Universiteit Leiden
Opleiding Informatica

RLIF: Reinforcement learning-based RNA design tool

Name: Andrius Bernatavicius
Studentnr: s1782975

Date: 12/10/2019

1st supervisor: Erwin Bakker
2nd supervisor: Alexander Goultiaev

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

RNA is a biopolymer that plays an essential role in all biological systems by regulating various vital processes. Research in recent years has shown the applicability of synthetic RNA structures in the fields of of nanomaterial engineering and gene therapeutics. The specific function of an RNA molecule depends on its structural elements which is in turn is determined by the string of nucleotides that it is made of. Therefore the prediction of nucleotide sequences that will fold into a specific shape, called RNA inverse folding is gaining more interest. In this thesis we introduce a new algorithm for the problem - RLIF - a reinforcement learning based method that produces state-of-the-art results on various benchmarks both in terms of speed and accuracy. A comparison between six other RNA inverse folding methods on several benchmark RNA structure datasets illustrates these performance gains. We also introduce a custom graphical interface for designing RNA molecules, that extends the functionality of RLIF algorithm and integrates it with various RNA sequence analysis tools.

Keywords— RNA Inverse Folding, Reinforcement Learning, RNA Design.

Aknowledgements

I would like to thank my supervisors Dr. Erwin Bakker and Prof. Alexander Goultiaev for their insight and support throughout the project. I would also like to thank Dr. Çaglar Senaras and Mariana Goldak for their guidance during my internship at Vanderlande Industries, in which I learned a lot about reinforcement learning. Lastly I would like to thank Andreas Papagiannis for his ideas and suggestions.

Contents

1 Introduction

RNA is a biopolymer that plays an essential role in all biological systems. By acting as a transmitter of information between the genome and the proteome, RNA mediates the vast majority of main functions within cells. Advances in next generation sequencing technologies are uncovering even more roles that RNA has. A subset of RNA sequences called long non-coding RNAs (lncRNAs) seem to be significant importance for a diverse range of cellular regulation processes. Based on conservative estimates at least 7% of all transcripts from human genome are non-protein coding, they seem to be regulated developmentally and their loci are distant from protein-coding regions within the genome. [1] This indicates that these sequences are being intentionally transcribed and are responsible for specific important functions within cells. So far lncRNAs have been identified as key components in transcription regulation, messenger RNA processing, translation mediation, protein activity regulation and various other signalling functions. [2]

Since the functions that RNA sequences can perform are based mainly on their structural conformation, a lot of research effort is going into uncovering more details about RNA structure.

1.1 RNA Structure

RNA primary structure consists of a chain of nucleotide bases that are linked by a phosphate backbone. Due to thermodynamic and electrostatic interactions, the strand of RNA folds itself and forms a complicated three dimensional structure. Hydrogen bonds form between G-C, A-U, which are called Watson-Crick pairs and slightly less stable wobble pairs between G-U and U-G. Stacks of paired nucleotides form helices or stems while unpaired nucleotides form various loop structures that are categorized based on the shapes or motifs that they create. A large variety of tertiary structural motifs are also categorized, however tertiary RNA structure prediction still remains a very hard computational problem. Therefore a lot of effort in the field focuses on predicting the secondary RNA structure as accurately as possible instead. Figure 1 summarizes the most frequent structural motifs found in secondary RNA structures.

Dot-bracket notation

RNA structure is made from two main elements: base pairs and unpaired bases. Therefore the secondary RNA structure can be translated to a string format using these two distinct element types. Nucleotides that form pairs are denoted as brackets - " (" or ") " which indicate an openings and closings of stems/helices respectively, while unpaired nucleotides are denoted as dots - " . " and are used to represent various other structural motifs - interior loops, hairpin loops, multi-loops and dangling ends. The figure 2 below illustrates this conversion. This format is called the *dot-bracket* notation and it is the most simple and compressed way of writing down the secondary structure of RNA. Extended formats of this notation also exist that denote each structural element type using a unique symbol and can include tertiary structural elements such as pseudoknots.

This simple string-based structure representation using dots and brackets is suitable for various computational methods and is therefore widely used in key problems concerning the structure of RNA. One of these problems is the secondary structure prediction or

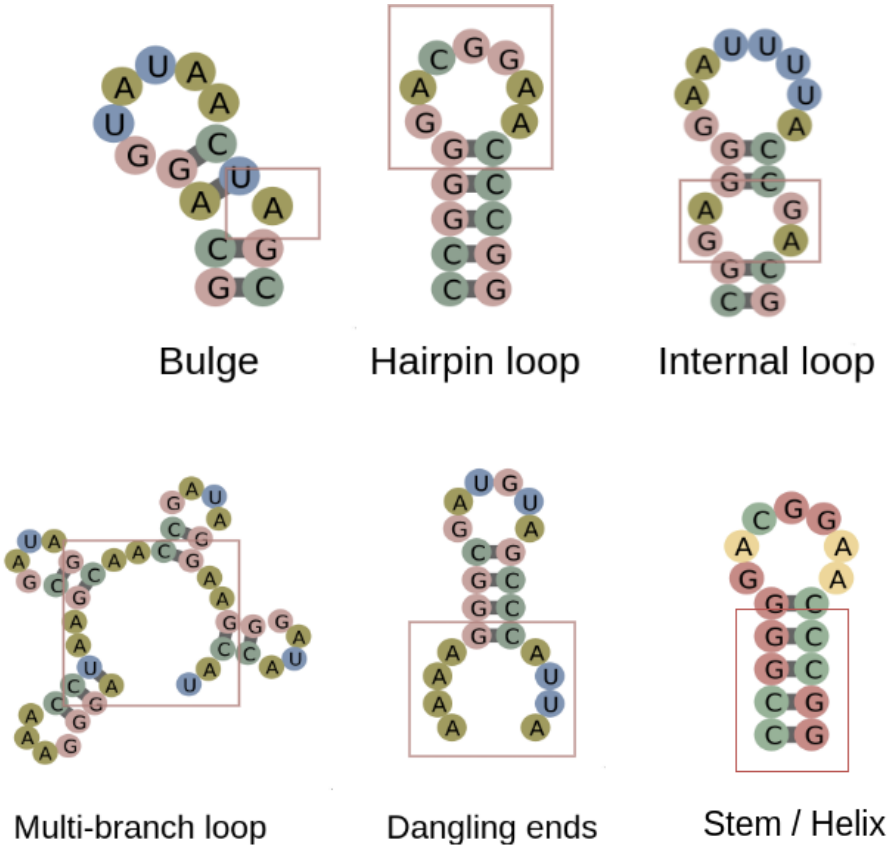

Figure 1: *The most common RNA secondary structure motifs.*

Figure 2: *Illustration of converting a RNA secondary structure into dot-bracket notation. Paired nucleotides are denoted as opening and closing brackets, while unpaired nucleotides are represented as dots.*

RNA folding problem which tries to predict the shape of an RNA molecule based on its nucleotide sequence. The main focus of this thesis project is an reverted variant of this problem, called RNA inverse folding.

1.2 RNA Inverse folding

RNA inverse folding *(RNAIF)* is concerned with correctly predicting a nucleotide sequence that will generate a desired target secondary structure when folded into its minimal free energy *(MFE)* structure.

Most RNA inverse folding algorithms share the same processing steps in order to generate nucleotide sequences for the target secondary structures:

1. Extract the features relevant for your model from the target secondary RNA structure.
2. Generate a prediction of the string of nucleotides that will fold into the target shape using the extracted features.
3. Find the minimum free energy *(MFE)* structure of the generated nucleotide sequence by using an RNA secondary structure prediction algorithm such as Zuker's algorithm [3].
4. Compare the resulting structure in dot-bracket format to the target structure using distance measures.
5. If the structures do not match, optimize the model based on its objective function. If they do match the problem is considered to be solved.

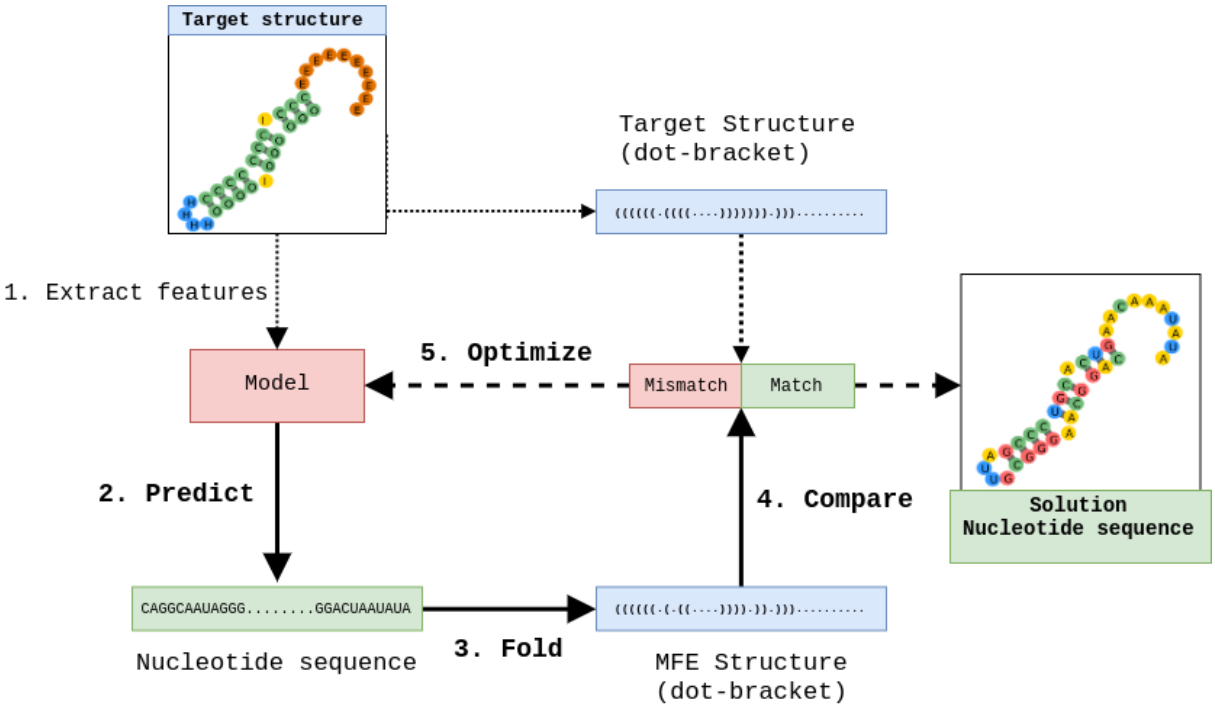

Figure 3: *The logic of RNA inverse folding illustrated in 5 main steps.*

Since there are four nucleotides that comprise RNA molecules as the length $|l|$ of the molecule increases, the number of possible sequence arrangements grows exponentially as 4^l. The search space is reduced when considering the fact that nucleotides that create base pairs can only have six different types of couplings: A-U, G-C, G-U and their inverted variants. With this constraint in mind, the number of possible sequences that any secondary structure can have when p is the number of nucleotide pairs and u is the number

of unpaired nucleotides can be expressed with the following equation:

$$N_{sequences} = 6^{p/2}4^{u} \tag{1}$$

Applications and significance

Since RNA is such a multipurpose molecule that can interact with a wide range of other ligands, a lot of potential for nano scale engineering applications are possible using RNA as a substrate. Here is a list of applications that have utilized the predictions of RNA inverse folding for various purposes:

- **Search of alternative natural RNAs** - RNA inverse folding can be used a precursor for non-coding RNA (ncRNA) sequence search within genomes. Since the function of ncRNAs is largely determined by their structure inverse RNA folding algorithms provide a solution for generating many alternative sequences that can then be identified within genome databases [4]. RNA inverse folding has also been applied to the search of other naturally occuring RNAs, such as ribozymes [5].

- **Riboswitch design** - many bistable RNA nucleotide sequences exist that can change their structural conformation based on a stimulus from its surroundings - light, temperature or ligand binding. Using RNA inverse folding approach, functional hammerhead ribozymes sequences have been successfully generated purely *in silico* that were then used as functional signalling molecules. [6] Publication by Win et al. [7] showed applied inverse RNA folding for designing molecular logic gates out of several RNA components involving multiple ligand binding.

- Nanomaterial design - multiple publications have shown the applicability of RNA as a material for modular structure design at nano scales. Various modular polyhedral structures have been succesfully generated using predictions of RNAIF algorithms [8].

Since the functional properties of RNA molecules are so reliant on its structure, being able to generate nucleotide sequences that fold into specific shapes is likely to gain more importance in the future as more roles that RNA plays within living systems are discovered.

2 Related Work

There have been many implementations of RNA inverse folding algorithms over the years which utilize a diverse range of computational strategies to generate nucleotide sequence solutions for target structures. These methods can be classified into local and global search methods. The majority of algorithms start with a random nucleotide string and gradually permute one or more nucleotides at a time and by repeatedly recomputing the resulting RNA secondary structure while being guided by their objective functions. Some of the methods try to optimize the matches between the target and folded nucleotide sequence in small local windows before reassembling them into a full sequence [9]. Other methods rely on a more globally oriented approach and work on multiple nucleotide sequences at once between the target and predicted structure. [10]

A survey by Churkin et al. (2017)[9] identified 20 algorithms that have been used to solve RNA inverse folding problem. Since then, at least four more algorithms have been designed [11, 12, 13, 14] that have improved upon many of existing algorithms by using neural network based design. Among other machine learning methods Reinforcement Learning *(RL)* appears to be an applicable approach for various problems in biology. Since a lot of RNA or DNA sequence-related problems are combinatorial in their nature, utilizing the ability of an algorithm that can be trained for an extended periods of time has produced successful results. RL has already been used for multiple sequence alignment [15], outperforming well established methods such as ClustalW in multiple scenarios. In the domain of RNA inverse folding, a reinforcement learning-based method *LEARNA*[12] has produced state-of-the-art results both in terms of speed as well as design accuracy. Reinforcement learning has also been successfully applied to other molecule design problems such as de novo chemical compound design [16] and protein folding problem. [17]

2.1 Vienna RNA Package

ViennaRNA package is a library written in C that contains various algorithms and programs relating to RNA secondary structure. This package includes *RNAfold* [18] which uses Zuker's dynamic programming algorithm to predict RNA secondary structures. ViennaRNA also provides efficient *SWIG* bindings for Python, therefore it can be used within our model training pipeline directly without needing to generate command line calls to its interface.

It also includes the first algorithm designed for RNA inverse folding - *RNAinverse (1994)* [19] which is going to be covered in more detail in the following section.

ViennaRNA is used in many various other RNA structure-related programs due to its *multi-core* implementations of the main RNA folding, alignment, pseudoknot prediction and sequence analysis algorithms as well as an accessible Application Programming Interface *(API)*.

RNAfold has been widely used by a majority of RNA inverse folding methods as a means to compare the secondary structure of the target RNA and the structure of the generated nucleotide solution. It also allows various configuration options such as setting the thermodynamic energy parameters or temperature, allowing or disallowing *wobble* (G-U) nucleotide pairs within the folded structure and various parameters regarding the free energies of the respective structural motifs.

2.2 Other RNA Inverse folding methods

In this section we are going to cover several inverse RNA folding algorithms. All six of these algorithms were used to evaluate the performance of our model in the later sections. Even though many more *RNAIF* algorithms are available, only this subset of algorithms could be reliably run and tested.

These methods utilize different computational methods in order to generate nuleotide sequences, therefore each has its strenghts and weaknesses. Some of these algorithms offer a lot of options for setting various nucleotide sequence and structure constraints while others are able to take pseudoknotted RNA structures as input. Other algorithms are better suited for rapid sequence design or providing solutions to more complex or longer target structures. Below we briefly summarize each of these methods:

- ***RNAinverse*** - Hofacker et al. intoduced the problem of RNA inverse folding in their paper in 1994, introducing the first algorithm for solving it - RNAinverse. [20] This algorithm uses adaptive random walks to minimize base pair distances between the MFE structure (or alternatively the ensemble centroid) of the predicted nucleotide sequence and the target secondary RNA structure. The algorithm works in small local windows that try optimizing RNA stuctural motifs first and then assembling them into the final solution. It is available both as a part of the *ViennaRNA* package [21] and as a webserver.

- ***MODENA*** - by Taneda et al. [10][22] is a method based on multi-objective evolutionary algorithms. This algorithm can generate solutions for bistable RNA structures that can change shape based on variables such as temperature or concentration of various compounds within the surroundings of an RNA molecule. It is also able of handling pseudoknotted structures as input as well as various sequence constraints [23].

  A dataset from the publication of this algorithm is going to be used as one of the benchmarks for the comparison of RNA inverse folding algorithms as it has been used in several other papers in the past [11, 12]. The source code and documentation of this package can be found here [23].

- **antaRNA** - this method is based on an ant colony optimization algorithm. [24] It has various customization options such as the usage of pseudoknotted structures, specification of sequence constraints, and target GC content. This model also can be used as a webservice and its source code is available in the following repository [25].

- **NUPACK** - [26] uses ensemble defect minimization as it's main objective to design the nucleotide sequences for pseudoknot-free target RNA structures. Optionally the user can specify sequence motifs that cannot be used within the solution sequences. It is one of the few methods that does not rely on third party algorithms for validating predictions and uses its own implementation of the Zuker [3] RNA structure prediction algorithm. It also has an option to use various sets of energy parameters and allows designing several designs in parallel. Source code of this method can be found on the following website. [27]

- **LEARNA** - by Runge et al. (2018)[12] is an RNA inverse folding method based on reinforcement learning. The model sequentially generates the nucleotide sequence, by converting the target secondary RNA structure (dot-bracket notation) into a binary format (0 for unpaired nucleotides, and 1 for base pairs). Models trained by reinforcement learning that this simple target representation produced the best results to date on several benchmarks. The authors also used a sophisticated hyper-parameter optimization pipeline that has not been used in a similar fashion before in the context of reinforcement learning. The authors also provided the data that they used for training their learning models and have also conducted an in depth analysis and comparison between various RNA inverse folding methods.

  This algorithm was the first RNA inverse folding algorithm based neural networks and a lot of ideas on which the *RLIF* model is based on were inspired by this publication. [28]

- **RNA-MCTS** - [11] uses Monte-Carlo tree search as its main computational method. It performs a guided tree search which allows it to regulate the GC content within its nucleotide sequence designs. This model has great efficiency and low computational overhead and an ability to backtrack through the search tree, once undesirable parts of the search space are reached, it is faster than most other RNA inverse folding methods. The source code and documentation of this algorithm can be found here. [29]

Method	Based on	Folding algorithm
RNAInverse (1994)	Adaptive walks	RNAfold
NUPACK (2011)	Ensemble defect minimization	Zuker
MODENA (2015)	Evolutionary algorithm	RNAfold
antaRNA (2015)	Ant colony optimization	RNAfold
RNA-MCTS (2017)	Monte-Carlo tree search	RNAfold
LEARNA (2018)	Reinforcement learning	RNAfold

Table 1: *Summary of the RNA inverse folding methods that are going to be used for comparison. All of them use Vienna RNAfold as the means of confirming their predictions, except for NUPACK which uses it's own implementation of the Zuker algorithm.*

Having introduced the RNA inverse folding problem, its applicability and existing solutions, in the next sections we will describe the details of the implementation of *RLIF* algorithm.

3 Implementation

This section will cover the details of how RNA inverse folding problem was adapted to be used with reinforcement learning. The neural network architectures and the datasets used to train, test and compare the method will be described in detail along with the implementation of user intefaces.

Our goal is: generate a nucleotide sequence whose minimum free energy secondary structure matches the target RNA structure in dot-bracket format. If we want to apply reinforcement learning to this problem we need a way to represent our sequence - we need a representation of its current state along with actions with which the agent can interact with the solution. The following sections will describe the design choices of this formulation. We want to generate the nucleotide sequence sequentially by observing the sequence generated by the model so far along with our target secondary structure.

3.1 Reinforcement learning

Markov Decision Processes

Markov Decision Process (MDP) is a Markov Chain that describes a semi-stochastic transitions between pairs of environment states $(s \rightarrow s') \in S$. This concept is at the core of all reinforcement learning algorithms and it enables the formalization of various different kinds of goals and tasks that involve decision making and utility maximization.

We will briefly describe the terminology of reinforcement learning by relating it to the RNA inverse folding problem. In the later sections, each of these components as well as their implementations will be described in detail.

Agent: the learning and acting part of the model that can perform actions in order to change the state of the environment and learn based on the reward it receives.

Environment: is everything that is not the agent itself. Everytime the state of the environment changes the agent receives a reward. In the case of RNA inverse folding the nucleotide sequence that is being generated is the part of the environment that the agent can directly modify through actions.

State $s_n \in S$: is the descriptor of the current configuration of the environment which includes all the relevant attributes for the problem that is being modelled. In our case we want our state to be representative of the target RNA secondary structure and the nucleotide sequence generated so far.

Actions $a_n \in A$: is a finite set of distinct methods that the agent can use to interact with the environment. In this case the agent can perform an action to insert a nucleotide or a base pair into the sequence that is being generated.

Reward function R: maps particular environment states s' to scalar values that indicate how good they are in terms of achieving the intended goal. For RNA inverse folding the reward corresponds to how close the fully generated nucleotide sequence is to the target when folded into its minimal free energy (MFE) structure.

Discount factor γ: regulates the extent to which the agent prioritizes immediate rewards versus long term rewards. By backtracking through the history of the rewards

that the agent has received so far we can apply the discount factor to obtain *Return* G_t from timestep t.

$$G_t = \gamma R_{t+1} + \gamma^2 R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

The return G_t is a more accurate estimate of how good particular actions are when compared to just a single reward as it accounts for all the future rewards as well. In the case of MDP for RNA inverse folding, the discount factor is set to one, therefore it is an undiscounted Markov Decision Process. This is done because the agent receives the reward only once - when the full nucleotide sequence is generated and its structure is compared to the target, therefore each action that led to that solution is retroactively assigned an identical reward.

Policy π: The policy function describes a probabilistic mapping of environment states that the agent observes into actions. In the case of RNA inverse folding, π describes the probabilities of inserting a certain nucleotide or a base pair into the solution sequence given a particular state of the environment.

$$\pi(a|s) = P[A_t = a|S_t = s] \tag{3}$$

Policy function is the main function that defines our agent and determines what kind of nucleotide sequence solutions get generated. It is parameterized using neural networks and its training procedure is covered in detail in later sections.

Value function $V(s)$: the value function is the part of the reinforcement learning model that outputs the expected return G that is going to be obtained given a particular state s.

$$V(s) = \mathbb{E}[G_t|S_t = s] \tag{4}$$

Value function V is very important in all reinforcement learning algorithms as it measures how well a model can understand the relevant features of the problem that it is trying to solve. Like the policy function π, value function V is also approximated using a neural network, training it in a supervised manner - use each state s_t in a batch as an input for the neural network and the the return G_t obtained from that state as an target label and use gradient descent to minimize the loss between the output of the network $V(s_t)$:

$$L_{Vt} = (G_t - V(s_t))^2 \tag{5}$$

Episode In the case of RNA inverse folding, an episode is defined as all the $\langle S_t, A_t, R_t \rangle$ tuples obtained for each time step t while sequentially generating a nucleotide sequence from *5'* to the terminal *3'* nucleotide. The agents in reinforcement learning are usually trained on an episodic basis - state, action and reward tuples obtained during the whole episode are aggregated into batches that are then used to train the models.

These concepts will be used frequently in the upcoming sections along with the details of implementation of each of these components within the Markov Decision Process for inverse RNA folding. Additional terms will be described in section 3.6 that covers the learning algorithm and the training of policy neural network π_θ.

3.2 State Representation

To represent the state of our target structure and the current design we need to encode both the structural information as well as what nucleotides have been generated so far. In order to make informed decisions/actions the RL agent has to have access to all the relevant local information of the RNA structure.

First we need to find an efficient way of numerically representing the target structure. Therefore we need to convert our dot-bracket annotated structure into a vectorized numerical form. As mentioned in the introduction, there are six main distinct structural motifs that we care about in the secondary RNA structure: stems/helices that consist of opening O $_{(1)}$ and closing C$_{(2)}$ pairs, internal loops I$_{(3)}$, hairpin loops H$_{(4)}$, multibranch loops M$_{(5)}$ and dangling ends E$_{(6)}$. After decomposing the dot-bracket structure into these particular structural elements we can assign each element e to a unique index by using a mapping:

$$M(e) = \{O, C, I, H, M, E\} := \{1, 2, 3, 4, 5, 6\} \tag{6}$$

A dot-bracket secondary structure can then be represented by a binary 6 x L matrix, where L is the length of a target and each row corresponds to one of the six unique structural elements.

The nucleotide sequence generated so far is also encoded in a similar fashion - the four nucleotides are first encoded numerically and turned into a one-hot binary matrix of size 6 x L.

These two representations of our current solution S are then concatenated into solution encoding $\mathbb{E}_S$, where each column indicates the structural element and the nucleotide at index i of the current solution.

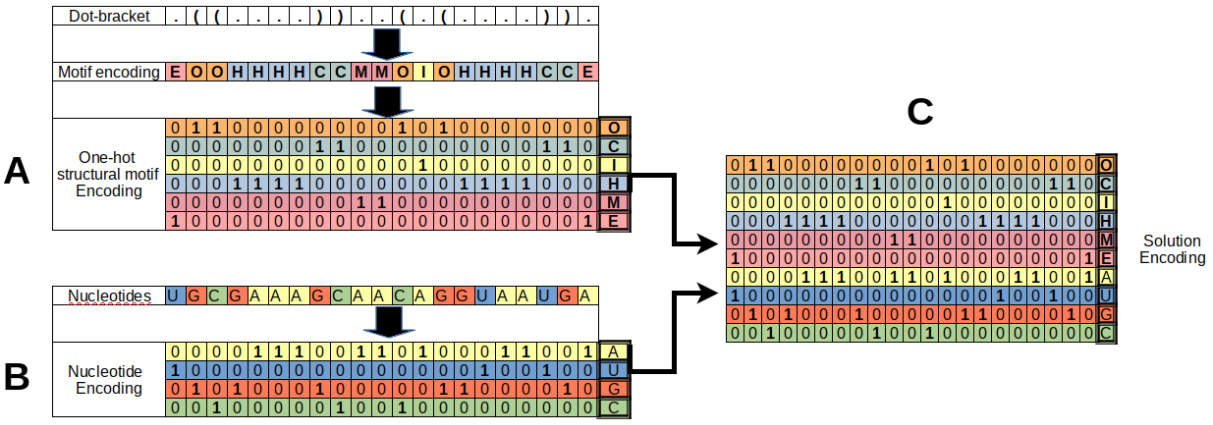

Figure 4: *The state representation of the current solution. An encoding of secondary structure (A) and the encoding of current nucleotide sequence (B) are concatenated to generate a combined representation (C) that captures the local features of the sequence.*

Since we want to be able to handle target sequences of any length, we need to constrain the number of columns of encoding $\mathbb{E}_S$ that are used as a state representation at any given point. Therefore a window size k is used as an extra parameter that regulates the size of the state input. Therefore the environment state at a given for index i is represented as:

$$s_i = \mathbb{E}_S[\text{i-k} : \text{i+k}] \tag{7}$$

The size of the window k acts as means of regulating the importance of local versus global structure of the current solution.

3.3 Action Space

After observing the current state of the environment/solution the *RL* agent then chooses an action based on its current policy function π. At any point the agent can insert one of the 4 nucleotides into the current solution $\mathbb{S}_T$. The set of available actions depends on the structural element e_i at current index i of the target sequence $\mathbb{T}$ (either a dot or a bracket). If the current structural element is a base pair (inside a stem/helix), a nucleotide is inserted both at the current index i along with a complementary base at index j. Therefore the action space of the MDP can be described as:

$$\mathbb{A} = \begin{cases} a_i \in \{A_i,\ U_i,\ G_i,\ C_i\}, & \textbf{\textit{if}}\ \ e_i = \textbf{\textit{"."}}\ \ \text{(dot)} \\ a_{ij} \in \{A_iU_j,\ U_iA_j,\ U_iG_j,\ G_iU_j,\ G_iC_j,\ C_iG_j\}, & \textbf{\textit{otherwise}} \end{cases} \tag{8}$$

After an action gets chosen from either of these sets, we can represent the nucleotide(s) in a numerical form by mapping them as $\boldsymbol{M(a)} = \{A, U, G, C\} := \{1, 2, 3, 4\}$. Relating back to our solution encoding $\mathbb{E}_S$, the nucleotide that is chosen according to the policy π_θ gets inserted into our binary solution encoding at the row corresponds to $n_1 = M(a_i)$ and column i that corresponds to the current index or timestep. In case the action contains two nucleotides, the encoding $\mathbb{E}_S$ is also modified at row $n_2 = M(a_j)$, and base pair index j, forming a complementary base pair.

$$\begin{aligned} &n_1, n_2 = M(a_{ij}) \\ &\mathbb{E}_S[n_1, i] = 1 \\ &\mathbb{E}_S[n_2, j] = 1, \quad \textbf{if } \exists n_2 \end{aligned} \tag{9}$$

3.4 Rewards and optimization

Reward functions play an essential role in reinforcement learning. A well formulated reward function is the main factor that enables efficient learning and the ability to produce actions that generate better solutions for a specific problem by gathering more experience from the environment.

For the inverse RNA folding problem our objective is to generate a nucleotide sequence whose MFE secondary structure matches the target structure. After we generate a full string of nucleotides, we then use an RNA secondary structure prediction algorithm, to obtain its secondary structure in dot-bracket format. We can then compare it to the target dot-bracket sequence by using various string comparison measures. One of them is Hamming distance, which simply counts the number of mismatches between the same indices of two strings of the same length.

Given our target structure $\mathbb{T}$ and structure $\mathbb{F}$ resulting from the folded nucleotide sequence that was generated by our model we can calculate the Hamming distance between them by:

$$H(T, F) = \sum_{i=0}^{n-1} T_i \neq F_i \tag{10}$$

Using Hamming distance between the target dot-bracket sequence $\mathbb{T}$ and our solution D as our main objective to minimize, we then normalize it by the length of the target $|\mathbb{T}|$ and scale it using an exponentiation factor ϵ:

$$R_S = (1 - \frac{H_{T,D}}{|T|})^{\epsilon} \tag{11}$$

The exponentiation factor ϵ ensures that only solutions with Hamming distance close to 0 get high rewards, while others remain relatively close to zero.

The reward is obtained only at the end of each episode - after the full sequence has been generated and the distances between the target and the designed sequence have been obtained. There is no real way of estimating which actions were the main contributors to the correct/incorrect folding, therefore we simply assign the same reward to each individual action. This is done by using a discount factor $\gamma = 1$ meaning that our formulation is an undiscounted Markov decision process.

This reward formulation has been used in the publication by Runge et al. [12], therefore it can be used reliably for training well performing models. Other more RNA-related distance measures were tried out during the research project, such as moutain distance and base pair distance [30]. However, Hamming distance alone seemed to be sufficient to produce efficient learning without overcomplicating the model. By using more distance measures we introduce additional tunable hyperparameters into the model as each of these distance measures has to be scaled by their corresponding coefficients if they are to be used within the same reward function. During previous research projects involving reinforcement learning we have found that keeping the reward functions as simple as possible makes model training a lot more and reliable and efficient.

However, other methods that are not directly related to distance measures or directly modifying the reward function were tried out and have proven to be very useful for training the models.

Reward ranking

A vast majority of recent breakthrough applications of reinforcement learning involve two-player games (chess, Go, strategy games). A huge advantage of training agents in two-player scenarios is the fact that we can find better policies by making the agent face the previous iterations of itself [31]. This way we can clearly tell which policy is best one and by repeating this process iteratively we can generate better agents without explicitly specifying the reward function. RNA inverse folding cannot directly be reformulated into a two player game, however we can approximate this process by enabling reward ranking [32]. The solution for a target structure $\mathbb{T}$ gets non-zero reward only if its reward belongs in the *90th* percentile of the rewards obtained so far when trying to solve that target. Therefore the full reward logic of our RNA inverse folding formulation is:

$$\mathbb{R}_S = \begin{cases} 2, & \text{if } R_S = 1 \quad (\text{when } \mathrm{H}(\mathbb{T}, D) = 0) \\ 1, & \text{if } R_S > \overline{R}_{T_{90\%}} \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

One of the advantages of this formulation is that it enables the use of diverse target RNA structures among the same training dataset. Each target secondary structure has

different difficulty level, associated with the number of unstable motifs it contains, making other alternative conformations more energetically favorable. The alternative conformations can also possibly be very distinct from the target structure as well, therefore the reward landscape for such difficult sequences is relatively discontinuous - several changes in the nucleotide sequence can produce a completely different secondary structure. Therefore even though in actuality the nucleotide sequence of the solution was relatively close to producing the correct structure, the resulting reward indicates otherwise as the Hamming distance between the target and the alternative conformation is very large. Therefore sequence in the training dataset obtains diverse range of rewards while using the same policy π based on their difficulty. What we really care about is not the maximization of the overall reward but whether the model becomes better at solving all targets. To overcome this we keep track of the rewards received for each target sequence $\mathbb{T}$ and only scoring a solution positively with a fixed reward of $\mathbb{R}_S = 1$ if it exceeds at least 90% of the solutions generated for that specific target so far. It gets double the reward if it generates a valid solution. By doing this we make sure that the model prioritizes a policy that generalizes over all targets in the training data and their difficulties are therefore normalized.

This reward ranking formulation has proved to be useful and improved the training speed as well as the overall performance of the models.

3.5 Neural Networks and function approximation

To approximate the policy and value functions various neural network architectures were tried out. The architectures that produced the best results had the following structure - the input layer consists of our state representation (the solution encoding $\mathbb{E}_S$), followed by feature extraction layers - using a combination of 2D and 1D convolutional layers and fully connected layers. The last part of the networks splits into two separate heads - one for value function V and the other one for policy function π and this architecture is called the *Actor-Critic* network. To reiterate, the output of the value head V is the estimate of the discounted future reward that the network approximates it is going to obtain, given a certain state s. The output of our policy π head consists of 6 nodes that correspond the action space described in section 3.3.

Convolutional Neural Networks

Convolutional neural networks were successfully used to process the state representation of the nucleotide sequence. We can treat our state representation $\mathbb{E}_S$ as a single channel image that is filtered by applying learned convolutions. This process continues for multiple stages, and the multiple filterings produce a different kind of state representation that then gets fed into the fully connected layers that follow the convolutional feature extraction part. The best neural architectures are going to be covered in more detail in the *Methods* section below.

LSTM

Long short term memory *(LSTM)* networks are a type of recurrent neural networks that can be successfully applied to process various time-series data. It has also been successfully used in reinforcement learning and since the task of RNA inverse folding

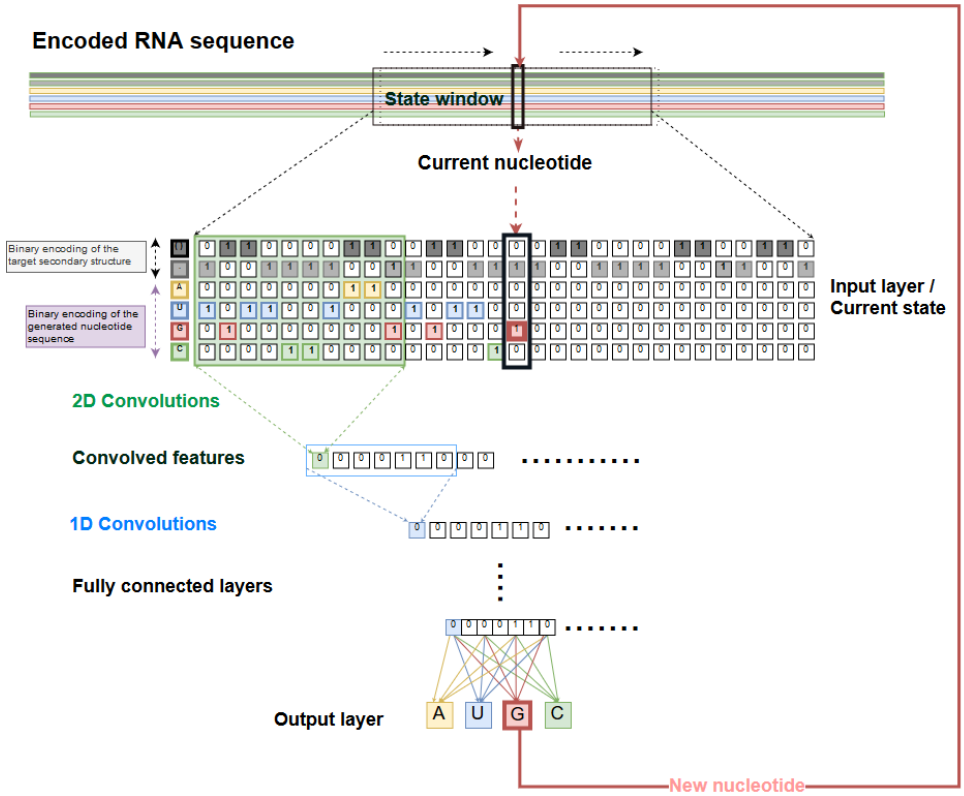

Figure 5: *Depiction of the neural network architecture used in RLIF. The encoding of the solution $\mathbb{E}_S$ bounded by the state window k is used as input to the convolutional neural network. 2D convolutions are followed by 1D convolutions which are then in turn fed into a set of fully connected layers. The output layer is the function policy π that parametrizes the probability distribution over actions for a given state and determines which nucleotide or nucleotide pair gets inserted into the solution. After an insertion the state window moves to the next unfilled nucleotide of the solution and the process is repeated until a full sequence is generated.*

involves a long sequence of actions, making use of the capabilities of LSTMs appears to be reasonable. Since the state window of our RNA sequence is of limited size (generally encompassing around 40-100 nucleotides) our network is only aware of the local and not global structure. Keeping track of the nucleotides that were generated previously expands this window and should provide more context for generating better actions. Therefore during this project we tried combining the LSTM networks with all of the architectures above but generally it failed to produce any considerable improvements. In addition the processing time increases drastically when incorporating LSTMs into the the architectures, therefore there were not enough incentives to warrant the usage of LSTM architecture in the final models.

3.6 Algorithm

Having described all of the elements of the Markov Decision Process formulation for the RNA inverse folding problem and the neural networks that are used to approximate the functions, we continue by describing how they are used within the context of reinforcement learning.

First, we combine all the sections above to formalize the algorithm of *RLIF* which takes a target sequence $\mathbb{T}$ in dot-bracket notation as input and returns a nucleotide sequence string $\mathbb{S}_{\mathbb{T}}$.

Algorithm 1: RLIF(target structure $\mathbb{T}$)

Result: Nucleotide string $S_{\mathbb{T}}$

Initialize;
Target structure in dot-bracket notation $\mathbb{T}$;
Sequence length $l = |\mathbb{T}|$;
Solution encoding matrix $\mathbb{E}_S = [10 \times l]$ (described in figure 4);
Empty nucleotide string $S_{\mathbb{T}}$ of length l;
Action-nucleotide mapping $M(a) = \{1, 2, 3, 4\} := \{A, U, G, C\}$;
Policy function π;
State window size k;
Current nucleotide index i $\leftarrow 0$;
while $i < l$ **do**
    State $s_i = \mathbb{E}_S[i - k : i + k]$;
    Action $a_{ij} = \pi(s_i), \quad a_i, a_j \in \{1, 2, 3, 4\}$;
    Modify encoding $\mathbb{E}_S$ **and string** $S_{\mathbb{T}}$ **based on action** a_{ij};
    $\mathbb{E}_S[a_i, i] = 1$;
    $S_{\mathbb{T}}[i] = M(a_i)$;
    if $\mathbb{T}_i \neq$ "." **then**
        j $\leftarrow$ index of complementary pair of $\mathbb{T}_i$;
        $\mathbb{E}_S[a_j, j] = 1$;
        $S_{\mathbb{T}}[j] = M(a_j)$;
    i $\leftarrow$ go to next unfilled nucleotide index in $S_{\mathbb{T}}$;
end;
Return $S_{\mathbb{T}}$;

In order to train the policy function that is used in the *RLIF* algorithm, we use Proximal Policy Optimization (PPO)[33] reinforcement learning algorithm. It is one of the most widely used algorithms due to its training stability, ability to handle a very diverse range of problems and action/state spaces as well as compatibility with all neural network architectures. PPO belongs to the class of policy gradient methods within the classification of *RL* methods which directly compute the gradient for the policy function parametrers (in our case the weights of the neural network) by using the concept of *Advantage* $A_t(a_i, s_i)$ and Action-Value function $Q(a_i, s_i)$. We will cover these concepts in a similar manner as in section 3.1.

Action-Value function (or Q-Function) $Q(a, s)$**:** action-value function $Q(a, s)$ is similar to the value function $V(s)$ and in addition to the state s it also takes the action a as input and estimates the expected return that would result from taking a particular action given a particular environment state.

$$Q(a, s) = \mathbb{E}[G_t | A_t = a, S_t = s] \tag{13}$$

The Q function underlies one of the most known RL algorithms - Q-Learning.[34] The main idea behind this algorithm is that by sampling a diverse range of state and action pairs by interacting with the environment using random actions (exploring) and receiving rewards, we can update the Q-Function to give a more and more accurate prediction of the future reward. Therefore as the training progresses we can start relying upon this Q-function to choose actions instead of sampling randomly. By repeating this process of gathering experiences and gradually decreasing the probability of choosing random actions the model converges to a policy that maximizes the overall expected return. Assuming that the reward function is well formulated this often results in a policy that is achieves the intended task.

In the context of Proximal Policy Optimization, the Q-function is used to obtain the advantage function A_t which combines the Value function $V(s)$ and Action-Value function $Q(a, s)$.

Advantage $A(a, s)$**:** measures how good a particular action a is given a certain state s. It is obtained by comparing the output of Value and Action-Value functions.

$$A(a, s) = Q(a, s) - V(s) \tag{14}$$

In very general terms, a positive $A(a, s)$ value indicates that a particular action a is better than the action that would be selected by the current policy π_θ given a state s in terms of expected return G (discounted future reward). The opposite holds for negative $A(a, s)$ values.

Therefore the general training pipeline for policy gradient methods is as as follows: run a Markov Decision Process for one episode for n timesteps and collect all the state, action and reward tuples $(\langle S_t, A_t, R_t \rangle_{1...n})$ for each timestep t. We can then calculate the return (discounted future reward) G_t for each timestep based on all the rewards in timesteps $\{t + 1....n\}$. We then obtain the advantage for each timestep t by using our model's estimates for $Q(a, s)$ and $V(s)$ functions and comparing it to the actual return G_t. Each action will therefore have either a positive or negative advantage value which indicates whether it should be more or less probable after the next network update. The advantage values are then used obtain the gradient for network parameters θ by differentiating the following loss function: [33]

$$L^{PG}(\theta) = \hat{\mathbb{E}} \left[log \pi_\theta(a_t | s_t) \hat{A}_t \right] \tag{15}$$

which would adjust the expectation of an action a_t given a state s_t under current policy π_θ by scaling its log probability based on the associated advantage value.

Since our Value function $V(s)$ is also parameterized by the same neural network (separate head) we can obtain additional gradient for the network updates using a secondary loss term:

$$L^V(\theta) = (G_t - V(s_t))^2 \tag{16}$$

which is the Mean Square Error between the actual return G_t and the prediction of the Value function $V(s)_t$. Our final loss function is then the sum of these two terms:

$$L^{PPO}(\theta) = \hat{\mathbb{E}}\left[log\pi_\theta(a_t|s_t)\hat{A}_t\right] + (G_t - V(s_t))^2 \tag{17}$$

Using the terms described above we can now formalize the learning algorithm that was used to train the *RLIF* policy π_θ.

Algorithm 2: PPO Training algorithm

Load dataset of target sequences D;
Initialize policy neural network π_θ;
while *time* $<$ *limit* **do**
    for *episode* $E = 1$ *to* N **do**
        Select random sequence $\mathbb{T}$ of length $l = |T|$ from dataset D;
        Nucleotide string $\mathbb{S}_T = \text{RLIF}(\mathbb{T})$, using policy π_θ;
        MFE structure $D_{\mathbb{S}} = \text{RNAfold}(\mathbb{S})$ (Lorenz et al. [18]);
        Episode reward $R_E = \mathbb{R}_S(\mathbb{T}, D_{\mathbb{S}})$ based on equation 12;
        Compute advantage estimates $A_1 ... A_l$ based on state, action, reward tuples ($\langle s_i, a_i, R_i \rangle_{1...l}$) generated during an episode of *RLIF*;
    Optimize policy parameters θ based on loss L (equation 17) with gradient descent;

In this section the final iteration of the this reinforcement learning model for RNA inverse folding was covered. In the following section we cover the software and hardware setup used for training the models and the implementation of user interfaces that extend the functionality of this model to be used for interactive RNA sequence design.

4 Methods

4.1 Software

The *RLIF* model was implemented in *Python* [35], utilizing Tensorflow [36] for designing the neural networks. The graphical interface was implemented using the *Python* bindings for the *Qt* [37] interface design library. The library used for rendering the RNA secondary structure within the graphical interface can be found in a github repository [38] by Michelle Wu.

Reinforcement learning

For the first iterations of the model a custom implementation of Proximal Policy Optimization (PPO [33]) was used that helped developing the proof of concept for of the RNA inverse folding environment. However having to tune the implementation of the learning algorithm along with the learning environment has proven to be difficult to debug. Therefore for the later stages of the project a switch to using a RL library *stable-baselines* [39] was made which simplified a lot of the development progress and allowed to focus specifically on optimizing the performance of the model. The functionality of this library was extended to include custom neural net architectures, environment configuration, scripts for training and testing saved models as well as running the benchmarks covered in the later sections.

4.2 Model training

During the course of developing the RLIF method, 84 different models with different configurations were trained. The majority of the models were trained on *LIACS Tritanium* cluster, a machine with a 20-core CPU and *8 NVIDIA K-80 GPUs*. The models were trained by running up to 40 workers threads in parallel, each having a separate instance of the RNA inverse folding environment and solving a random target RNA secondary structure. Each of the workers communicates with the global policy network π to obtain actions based on the current state of their environments. Training in this fashion allows the pooling of all worker experiences into a main buffer that is then used to train the main policy network. This increases the training speed considerably and combined with a high memory *GPU* allows the usage of larger batches for gradient descent updates that make the training process more stable.

The final model of *RLIF* is a combination of 4 models with different hyperparameters that run in parallel and produce slightly different sets of sequences for a given target RNA secondary structure. Combining several models has proved to be the best approach as it increases the search space that the model is capable of covering. We describe the hyperparameters of each of these models in table 2.

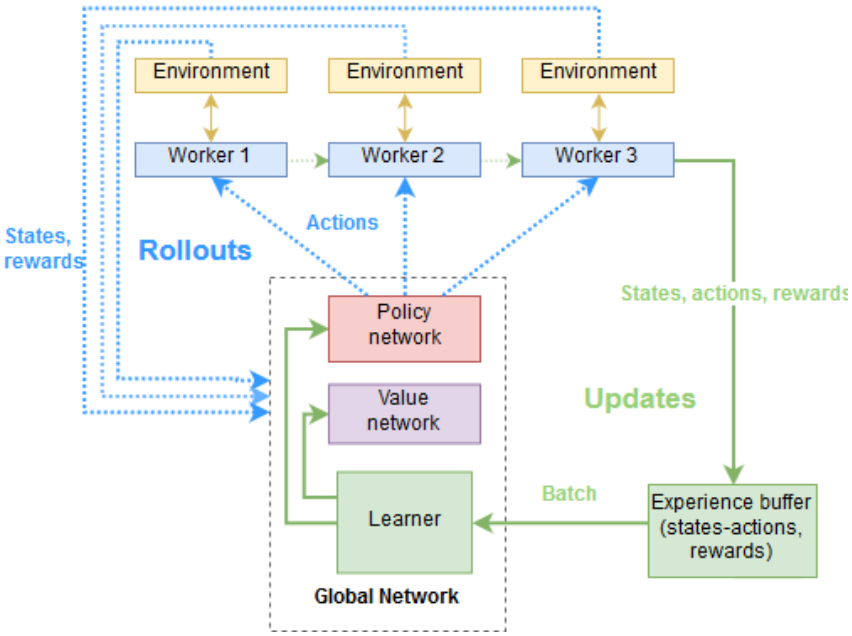

Figure 6: *Parallel training pipeline. Each worker has a separate instance of an RNA design environment and produces actions based on the global policy network π.*

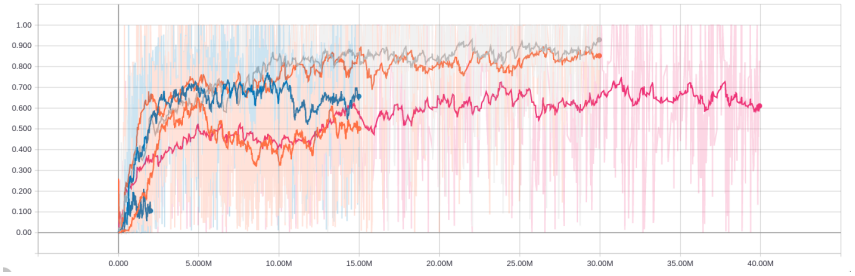

Figure 7: *Training curves of the 4 best performing models of RLIF. Y axis displays the percentage of solved sequences in a single prediction, X axis denotes the number of total nucleotides. Models were trained for 15M, 30M, 30M and 40M steps respectively.*

Parameter	M1	M2	M3	M4
Learning rate	9×10^{-5}	2×10^{-4}	5×10^{-5}	3×10^{-4}
Batch size	8192	4096	8192	4096
# Convolutional layers	4	4	4	3
Convolutional filters	32, 256, 128, 16	64, 256, 128, 16	32, 256, 128, 32	32, 256, 16
Kernel sizes	(10,3), (1,5), (1,5), (1,5)	(10,5), (1,10), (1,10), (1,3)	(10,1), (1,10), (1,10), (1,3)	(10,1), (1,10), (1,3)
Nodes / hidden layer	256, 128, 64	256, 256, 128	128, 64	128, 64
Nodes / hidden layer π	64	64	64	16

Table 2: *Neural network hyperparameter configurations of the 4 models used in RLIF.*

4.3 User Interfaces

One of the things noticed during the project was that most of the available programs for RNA inverse folding have interfaces that are hard to use or interpret. This issue was also raised in the survey by Churkin et al. (2018) that reviewed these programs.[9] Therefore designing a good user interface became a goal of this thesis project which also allowed to extend the functionality of the *RLIF* algorithm. Two different user interfaces were created: command line interface and a graphical user interface the details of which are explained below.

4.3.1 Command Line Interface

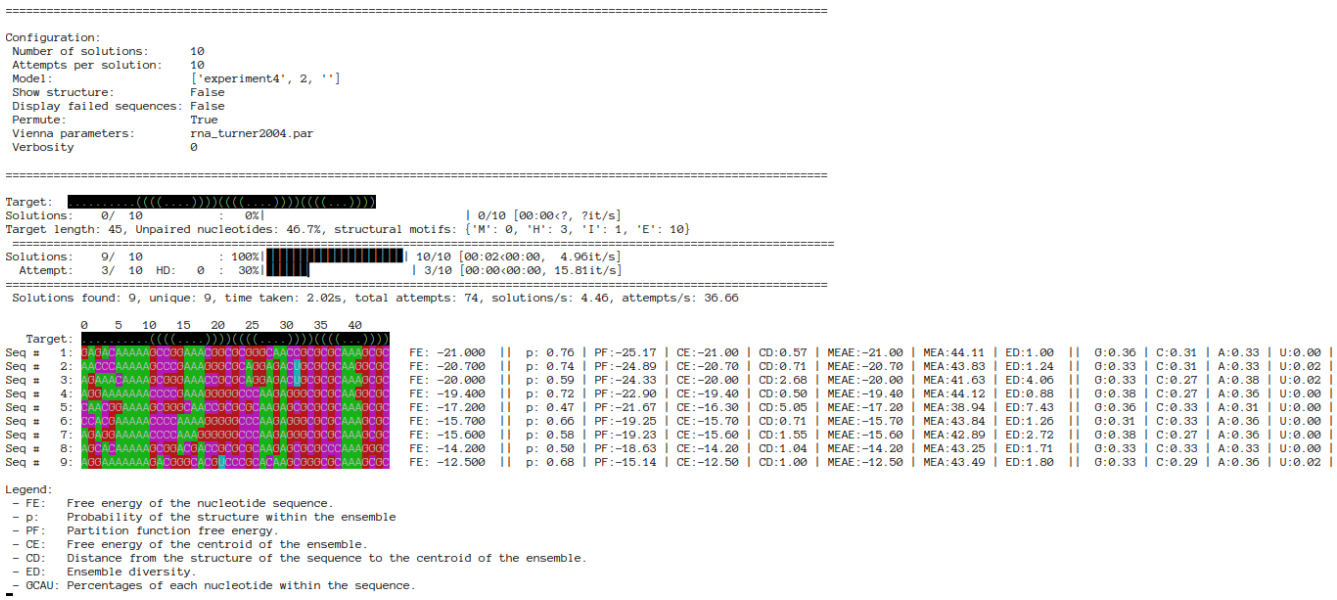

Figure 8: *Command line interface (CLI) of RLIF.*

The command line interface of *RLIF* can be used for calling the model to generate multiple solutions for a target RNA secondary structure. The number of result solutions and attempts can be specified along with the energy parameters used. After entering a target structure, the number of structural motifs it contains is displayed. The generated valid nucleotide sequence solutions are then sorted by their free energy and various statistics regarding the structure ensemble of that sequence are displayed. Optionally the failed solutions can also be displayed and they are sorted based on their Hamming and mountain distances.

4.3.2 Graphical Interface

The graphical user interface of *RLIF* contains various elements that enable the user to specify the desired RNA secondary structure and generate the valid nucleotide sequence solutions for it. ViennaRNA[18] Python bindings are used extensively in this program both for validating the MFE structure of each nucleotide sequence using *RNAfold* and other statistics relating to RNA structures. Therefore controls for configuring various parameters of *ViennaRNA* are included, allowing the user to inspect how the energy parameters for the folding algorithm or other parameters such as temperature influence the predicted shape of RNA secondary structures in real time. Most of the secondary RNA structures have multiple nucleotide sequences that fold into that shape, therefore the interface includes various graphical elements that separate each generated solution based on their characteristics.

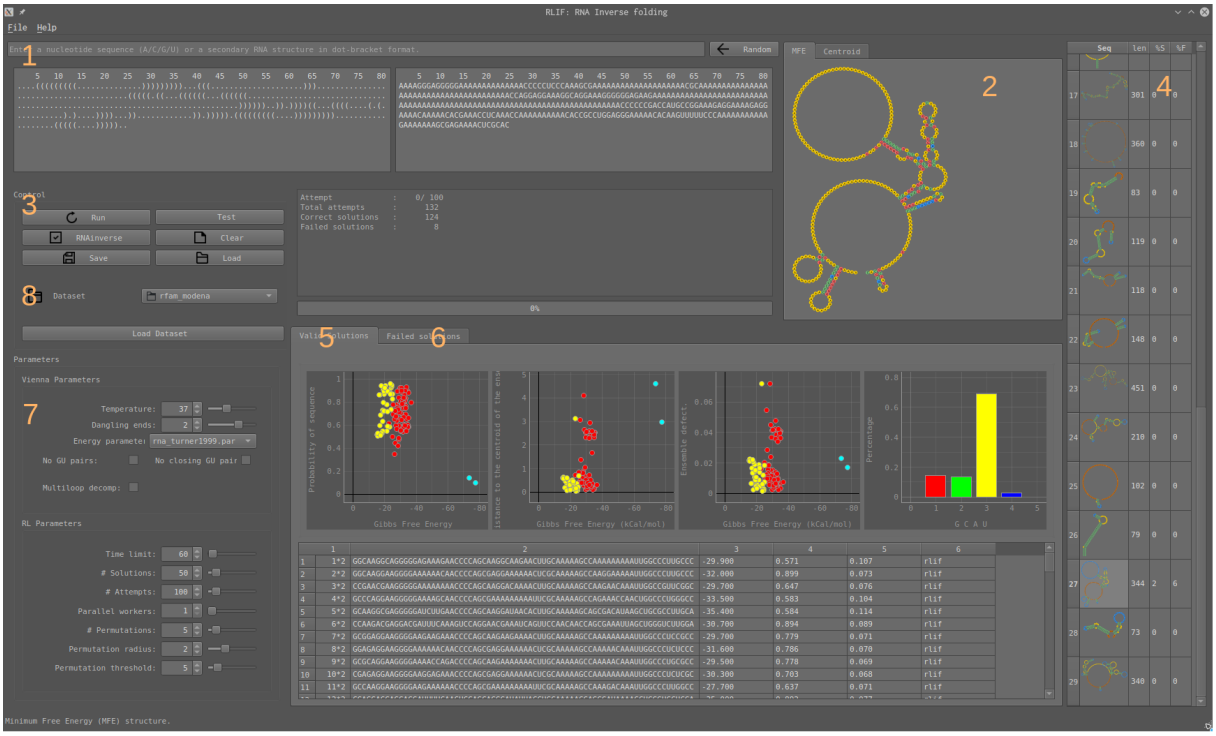

Figure 9: *Graphical user interface (GUI) of RLIF.*

Below the main interface elements and their uses are summarized:

1. ***Sequence input:*** the target sequence can either be a nucleotide sequence or a secondary RNA structure in dot-bracket format. The sequences can be inputted either one symbol at a time (hitting *Enter* will create an entry in the target selection table). A button for generating random secondary structures is located on the right hand side from the input box.

2. **Structure display:** is used to display either dot-bracket annotated RNA structures, or the MFE structure of a selected nucleotide sequence. Additional tab for displaying the structure of the ensemble centroid is available as well.

3. ***Button controls***: Contains buttons for:

- ***Run***: runs the RLIF algorithm to produce nucleotide sequences for the currently selected RNA secondary structure.
- ***Test***: runs RLIF in a sequential mode, where nucleotides are generated one at a time. The progress box shows the current state representation $\mathbb{E}_S$ that the neural network receives as an input in order to generate the next nucleotide.
- ***Save***: saves all the current failed and valid solutions into a log file that can be restored later.
- ***Load***: load RNA secondary structures from log files previously saved using this interface. *Fasta* or *.fa* file extensions are also supported for loading nucleotide sequences that can then be converted into target sequences for inverse RNA folding.
- ***Clear:*** clears all the current solutions and targets.
- ***RNAinverse:*** runs RNAinverse folding algorithm from *ViennaRNA* package, mostly implemented for comparison reasons. Not advised to be used with sequences longer than 100bp.

4. ***Target selection:*** all of the valid target structures loaded or generated from either source are shown in the target selection table on the right side of the interface.

5. ***Solution tab:*** contains plots for centroid distance, ensemble defect and the probability of the minimum free energy structure within the ensemble for each generated nucleotide sequence. The points on all of the plots can be individually selected to highlight the statistics of each solution. The table below also contains all the solutions along with their nucleotide sequences.

6. ***Failed solution tab:*** shows all nucleotide sequences whose minimum free energy (MFE) structure does not match the target secondary structure. The tab also has a visual display of the MFE secondary structure of each solution along with plots for hamming and mountain distances.

7. ***ViennaRNA controls:*** in this part of the interface the set of energy parameters can be chosen. The default settings for *ViennaRNA 2.0+* are the Turner et al. (2004)[40]. The alternative folding parameters include Turner et al. (1999) [41], Andronescu et al. 2007 [42] and Langdon et al. (2018) [43], and changing these parameters can substantially alter the MFE structures of nucleotide sequences. This control section also includes the temperature dial, and checkboxes for disabling wobble G-U pairs either completely or at the endings of stems.

8. ***Dataset selection:*** this part of the interface contains allows testing secondary RNA structures from three of the benchmark datasets that are covered in the following section.

This graphical user interface is suitable not only for RNA inverse folding but for other purposes as well. The *.fasta* file support makes it possible to analizing batches of RNA sequences as it provides various useful metrics as well as real-time editing and visualization of resulting RNA secondary structures.

5 Results

5.1 Comparison to existing methods

Six other RNA inverse folding methods that were covered in the related work section were compared alongside *RLIF*: *RNAinverse, NUPACK, MODENA, antaRNA, rnaMCTS* and *LEARNA*. The Python-based methods (*RNAinverse, antaRNA, rnaMCTS, LEARNA*) were integrated into the benchmark scripts directly while others *('NUPACK, MODENA')* were called by communicating to their command line interface binaries. The exact commands that were used to call them are listed in the appendix 6.1.

5.1.1 Benchmark performance

Several benchmarks for testing RNA have been developed and used in various publications in recent years. These datasets of secondary RNA structures contain sequences that are usually hard to solve because of the unusual structural motif combinations that they contain. These structures often have low energetic stability and have only a limited number of possible nucleotide sequences whose minimum free energy structure their structure.

All of the benchmarks were run on a computer with *Intel-i7 4700HQ* Quad-Core CPU and 16GB RAM. No GPU-support was used for the models utilizing neural networks. The table 3 summarizes all various attributes of the benchmark datasets. We cover each one of them in more detail along with their results in the following sections.

Dataset	# of Sequences	Sequence lengths	Time (s) / Sequence	# of runs
EteRNA100	100	12-400	300	3
Rfam-Modena	29	54-451	300	3
Rfam-Runge	100	50-446	120	3
Rfam-Test	7500	50-452	30	1

Table 3: *Benchmark dataset summary.*

The tests were carried out using the energy parameters from Zuker, Turner et. al. 1999 [3] which were the default parameters for *ViennaRNA* < 2.0. The reason for choosing this parameter set is that the all the sequences for the *EteRNA100* benchmark were designed under these parameters. In order to maintain consistency the same parameters were used for other benchmarks as well.

Each of the benchmarks were run for a different timeout given for solving each sequence, averaged over 3 runs. In the case when either of the algorithms did not provide a result before the time limit, the process was terminated and the run was counted as unsuccessful.

EteRNA100

One of the benchmarks used is the *EteRNA100* dataset [44]. The *EteRNA* project is an online collaborative game where players compete against each other to generate valid nucleotide sequences for various RNA structures. Even though the players of this game

are non-experts in the field, this experiment has produced very interesting results and the combined knowledge of the community has provided many insights for the inverse RNA folding problem. [44] Currently expert players of the *EteRNA* project are the best solvers for multiple difficult RNA design problems, outperforming all algorithms to date. [45]

This dataset consists of 100 sequences that were created by the best performing Eterna players based on their assessment of how difficult certain structural motifs and their combinations are. This dataset includes a lot of symmetrical structures, fractal-like patterns and various artificial structures that resemble ones generated in the RNA nano-engineering experiments. The secondary structures in the dataset are ordered based on their rated difficulty, with sequence #100 being the hardest to solve. This dataset is notoriously difficult for all of the existing RNA inverse folding methods and most of them can only solve about half of the secondary structures, mostly from the first half of the dataset.

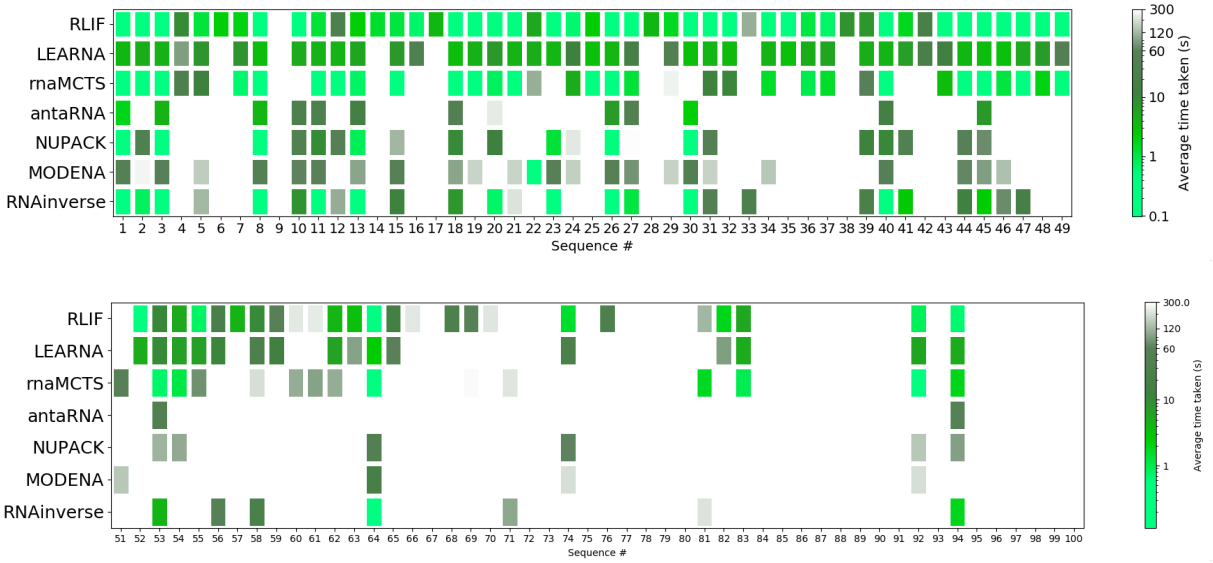

Figure 10: *Results on the EteRNA100 benchmark. Rows represent different methods and columns correspond to each sequence in the dataset. Colors indicate the time taken for obtaining a valid solution.*

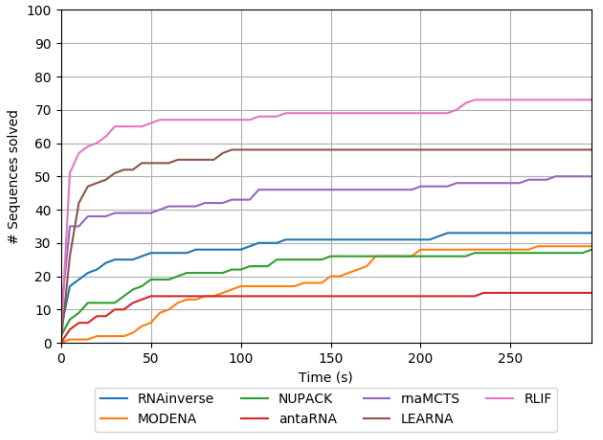

Figure 11: *Number of sequences solved at specific points of the time limit.*

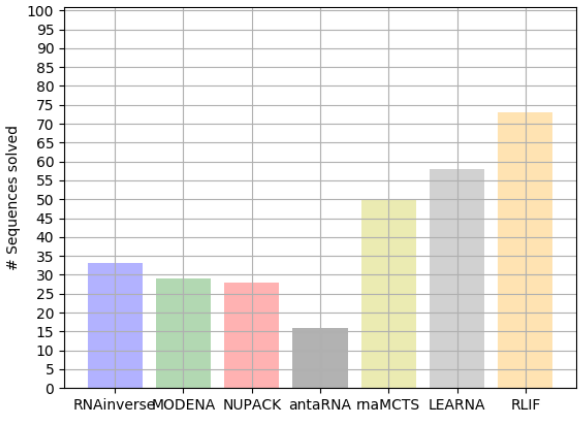

Figure 12: *Total number of sequences solved on EteRNA100 with 300s time limit per sequence.*

Figures (10-12) summarize the results on the EteRNA100 benchmark for the 7 methods. Based on these results we can tell that *RLIF* model achieves the best performance both in terms of speed and accuracy. Given 5 minutes per sequence the model solves 73/100 sequences of the EteRNA100 benchmark. The previous best reported score on this benchmark was in the publication of *LEARNA* which reported 68/100 sequences with a 24h timeout for each target sequence. Furthermore, 53 sequences take less than 10 seconds to solve which is also a marked improvement over other methods.

RFAM-Runge benchmark

This benchmark was introduced by Runge et al. [12] and it consists of 100 sequences from RFAM [46] database that range from 50-446 in length and contain a diverse range of structural motifs and shapes. The sequences in this dataset are ordered based on their length.

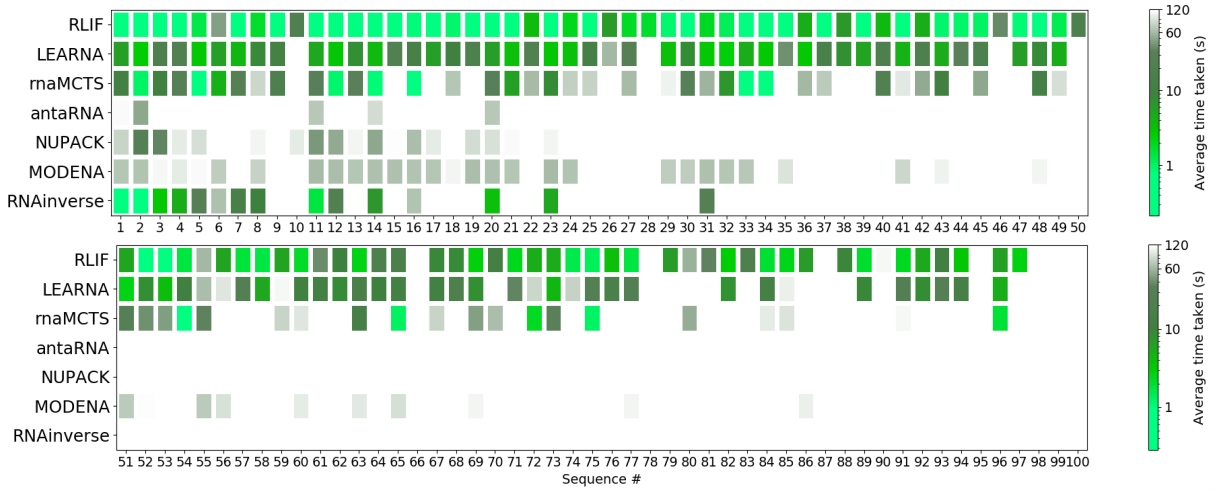

Figure 13: *Results on the Rfam-Runge benchmark. Rows represent different methods and columns correspond to each sequence in the dataset. Rectangle colors indicate the time taken for obtaining a valid solution, white color indicates no solution.*

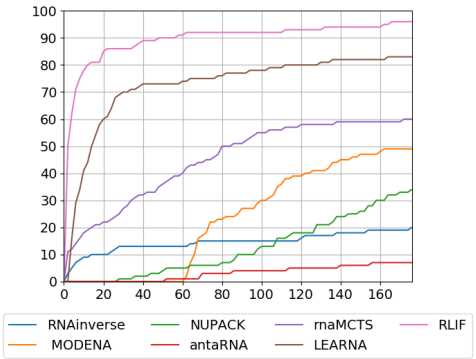

Figure 14: *Number of sequences solved at specific points of the time limit.*

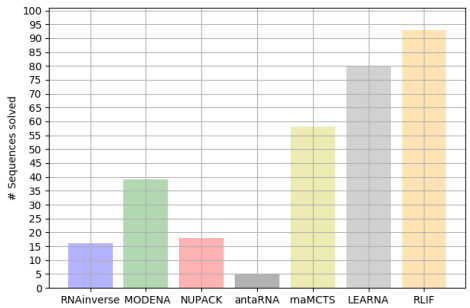

Figure 15: *Total number of sequences solved on Rfam-Runge benchmark with 120s time limit per sequence.*

A similar trend is seen for this benchmark as well. *RLIF* model generates the most solutions in the lowest amount of time.

RFAM-Modena benchmark

The last benchmark used for comparison was first introduced by authors of the *MODENA* inverse RNA folding algorithm. [10] It has been used in other research papers [12] and it consists of 29 difficult natural RNA secondary structure sequences from the *R-Fam* database.

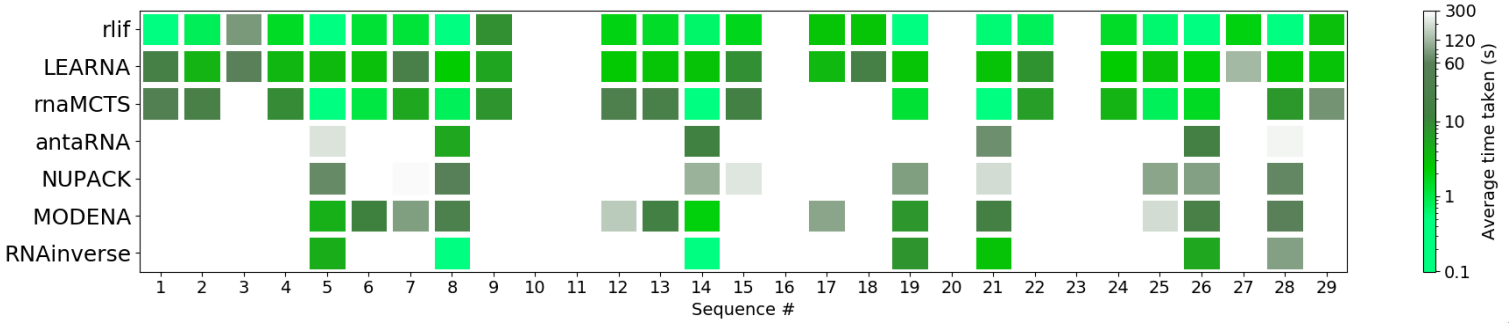

Figure 16: *Results RFAM-Modena benchmark. Rows represent different methods and columns correspond to each sequence in the dataset. Rectangle colors indicate the time taken for obtaining a valid solution, white color indicates no solution.*

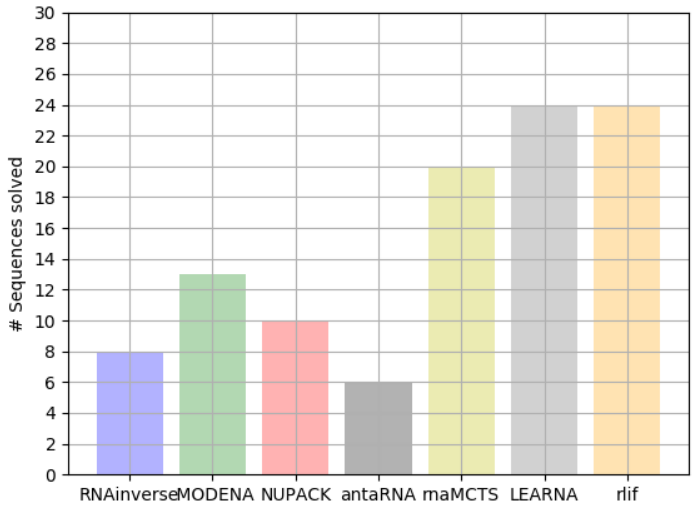

Figure 17: *Results on the EteRNA100 benchmark.*

For this dataset both *LEARNA* and *RLIF* generated the same number of valid solutions. Additional extended runs for this dataset have shown that he five remaining sequences in the dataset are really difficult to generate solutions for and it seems that the current methods for RNA inverse folding are not capable of solving them as of yet.

5.1.2 Test set

The test set for evaluating *RLIF* algorithm consisted of 7500 sequences varying from 50-446 in length. These sequences were obtained from the *R-Fam* database and it contains various families of RNA structures. For solving each sequence the model was given 30 seconds.

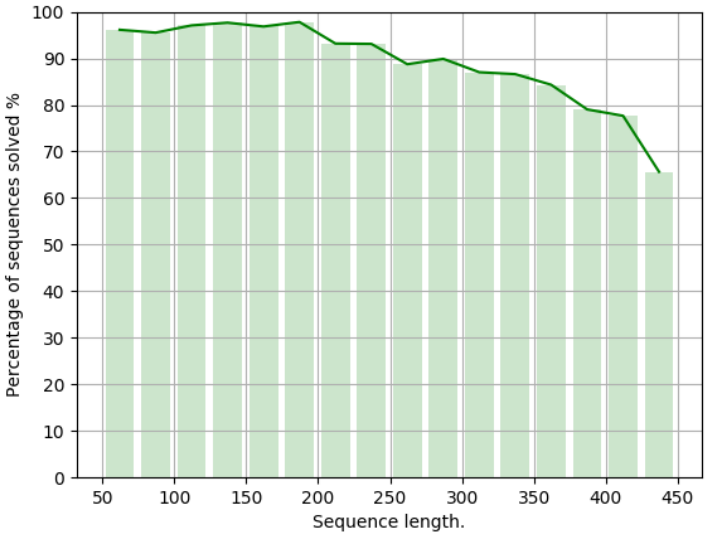

Figure 18: *The percentage of solved target structures based on their lenght.*

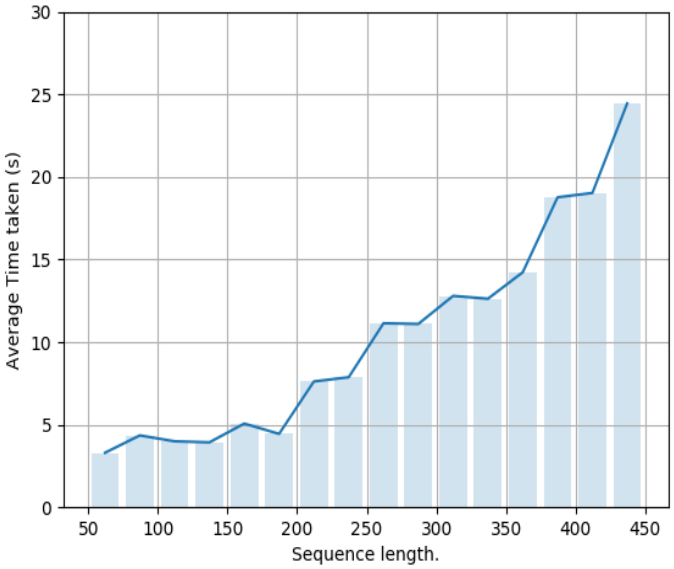

Figure 19: *Relation of the length of the sequence to the average time taken to produce a solution.*

In total 90.2% (6980/7500) sequences were solved under the 30 second limit per sequence. Figures (16-17) show how the *RLIF* model handles sequences of varying lengths. We can see that the has a relatively stable time performance curve which results from the fact that the generates the nucleotide sequence end-to-end only performing an evaluation once. The policy obtained during training generalizes well enough to solve a 41.6% (3124/7500) of the target sequences from the first attempt.

6 Conclusion

The *RLIF* reinforcement learning model for inverse RNA folding has proven to be a well performing method in terms of speed and the range of RNA secondary structures that it is capable of solving. Having only local nucleotide sequence and secondary structure information as a state representation seems to be sufficient to solve large subset of RNA design problems, specifically for naturally occuring structures. However, benchmarks that are specifically designed to test the drawbacks of RNA inverse folding methods such as EteRNA100 do show that there are still many improvements to be made in future algorithms built for this problem.

The speed of the model introduced in this thesis makes it suitable for real-time applications. The graphical user interface developed for the usage of this model illustrates this ability and it can be used to interactively design various RNA structures and find nucleotide sequences that would fold to that shape.

6.1 Future work

RLIF algorithm can rapidly produce valid sequence solutions for a diverse set of structures. However, these are not the only criteria which are important when evaluating inverse folding algorithms. Several other methods that were used for comparison in the benchmarks offer more options in terms of specifying sequence constraints, regulating the target GC content, while some of the algorithms can also handle pseudoknot structures. These attributes are among the main directions in which the *RLIF* model could be improved in the future.

The results indicate that sequences for which the model cannot find valid solutions contain various difficult structural motifs. Most of these motifs require some form of global sequence and structure knowledge that is not available when using the current implementation of state representation bounded by the state window. Several more globally oriented state representations have already been tested during the project but have not been incorporated into the main model. One of the possible ways to include more global sequence features is to use a graph-based representation of the surroundings of the current nucleotide. The subgraph of the surrounding of the current nucleotide can be vectorized using *Word2Vec*[47] for creating graph embedding or by directly using graph convolutional networks within the RL model. The initial tests have shown that learning with such state representations is possible but is much slower and the results are not yet competetive when compared to the current implementation.

The interface designed for using *RLIF* also has many avenues for potential improvements and it could be improved to be a more general RNA analysis tool in the future.

References

[1] John S Mattick. "RNA regulation: a new genetics?" In: *Nature Reviews Genetics* 5.4 (2004), p. 316.

[2] John S Mattick and Igor V Makunin. "Non-coding RNA". In: *Human molecular genetics* 15.suppl_1 (2006), R17–R29.

[3] Michael Zuker, David H Mathews, and Douglas H Turner. "Algorithms and thermodynamics for RNA secondary structure prediction: a practical guide". In: (1999), pp. 11–43.

[4] Ivan Dotu et al. "Complete RNA inverse folding: computational design of functional hammerhead ribozymes". In: *Nucleic acids research* 42.18 (2014), pp. 11752–11762.

[5] Matan Drory Retwitzer et al. "An efficient minimum free energy structure-based search method for riboswitch identification based on inverse RNA folding". In: *PloS one* 10.7 (2015), e0134262.

[6] Sven Findeiß et al. "In silico design of ligand triggered RNA switches". In: *Methods* 143 (Apr. 2018). DOI: 10.1016/j.ymeth.2018.04.003.

[7] Maung Nyan Win and Christina D Smolke. "Higher-order cellular information processing with synthetic RNA devices". In: *Science* 322.5900 (2008), pp. 456–460.

[8] Isil Severcan et al. "A polyhedron made of tRNAs". In: *Nature chemistry* 2.9 (2010), p. 772.

[9] Alexander Churkin et al. "Design of RNAs: comparing programs for inverse RNA folding". In: *Briefings in bioinformatics* 19.2 (2017), pp. 350–358.

[10] Akito Taneda. "MODENA: a multi-objective RNA inverse folding". In: *Advances and applications in bioinformatics and chemistry: AABC* 4 (2011), p. 1.

[11] Xiufeng Yang et al. "RNA inverse folding using Monte Carlo tree search". In: *BMC bioinformatics* 18.1 (2017), p. 468.

[12] Frederic Runge et al. "Learning to Design RNA". In: *CoRR* abs/1812.11951 (2018). arXiv: 1812.11951. URL: http://arxiv.org/abs/1812.11951.

[13] Peter Eastman et al. "Solving the RNA design problem with reinforcement learning". In: *PLoS computational biology* 14.6 (2018), e1006176.

[14] Jade Shi, Rhiju Das, and Vijay S Pande. "SentRNA: Improving computational RNA design by incorporating a prior of human design strategies". In: *arXiv preprint arXiv:1803.03146* (2018).

[15] Reza Jafari, Mohammad Masoud Javidi, and Marjan Kuchaki Rafsanjani. "Using deep reinforcement learning approach for solving the multiple sequence alignment problem". In: *SN Applied Sciences* 1.6 (May 2019), p. 592. ISSN: 2523-3971. DOI: 10.1007/s42452-019-0611-4. URL: https://doi.org/10.1007/s42452-019-0611-4.

[16] Benjamin Sanchez-Lengeling and Alán Aspuru-Guzik. "Inverse molecular design using machine learning: Generative models for matter engineering". In: *Science* 361.6400 (2018), pp. 360–365.

[17] G. Czibula, Maria-Iuliana Bocicor, and Istvan-Gergely Czibula Babeş-Bolyai. "A Reinforcement Learning Model for Solving the Folding Problem". In: (2011).

[18] Ronny Lorenz et al. "ViennaRNA Package 2.0". In: *Algorithms for molecular biology* 6.1 (2011), p. 26.

[19] I. L. Hofacker et al. "Fast folding and comparison of RNA secondary structures". In: *Monatshefte für Chemie / Chemical Monthly* 125.2 (Feb. 1994), pp. 167–188. ISSN: 1434-4475. DOI: 10.1007/BF00818163. URL: https://doi.org/10.1007/BF00818163.

[20] Ivo L Hofacker et al. "Fast folding and comparison of RNA secondary structures". In: *Monatshefte für Chemie/Chemical Monthly* 125.2 (1994), pp. 167–188.

[22] Akito Taneda. "Multi-objective optimization for RNA design with multiple target secondary structures". In: *BMC bioinformatics* 16.1 (2015), p. 280.

[24] Robert Kleinkauf et al. "antaRNA–Multi-objective inverse folding of pseudoknot RNA using ant-colony optimization". In: *BMC bioinformatics* 16.1 (2015), p. 389.

[26] Joseph N Zadeh et al. "NUPACK: analysis and design of nucleic acid systems". In: *Journal of computational chemistry* 32.1 (2011), pp. 170–173.

[30] Vincent Moulton et al. "Metrics on RNA secondary structures". In: *Journal of Computational Biology* 7.1-2 (2000), pp. 277–292.

[33] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[34] Richard S. Sutton and Andrew G. Barto. "Introduction to Reinforcement Learning". In: (1998).

[40] David H. Mathews et al. "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure". In: *Proceedings of the National Academy of Sciences* 101.19 (2004), pp. 7287–7292. ISSN: 0027-8424. DOI: 10.1073/pnas.0401799101. eprint: https://www.pnas.org/content/101/19/7287.full.pdf. URL: https://www.pnas.org/content/101/19/7287.

[41] Douglas H Turner, Naoki Sugimoto, and Susan M Freier. "RNA structure prediction". In: *Annual review of biophysics and biophysical chemistry* 17.1 (1988), pp. 167–192.

[42] Mirela Andronescu et al. "Efficient parameter estimation for RNA secondary structure prediction". In: *Bioinformatics* 23.13 (July 2007), pp. i19–i28. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btm223. eprint: http://oup.prod.sis.lan/bioinformatics/article-pdf/23/13/i19/23707840/btm223.pdf. URL: https://doi.org/10.1093/bioinformatics/btm223.

[43] Erin M. Langdon et al. "mRNA structure determines specificity of a polyQ-driven phase separation". In: *bioRxiv* (2017). DOI: 10.1101/233817. eprint: https://www.biorxiv.org/content/early/2017/12/13/233817.full.pdf. URL: https://www.biorxiv.org/content/early/2017/12/13/233817.

[44] Jeff Anderson-Lee et al. "Principles for predicting RNA secondary structure design difficulty". In: *Journal of molecular biology* 428.5 (2016), pp. 748–757.

[45] Paul P Gardner and Robert Giegerich. "A comprehensive comparison of comparative RNA structure prediction approaches". In: *BMC bioinformatics* 5.1 (2004), p. 140.

[46] Sam Griffiths-Jones et al. “Rfam: an RNA family database”. In: *Nucleic acids research* 31.1 (2003), pp. 439–441.

Web resources

[21] *ViennaRNA package: A C code library and several stand-alone programs for the prediction and comparison of RNA secondary structures.* URL: https://github.com/ViennaRNA/viennarna.

[23] *MODENA (Multi-Objective Design of Nucleic Acids).* URL: http://rna.eit.hirosaki-u.ac.jp/modena/multi/.

[25] *antaRNA: Designing bistable RNA.* URL: https://github.com/RobertKleinkauf/antarna.

[27] *NUPACK: Nucleic acid sequence design via efficient ensemble defect optimization.* URL: http://www.nupack.org/downloads.

[28] *LEARNA: Learning to Design RNA.* URL: https://github.com/automl/learna.

[29] *RNA-MCTS.* URL: https://github.com/tsudalab/MCTS-RNA.

[31] Trapit Bansal et al. *Emergent Complexity via Multi-Agent Competition.* 2017. arXiv: 1710.03748 [cs.AI].

[32] Alexandre Laterre et al. *Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization.* 2018. arXiv: 1807.01672 [cs.LG].

[35] *Python: language reference.* URL: https://docs.python.org/2/reference/.

[36] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[37] *Qt: Inteface design.* URL: https://github.com/qt.

[38] *Draw RNA: Michelle Wu - Generate secondary structure diagrams for nucleic acids.* URL: https://github.com/wuami/draw_rna.

[39] *Stable Baselines: reinforcement learning library.* URL: https://github.com/hill-a/stable-baselines.

[47] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space.* 2013. arXiv: 1301.3781 [cs.CL].

Appendix

Usage of other RNA inverse folding algorithms

In order to make fair comparisons, all benchmarks were run using the same energy parameters. These parameter files are included in the *ViennaRNA* package and have to be set for each method individually.

- RNAinverse

```
./RNAfold -P ../rna_turner1999.par
```

- NUPACK - the executable takes an input file containing the secondary RNA structure in dot-bracket notation as input.

```
./design -T 37 -material rna1999 <name_of_the_input_file>

    Input file contents:

        ---
        (((....)))
        ---
```

- MODENA - for this algorithm a specially formatted input file with objectives has to be generated and used as input for the executable.

```
./modena <input_file>

    Input File Formatting:

        ---
        (((....)))

        ;
        -1*((F:CONT-50)^2)^0.5
        -1*(C:FE-B:EFE)
        ;
        B RNAfold-p 1 "-d2" "-P" "../rna_turner1999.par"
        C RNAeval 1 "-d2" "-P" "../rna_turner1999.par"
        ---
```

The following three methods are written in Python, therefore their source code was modified and integrated directly into the testing pipeline. Instead of using the *ViennaRNA* through *subprocess* module calls, they were modified to use the same *ViennaRNA* Python bindings as *RLIF*. This ensures that exactly the same configuration is used for all of the algorithms.

- antaRNA
- RNA-MCTS
- LEARNA* - for this method a custom Python class was written to obtain the generated nucleotide sequences as it does not have a direct interface for sequence input. This custom class is available at in the source code:

  `rlif/rlif/utils/comparison/learna.py`