", "pair"])

    # Get all .fa files
    fasta_files = [f for f in os.listdir(FASTA_FOLDER) if f.endswith(".fa") or f.endswith(".fasta")]

    if not fasta_files:
        print(f"No FASTA files found in '{FASTA_FOLDER}'.")

    # Loop over the files with a progress bar
    for file_name in tqdm(fasta_files, desc="Processing FASTA files"):
        # Derive protein name from file name (first token before underscore)
        protein_name_raw = file_name.split("_")[0]
        protein_seq = protein_dict.get(protein_name_raw.lower(), "NA") # Default to "NA" if not found
        fa_path = os.path.join(FASTA_FOLDER, file_name)

        current_rna_sequence = ""
        header_info = "" # Store the most recent header

```

```
        # Read all lines from the FASTA file
        try:
            with open(fa_path, "r") as fa_in:
                for line in fa_in:
                    line = line.strip()
                    if not line: continue # Skip empty lines

                    if line.startswith(">"):
                        # If we have a sequence buffered, process it
                            with previous header
                        if current_rna_sequence and header_info:
                            strand = "NA"
                            # Process header_info for strand
                            temp_header = header_info
                            if temp_header.endswith("(+)"):
                                strand = "+"
                                temp_header = temp_header[:-3].strip()
                            elif temp_header.endswith("(-)"):
                                strand = "-"
                                temp_header = temp_header[:-3].strip()
                            writer.writerow([protein_name_raw,
                                protein_seq, current_rna_sequence,
                                strand, "+"])

                        # Start new sequence
                        header_info = line[1:] # Store new header
                            without '>'
                        current_rna_sequence = ""
                    else:
                        current_rna_sequence += line # Append
                            sequence lines

                # Process the last sequence in the file
                if current_rna_sequence and header_info:
                    strand = "NA"
                    temp_header = header_info
                    if temp_header.endswith("(+)"):
                        strand = "+"
                        temp_header = temp_header[:-3].strip()
                    elif temp_header.endswith("(-)"):
                        strand = "-"
                        temp_header = temp_header[:-3].strip()
                    writer.writerow([protein_name_raw, protein_seq,
                        current_rna_sequence, strand, "+"])

        except Exception as e:
            print(f"Error processing file {file_name}: {e}")
            continue # Skip to next file

        # Optional: print progress for each file (can be too verbose
            with tqdm)
        # print(f"Processed file: {file_name}")
print(f"Processing complete. Output written to {OUTPUT_CSV}")
```

File: grpo-trainer/util/test_tokenizer.py

Listing 18: Code Snippet

```
from transformers import PreTrainedTokenizerFast
from itertools import islice
import os # For checking file existence

TOKENIZER_FILE_PATH =
    "/data6/alpsencer/tokenizers/bpe_protein_1000_1024.json"
```

```python
TRAIN_DATA_PATH = "/data6/alpsencer/reinforce_rna/sample_1K.txt" # Renamed TRAIN_DATA to TRAIN_DATA_PATH


if not os.path.exists(TOKENIZER_FILE_PATH):
    print(f"Error: Tokenizer file not found at {TOKENIZER_FILE_PATH}")
    exit()

tokenizer = PreTrainedTokenizerFast(
    tokenizer_file=TOKENIZER_FILE_PATH,
    unk_token="<unk>",
    pad_token="<pad>",
    eos_token="</s>", # EOS token if your model uses it for proteins
    bos_token="<s>", # BOS token if your model uses it
    model_max_length=1024,
    padding_side="left", # Ensure this matches model's expectation for T5 encoder
)

# T5 usually expects a sentence-piece like format, often with EOS at the end of input.
# If your BPE tokenizer was trained without explicit EOS for protein inputs,
# this setup is fine. The main thing is consistency with training.

if not os.path.exists(TRAIN_DATA_PATH):
    print(f"Error: Train data file not found at {TRAIN_DATA_PATH}")
    exit()

print(f"Using tokenizer: {TOKENIZER_FILE_PATH}")
print(f"Pad token: '{tokenizer.pad_token}', ID: {tokenizer.pad_token_id}")
print(f"EOS token: '{tokenizer.eos_token}', ID: {tokenizer.eos_token_id}")
print(f"UNK token: '{tokenizer.unk_token}', ID: {tokenizer.unk_token_id}")
print(f"BOS token: '{tokenizer.bos_token}', ID: {tokenizer.bos_token_id}")


with open(TRAIN_DATA_PATH) as f:
    for line_num, line_content in enumerate(islice(f, 5)): # Test a few lines
        line_content = line_content.strip()
        if not line_content or "$" not in line_content:
            print(f"Skipping malformed line {line_num+1}: {line_content}")
            continue

        # Assuming format is protein$rna or protein$something_else
        protein_seq, _ = line_content.split("$", 1) # Renamed protein to protein_seq

        # For T5, inputs are often not lowercased by default unless normalizer does it.
        # The custom BpeTokenizer class had Lowercase normalizer.
        # PreTrainedTokenizerFast directly from file might or might not apply it
        # depending on how the tokenizer.json was saved. Assume it's handled or not critical for this test.

        # Some tokenizers add special tokens automatically, some don't.
        # For T5, usually you'd add EOS to the protein input if it was trained that way.
```

```
        # protein_input_for_tokenizer = protein_seq + tokenizer.eos_token # Example if EOS is needed
        protein_input_for_tokenizer = protein_seq

        enc = tokenizer(
            protein_input_for_tokenizer,
            truncation=True,
            padding="max_length", # Pad to model_max_length
            max_length=tokenizer.model_max_length, # Use tokenizer's max_length
            return_tensors=None, # Get lists of IDs
            add_special_tokens=True # Let tokenizer handle BOS/EOS based on its config
        )

        input_ids_list = enc["input_ids"] # Renamed ids to input_ids_list
        attention_mask_list = enc["attention_mask"] # Renamed mask to attention_mask_list

        # Calculate real length based on attention mask
        # For left padding, actual tokens are at the end.
        try:
            first_real_token_idx = attention_mask_list.index(1)
            real_len = len(input_ids_list) - first_real_token_idx
            real_ids = input_ids_list[first_real_token_idx:]
        except ValueError: # If no 1 in attention_mask (e.g. all padding)
            real_len = 0
            real_ids = []


        print(f"\n=== PROTEIN PROMPT (Line {line_num+1}) ===")
        print("Original:", protein_seq[:60] + ("..." if len(protein_seq) > 60 else ""))
        # print("Input to tokenizer:", protein_input_for_tokenizer[:60] + ("..." if len(protein_input_for_tokenizer) > 60 else ""))

        print("\nTokenized IDs (first 10 of padded):", input_ids_list[:10])
        print("Tokenized IDs (last 10 of padded):", input_ids_list[-10:])
        print("Attention Mask (first 10):", attention_mask_list[:10])
        print("Attention Mask (last 10):", attention_mask_list[-10:])

        print("\n[Padded left     real tokens at end]")
        if real_ids:
            print("Real IDs (first 10):", real_ids[:10])
            print("Real IDs (last 10):", real_ids[-10:])
            print("Real Tokens (first 10):", tokenizer.convert_ids_to_tokens(real_ids[:10]))
            print("Real Tokens (last 10):", tokenizer.convert_ids_to_tokens(real_ids[-10:]))
            print("Real token count:", real_len)
            decoded_real = tokenizer.decode(real_ids, skip_special_tokens=False) # See special tokens
            decoded_real_skip_special = tokenizer.decode(real_ids, skip_special_tokens=True)
            print("Decoded (real tokens, keep special):", decoded_real[:60] + ("..." if len(decoded_real) > 60 else ""))
            print("Decoded (real tokens, skip special):", decoded_real_skip_special[:60] + ("..." if len(decoded_real_skip_special) > 60 else ""))
```

```
        else:
            print("No real tokens found (all padding or empty
                input).")
        print("
        # break  # remove 'break' if you want to sample more lines
```