This is a preprint of the following chapter: Runge F and Hutter F, Machine Learning for RNA Design: LEARNA, published in RNA Design: Methods and Protocols, edited by Churkin A and Barash D, 2024, Humana Press reproduced with permission of Springer Science+Business Media, LLC, part of Springer Nature. The final authenticated version is available online at: http://dx.doi.org/10.1007/978-1-0716-4079-1_5.

Machine Learning for RNA Design: LEARNA

Frederic Runge

University of Freiburg

Department of Computer Science

79110 Freiburg, Germany

runget@cs.uni-freiburg.de

Frank Hutter

University of Freiburg

Department of Computer Science

79110 Freiburg, Germany

fh@cs.uni-freiburg.de

Abstract

Machine learning algorithms, and in particular deep learning approaches, have recently garnered attention in the field of molecular biology due to remarkable results. In this chapter, we describe machine learning approaches specifically developed for the design of RNAs with a focus on the learna_tools Python package, a collection of automated deep reinforcement learning algorithms for secondary structure-based RNA design. We explain the basic concepts of reinforcement learning and its extension automated reinforcement learning and outline how these concepts can be successfully applied to learn the design of RNAs. The chapter is structured to guide through the usage of the different programs with explicit examples, highlighting particular use cases of the individual tools.

Key words:

- Machine Learning
- Automated Machine Learning
- Deep Learning
- Reinforcement Learning
- Automated Reinforcement Learning
- RNA Design
- Partial RNA Design

1 Introduction

1.1 Machine Learning Methods for RNA Design

Machine learning (ML) is a field of artificial intelligence (AI) focused on building systems that can learn from and make decisions based on data. Unlike traditional programming, where rules are explicitly defined, ML algorithms identify patterns in data and make predictions or decisions without being explicitly programmed for each task. Machine learning algorithms have recently garnered significant attention in both academic and industrial circles, particularly within the realm of molecular biology***(1)(2)(3)***. Among these, deep learning (DL)***(4)*** methodologies stand out as especially promising due to their ability to leverage the vast quantities of experimental data generated by wet laboratories. In contrast to more traditional ML approaches that typically require substantial data preprocessing to derive meaningful input features, deep learning can directly be applied to raw data to learn a useful representation by processing it in various layers of a deep neural network (DNN). The ultimate objective of these models is to generalize and apply their learned insights to new, unseen tasks. Theoretically, even basic neural networks (NNs) are universal function approximators***(5)(6)***, which propelled deep learning to the forefront of numerous fields with remarkable successes***(7)(8)(9)***.

In light of these advancements, there has been a notable emergence of machine learning algorithms aimed at addressing the complex challenges of RNA structure and function prediction

problems***(10)(11)(12)(13)(14)***. RNA design, however, considers the inverse problem: Given a set of desired properties or functions, find one or more RNA nucleotide sequences with the desired features. Different machine learning approaches were proposed to tackle the problem from various perspectives.

Some researchers focus on learning to generate RNAs from nucleotide sequence data directly to e.g. aid in specific application areas such as aptamer discovery through Systematic Evolution of Ligands by Exponential Enrichment (SELEX)***(15)***. For this application, methods typically try to reduce the scale of initial libraries for experimental screening approaches, which is vital for cost and efficiency in experimental workflows. These varied approaches include using deep learning with recurrent neural networks (RNNs)***(16)***, variational autoencoders (VAE)***(17)***, and other innovative techniques***(18)(19)***. The basic learning paradigm of such algorithms is to learn a model from RNA sequence data to generate similar (but different) sequences to the training data set at test time.

Another critical aspect of RNA design revolves around its secondary structures. The function of a molecule is intrinsically linked to its structure, and thus, algorithms that can generate RNA sequences folding into desired structures are highly sought after. In this regard, ***(20)*** recently proposed a generative approach consisting of a deep learning based scoring network and a second model based on a generative adversarial network (GAN)***(21)*** for the generative design of toehold switches***(22)***. This approach bears great potential, since it incorporates knowledge of the RNA sequence and its secondary structure into the design process.

In general, research to secondary structure-based RNA design can be roughly divided into three main approaches: one incorporates human insights into the design process***(23)(24)*** gained from the online game Eterna***(25)***, the second one leverages secondary structure information to modify an initial sequence until it folds into the desired target structure***(26)***, while the third employs a generative design approach based on structure information***(20)(27)(28)***.

In this chapter, we delve into the latter category of algorithms, focusing particularly on the LEARNA framework and its related algorithms available in the learna_tools Python package. The package gathers a set of generative approaches to the RNA design problem that employ automated reinforcement learning. We will provide an in-depth exploration of these algorithms, offering detailed insights into their backgrounds, installation processes, input formats, and practical applications. The chapter is structured to guide the reader through the necessary background on reinforcement learning, the basic principles of the LEARNA framework, and its various implementations. We will also provide insights into a new RNA design paradigm that considers both, sequence- and structure-based design from provided motifs and an algorithm to solve it.

For reasons of simplicity, we often use LEARNA as an umbrella term for the algorithms of the learna_tools package in this chapter since the algorithms share a general framework. However, we clearly mark cases where the algorithms differ and explain individual use cases and properties where necessary.

1.2 Reinforcement Learning

In order to understand the basic ideas behind the LEARNA framework, we would like to give a brief introduction to the concept of reinforcement learning (RL)***(29)***. Reinforcement learning is a branch of machine learning where an agent learns optimal behavior through trial-and-error interactions with its environment. It involves three key elements: states, actions, and rewards. The agent observes a state provided by the environment (the situation or condition it is in), takes an action, and then receives a reward based on the outcome of that action. The goal is to maximize the cumulative rewards over time.

Deep Reinforcement Learning (DRL) is an advanced form of RL that integrates deep learning, particularly useful for environments with complex or high-dimensional states. In DRL, deep neural networks are used to interpret these states and assist the agent in decision-making. This integration

allows the agent to process and learn from complex, intricate data inputs, such as images or sequential patterns, which is a common requirement in many sophisticated applications.

An important aspect of DRL is the optimization of the policy, the strategy that the agent follows to decide its actions. The LEARNA family of algorithms uses Proximal Policy Optimization (PPO)***(30)***. PPO optimizes the policy network in a manner that prevents drastic changes in the policy between successive learning iterations. This controlled approach to optimization ensures a stable and consistent learning process, making PPO an attractive choice for tasks where balance and reliability in learning are crucial.

Through the combination of deep learning and sophisticated policy optimization techniques, DRL enables agents to learn efficiently in complex environments.

According to the common convention in the field, we will use the term reinforcement learning (RL) in the remainder of this chapter as a synonym for deep reinforcement learning, since all of the described algorithms employ deep neural networks to approximate the policy of the agent.

1.3 Automated Reinforcement Learning

During the development of an algorithm, developers often have to decide about certain parameters that have to be set in advance. These so-called hyperparameters can have substantial impact on the performance of an algorithm, particularly in the field of reinforcement learning (RL)***(31)***. Automated reinforcement learning (AutoRL)***(32)*** enhances RL by integrating Automated Machine Learning (AutoML)***(33)*** principles. It aims to automate critical aspects of RL, such as selecting algorithms, optimizing hyperparameters, and designing network architectures, seeking to mitigate intensive manual tuning and enhance efficiency and accessibility. AutoRL is particularly beneficial in complex scenarios where manual optimization is impractical, resource-intensive, or in situations where there is no prior knowledge about good parameter settings.

1.4 RNA Design as a Reinforcement Learning Problem

As outlined in Section 1.1, reinforcement learning (RL) describes a periodic interaction between an agent and its environment. The crucial steps to define RNA design as a RL problem are (1) to provide informative states that guide the agent's decisions, (2) to define actions that represent the design of an RNA sequence, and (3) to define a reward signal that helps the agent to learn a policy that yields the desired outcomes. The choice of the architecture of the policy network, the method for policy optimization, the training process, as well as other hyperparameters that have to be set in advance are further challenges for developers of an RL system for RNA design.

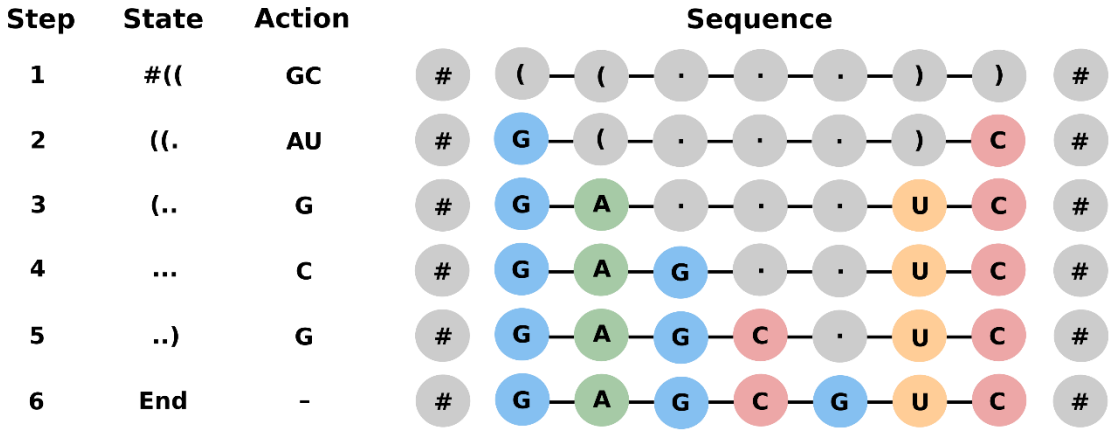
Two RL approaches have been proposed to tackle the RNA design problem. ***(26)*** use a convolutional neural network architecture***(4)*** for the policy network. The agent is trained to modify a sequence, given to the network as a $N \times 4$ tensor, where N is the number of bases, representing the current sequence in one-hot encoding. As a starting point, ***(26)*** use a randomly generated input sequence with the same length as a provided secondary structure. The states are defined as the entire input sequence; actions correspond to modifying the type of a single nucleotide at a certain position, or in some cases two nucleotides. After each episode the modified sequence is folded using ViennaRNA's RNAFold***(34)***, and the agent receives a fixed positive reward if the resulting sequence is predicted to fold into the target structure. All other actions receive a reward of zero.

In contrast, the LEARNA algorithms tackle the problem with a generative approach. Given a target secondary structure, LEARNA predicts a nucleotide for each position of the target structure without receiving intermediate rewards. Once all sites have been assigned nucleotides, the designed candidate sequence is folded using RNAFold and compared to the desired structure.

The **actions** correspond to placing a nucleotide for a given position in the target structure. Consequently, LEARNA can choose from four actions that correspond to 'A', 'C', 'G', and 'U'. However, the algorithm leverages further information provided with the target structure: If a certain

position ought to be paired, the algorithm places two nucleotides in a single step using the complementary nucleotide for the paired position, following Watson-Crick base interactions ('A-U', 'U-A', 'G-C', 'C-G').

The **states** are defined as local representations of the input structure. More precisely, for each position in the target structure, LEARNA uses an n-gram centered around the current position of the target structure. In essence, this approach can be interpreted as a window sliding over the target structure to provide a local view for each position. To be able to define the n-gram at any position (for example at position 1 the upstream part of the sequence would be undefined), the target structure is first padded with a padding symbol. The size of the n-gram is a hyperparameter that has to be defined in advance. Figure 1 gives an example of an action rollout of LEARNA including state representations for a window size of three. However, we note that the actual size of the window for the algorithms of the learna_tools package is between 21 and 65.



[Figure 1] Illustration of an action rollout of the LEARNA algorithms. The agent sequentially builds a candidate sequence by choosing actions to place nucleotides. At paired sites, as indicated by a pair of brackets, two nucleotides are placed simultaneously (step 1 and step 2); while at unpaired sites a single nucleotide is placed (step 3-5). The actions are chosen based on the states provided by the environment. The states represent a local view on the target structure (here using a window size of three). To be able to build the states at each position, the target structure is padded at each end with a padding symbol '#'.

Once all sites have been assigned nucleotides, the **reward** is computed based on the Hamming distance***(35)*** between the target structure and the structure derived from folding the predicted candidate sequence with RNAFold. The distance is normalized by the length of the input structure, which provides a final distance value between zero (perfectly matching structures) and one (completely different structures). To provide a more informative reward signal for optimization, LEARNA uses a hyperparameter α that shapes the reward. The final reward function for structure-based RNA design with LEARNA then looks as follows:

$$R = (1 - \frac{D}{L})^{\alpha},$$

where *D* is the Hamming distance, *L* is the length of the target structure, and α is the parameter for shaping the reward.

The LEARNA framework encompasses three strategies for the design of RNAs: LEARNA, Meta-LEARNA, and Meta-LEARNA-Adapt. We will detail these strategies in Section 3. The described formulation of the states, actions, and the reward function, however, applies for all three versions. For a more formal description of the LEARNA approach, we refer the interested reader to ***(27)***.

However, the learna_tools package also provides libLEARNA***(28)***, an extension to the original LEARNA framework that considers a more general RNA design paradigm, partial RNA design***(36)***. We briefly outline this RNA design formulation in the following section.

1.5 The Partial RNA Design Paradigm

The design of RNA molecules often involves integrating distinct RNA fragments, known as motifs, each carrying a specific function***(37)(38)(39)***. The new paradigm of partial RNA design***(36)*** follows

this concept by allowing flexible and precise RNA design from provided sequence and structure motifs.

Traditionally, RNA design has been constrained by the need to adhere to a single, predetermined secondary structure. However, this approach does not always align with the multifaceted nature of RNA design endeavors. Consider, for instance, designing an RNA molecule for ligand-dependent gene expression regulation. This task might involve combining a riboswitch***(40)*** with a Shine-Dalgarno sequence***(41)***, necessitating different types of motifs: a motif with specific sequence and structural requirements for binding the ligand (the aptamer), a variable-length motif (a spacer region), a motif focusing solely on secondary structure features (the expression platform), and a sequence-specific motif without structural constraints (the Shine-Dalgarno sequence).

Existing RNA design methods fall short in addressing such multifaceted design tasks, leading to the development of partial RNA design and the libLEARNA algorithm.

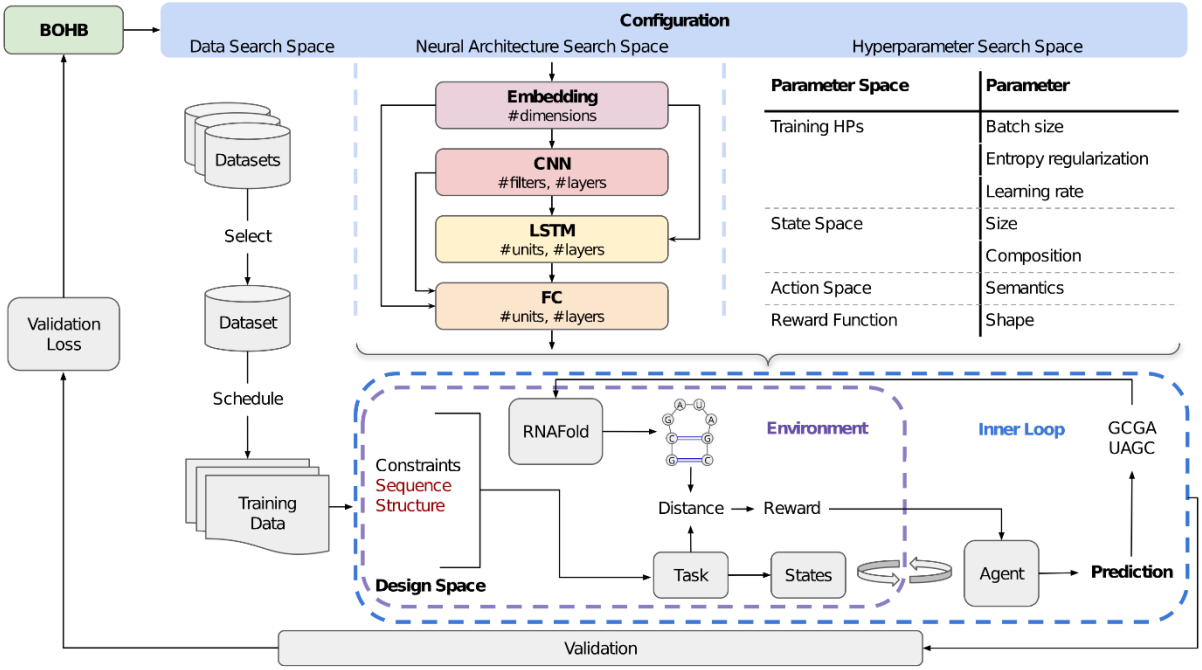
In this approach, we formulate motifs that constrain specific regions of the sequence and secondary structure of an RNA, while leaving the other regions unconstrained. These unconstrained parts can then be defined as variable length regions which can be filled with arbitrary motifs by an algorithm***(36)***. The design can be viewed as a search in a partially restricted design space that allows for exploration by a machine to provide diverse solutions. libLEARNA leverages these constraints to navigate the RNA design space, ensuring the inclusion of desired motifs while exploring various combinations within the set parameters.

Partial RNA design represents a significant shift from conventional RNA design methods. It offers greater flexibility, enabling the design of complex RNA molecules without the exhaustive need to specify and refine a target structure. However, designing effective search spaces for libLEARNA does require thoughtful consideration to achieve the intended outcomes, which we will explore in detail in Section 3 (see also note 1 and note 5).

1.6 Automated Reinforcement Learning for RNA Design

For most practitioners, it is useful to have an algorithm that can be applied out-of-the-box to a problem at hand. Therefore, the learna_tools package provides pretrained models with exhaustively tuned parameters. However, to provide the full picture of the LEARNA framework, we briefly introduce the underlying automated reinforcement learning (AutoRL) approach used to derive the different models of the package.

As already explained in Section 1.3, developing a reinforcement learning algorithm for the design of RNAs is a challenging task. The architecture of the policy network, the hyperparameters, as well as other parts of the pipeline, require careful selection and tuning to achieve good performance. Since tuning these parameters manually is tedious, error prone and time-consuming, all algorithms provided with the learna_tools package follow an AutoRL approach that jointly optimizes the neural architecture as well as other hyperparameters in a single algorithm run. The AutoRL framework is implemented as a two-stage algorithm: In an inner-loop, a reinforcement learning (RL) algorithm learns an RNA design policy across thousands of different RNA design tasks. Afterwards, the algorithm is evaluated on a hold-out validation set and the validation loss is communicated to an efficient Bayesian Optimization method, *BOHB**(42)***, which jointly optimizes the configuration of the RL system in an outer-loop. The RL agent in the inner loop seeks to maximize its reward with each trial, i.e., designing an RNA sequence for a given task, while the meta-optimizer (BOHB) tries to minimize the validation loss with each new configuration sampled. The best performing RL algorithm based on the validation loss is finally saved and can be used for further evaluations. The basic AutoRL loop is outlined in Figure 2.



[Figure 2] Outline of the automated reinforcement learning loop of the LEARNA algorithms. In the outer loop, the optimizer (BOHB) samples a configuration for an RL system. The configuration defines the neural architecture of an agent, the shape of the reward function, the state space, and different other training hyperparameters. Once defined, the RL system is trained in the inner loop and validated afterwards. The resulting validation loss is communicated to the optimizer which updates its model based on the observed loss to sample a new configuration for the next iteration. The best performing configuration is used for final evaluations.

Depending on the specific algorithm, the configuration space of the RL system contains 14 to 18 dimensions that are jointly optimized with BOHB. These include environment parameters to define the state space, the shape of the reward function, and the action semantics, training hyperparameters like the learning rate for optimization, parameters that define the neural architecture of the policy network of the RL agent, as well as parameters to define the training data or the schedule of tasks presented to the agent during training. The optimization framework is provided with the source code of the learna_tools package. For information on how to run the optimization, please see note 2. A detailed description of the parameters of the individual programs of the learna_tools package and more information on the meta-optimization procedure can be found in the respective publications of LEARNA***(27)***, libLEARNA***(28)***, and BOHB***(42)***.

2 Materials

2.1 Installation and Software Requirements

The learna_tools Python package gathers the following algorithms: LEARNA, Meta-LEARNA, Meta-LEARNA-Adapt, and libLEARNA. The package requires Python 3.6 and ViennaRNA's RNAFold with python bindings. All algorithms can be executed on a single CPU. The source code and a detailed documentation can be found in the learna_tools repository on github under https://github.com/automl/learna_tools. The repository accompanying the original publication of LEARNA can be found on github at https://github.com/automl/learna.

Once you have a running version of Python 3.6 and *ViennaRNA* on your system, you can install the learna_tools package via the Python standard package manager *pip*:

```
pip install learna_tools
```

We recommend installing learna_tools within a conda environment. A detailed description of the install and some troubleshooting can be found in the github repository at https://github.com/automl/learna_tools. We further note that, for running the provided examples in the following sections, it would be useful to clone the repository from github and run the examples from the root directory:

```
git clone https://github.com/automl/learna_tools.git && cd
learna_tools
```

Optionally, learna_tools allows plotting of RNA secondary structures using the java applet VARNA***(43)***. For a seamless integration, we recommend the installation of VARNA via conda (see note 3).

3 Methods

3.1 Inverse RNA Folding with the LEARNA Approaches

3.1.1 Description and Scope of the Programs

All three versions of the LEARNA approach, LEARNA, Meta-LEARNA, and Meta-LEARNA-Adapt, consider the inverse RNA folding problem: Given a secondary structure in dot-bracket format, the algorithms design RNA sequences that fold into the desired target structure. However, each algorithm has individual strengths:

The **LEARNA** approach learns a policy from scratch whenever requested for a design, i.e. in contrast to the other algorithms, the policy of the agent is not previously trained. This has the advantage that LEARNA is not biased towards any specific sequence patterns but optimizes its policy for a specific target structure at hand.

The **Meta-LEARNA** approach samples RNA sequences for a given target structure from a pretrained policy without applying any further updates to the learned policy. This results in rather fast sampling of RNA sequences. The RNA design policy was learned across thousands of different inverse RNA folding tasks. Due to the fast sampling of candidates, Meta-LEARNA is a very useful tool when a quick solution for a given structure is required.

The **Meta-LEARNA-Adapt** approach combines the best from both worlds, LEARNA and Meta-LEARNA. In essence, Meta-LEARNA-Adapt is running Meta-LEARNA with weight updates during evaluation to be able to adapt to a given input structure at test time. With this approach, Meta-LEARNA-Adapt often solves a given task very quickly, leveraging the pretrained policy, but at the same time, the algorithm can adapt to a given task if necessary.

We discuss the advantages and potential disadvantages of the different versions in note 4.

3.1.2 Input Formats

The input to LEARNA, Meta-LEARNA, and Meta-LEARNA-Adapt is a secondary structure in dot-bracket format***(34)***, either provided in a two line fasta-like file or as direct input via the command line options --*target_structure* and --*target_id* (see Section 3.1.3). The first line of the input file provides the target Id, and the second line is the target structure in dot-bracket notation. An example input file should look as follows:

```
>Example 1
...((((........))))....
```

All algorithms additionally support to specify multiple targets in a single file:

```
>Example 1
...((((........))))....
>Example2
..(((...)))...
>Example3
((((((((((........))))))))))
```

If multiple targets are provided, the algorithms sequentially process one target after the other and report the results for each target when it was processed.

3.1.3 Basic Program Options

The command line options for the three algorithms differ only marginally. The general command line interface for the three algorithms is as follows.

```
learna [options]
meta-learna [options]
meta-learna-adapt [options]
```

For reasons of simplicity, we explain the command line interface using the *learna* command. Cases where the interface differs between the approaches will be clearly marked.

To see a list of all available command line options with descriptions, you can run:

```
learna -h
```

or

```
learna --help
```

Most options are set to the values of the configuration that achieved the lowest validation loss during the AutoRL optimization process. These options typically do not require any changes to achieve good performance. The input target structure in dot-bracket notation, however, is a mandatory argument and can either be provided in a fasta-like file as described in Section 3.1.2, or as direct input through the *--target_structure* option. The basic command to run the algorithms then looks as follows:

```
learna --target_structure "...(((...)))..."
```

or

```
learna --input_file <path to file>
```

The Id for a given task can also be provided via the command line, using the *--target_id* option. This is particularly useful when saving plots or results where the Id is used to define the filenames. If not provided, the Id will be set automatically to the index of the solution starting with one. We also note that multiple targets can only be defined in a file but not via the command line option *--target_structure*.

By default, all algorithms design a single candidate sequence for a given target structure. To change this behavior one can use the *--num_solutions* option. For example, to design five candidate sequences for a structure of a Hammerhead ribozyme, you can use:

```
learna --target_structure
".(((((....((((((.......((((......))))...(((.....))).))))))...))))).
" --num_solutions 5
```

The resulting output reports all solutions in a Markdown table format by default. The results can be saved to a specific directory via the *--results_dir <path to results directory>* option. The algorithms store results as a pandas dataframe in pickle format by default. However, the programs allow changing the output format with the *--output_format* option. Currently, the following formats are supported: “pickle”, “csv” (a comma separated file), or “fasta”. The “fasta” format saves the output to a fasta-like file format that looks as follows:

```
>Id 1
<designed sequence 1>
<corresponding structure 1 in dot-bracket format>
>Id 2
<designed sequence 2>
<corresponding structure 2 in dot-bracket format>
```

All algorithms further allow setting a runtime limit in seconds with the *--timeout <time in seconds>* option. When the limit is reached, the algorithms return all solutions found so far if there are any. When using multiple targets as input, the timeout counts for each individual target. By default, the algorithms run with a runtime limit of 600 seconds.

Sometimes, it might be helpful to also return suboptimal solutions. This is particularly helpful if the timeout is set very optimistically. You can return suboptimal solutions with the *--hamming_tolerance <natural number>* option. The option sets a threshold to include candidates with a Hamming distance

to the target structure that is smaller or equal to the provided threshold value after folding. Alternatively, you can use the --*show_all_designs* flag to return all the designed candidates of a run. Depending on the target structure and the provided runtime limit, however, this flag might result in relatively large amounts of designed candidates.

3.1.4 Advanced Program Options

The algorithms of the learna_tools package allow precise control over all individual parameters used during the optimization process via AutoRL. As an example, one can change the number of fully-connected layers of the network by setting the --*num_fc_layers* <*natural number*> option. However, the default options typically provide good results and especially the pretrained models (Meta-LEARNA, Meta-LEARNA-Adapt, and libLEARNA) do not always allow to set arbitrary combinations for all parameters due to conflicts with the saved model parameters which are loaded by default to facilitate the command line interface. Therefore, we recommend to either keep the parameters unchanged, or to run a completely new optimization from scratch (see Note 2).

LEARNA, Meta-LEARNA-Adapt (and libLEARNA) all adapt to a given target at test time by updating the weights of the policy network for the task at hand. Consequently, the algorithms might get stuck in a local minimum during optimization. To avoid this, the three algorithms provide the --*restart_timeout* <*time in seconds*> option to restart the algorithms with the initial weights of the policy network after a certain time. We discuss the behavior of the restart option in more detail in note 4.

We further would like to note that the agent is shared across all tasks when designing RNAs for multiple targets in a single run. This means that the policy is optimized across all the provided targets and not optimized on each individual target only. To avoid this behavior, you can use the --*no_shared_agent* flag to ensure a reset of the weights of the agent before processing the next target.

An explicit example command for the design of 100 candidates for the “Frog Foot” target of the Eterna100 benchmark***(44)*** using Meta-LEARNA-Adapt with a restart of the algorithm every five seconds (as used in note 4), and a runtime limit of two minutes, would look as follows:

```
meta-learna-adapt --input_file examples/if_frog_foot_example.input --restart_timeout 5 --num_solutions 100 --timeout 120
```

3.1.5 Description of the Output

All programs of the learna_tools package print general information about the current job to the console after starting. These include e.g., the number of desired solutions or the target Ids. The final solutions are printed in tabular format by default. The table contains the following fields:

- An unnamed field to index the solutions.
- **Id**: The Id of the target.
- **time**: The time required to find the solution.
- **hamming_distance**: The Hamming distance between the folded candidate sequence and the target structure.
- **rel_hamming_distance**: The Hamming distance normalized by the length.
- **sequence**: The sequence of the candidate solution.
- **structure**: The predicted structure of the candidate sequence.

In case the input consists of multiple targets, the programs print one table for each individual target. If there were no or fewer solutions found than requested, the program prints a Warning message to inform the user and reports all solutions found within the given timeout. As described before, the output format can be changed via the *--output_format* option.

The learna_tools package further provides tools for visualization of the outputs. These include logo plots for the designed sequences to plot the distribution of nucleotides at each position using the

logomaker python package***(45)***, as well as secondary structure plots using VARNA. The plotting can be triggered by setting the *--plot_logo* and *--plot_structure* flags. By default, all plots are saved into the 'plots' directory relative to the current working directory. However, one can specify a path for saving the plots using the *--plotting_dir <path to directory>* option. For showing the plots besides saving, one can use the *--show_plots* option. We show examples of logo plots in Figure 6 in Section 4 where we discuss the diversity of the generated sequences in note 4. Using the "Frog Foot" example from the Eterna100 benchmark again, the command for LEARNA to reproduce the logo plots of Figure 6 for 100 solutions without restarts of the algorithm looks as follows:

```
learna --target_structure
".........((((....))))((((....))))((((...))))" --target_id "Frog
Foot" --timeout 60 --num_solutions 100 --plot_logo --show_plots
```

3.2 RNA Design in Partially Restricted Search Spaces – libLEARNA

3.2.1 Description and Scope of the Program

libLEARNA is the most recent extension to the LEARNA framework. It uses the Meta-LEARNA-Adapt approach but allows very flexible RNA design applications. libLEARNA can design RNAs of variable lengths from provided sequence and structure motifs. The algorithm ensures that provided sequence and structure constraints are satisfied at the given positions, but it can also explore a search space automatically to find multiple reasonable solutions. We demonstrate the capabilities of libLEARNA with explicit examples in the following sections. Since libLEARNA offers a new RNA design interface, using libLEARNA for the first time might appear new to users that are familiar with more traditional RNA design approaches. However, we will guide through specific use cases to help to get more familiar with the approach and its interface.

3.2.2 Input Formats

The sequence and structure inputs for libLEARNA can only be provided in a file with specific tags. The sequence constraints can be defined after a '#seq' tag using IUPAC notation. These constraints can be provided as motifs. Positions in the input sequence that are marked with whitespace or 'X' ('Xtend here') are considered unconstrained positions where the algorithm is allowed to extend the sequence as long as a hard maximum length limit for the entire sequence is not violated. Similarly, the structure constraints (following the '#str' tag) can be provided as motifs. The algorithm allows placing 'N' in the structure constraints to mark a position that is not structurally constrained. Positional constraints for the structure are expected in dot-bracket notation ***(34)***, using the symbols '(', ')', '.', 'N' only. libLEARNA considers any motif to consist of a sequence part and a structure part of the same length. A desired GC-content can be added to the file via the '#GC' tag. By default, libLEARNA designs candidates with a tolerance of 0.01 for a given GC-content. This can be adjusted via the *--gc_tolerance* option. An example input file to design RNAs of variable lengths from two provided sequence and structure motifs could then e.g., look as follows:

```
>libLEARNA Example1
#seq AYUUNN CUMUU
#str NN..(( ))...
#GC 0.5
```

In this example file, the sequence constraints as well as the structure constraints are given as two motifs that formulate a partially restricted RNA design space. The algorithm will ensure to start each candidate sequence with 'A', 'C' or 'U' (to satisfy the 'Y' constraint given in IUPAC notation), 'U', 'U', and two positions with arbitrary nucleotides ('A', 'C', 'G', 'U') to satisfy the 'NN' constraint. The resulting structures after folding the candidate sequence will start with two arbitrary structure symbols followed by '..((('. After this motif, there might be a region that contains any combination of nucleotides and dot-bracket symbols as the whitespace between the motifs indicates a position for

potential extension. Finally, the designed candidates will end with ‘C’, ‘U’, ‘A’ or ‘C’ (satisfying the ‘M’ constraint), ‘U’, ‘U’, while the corresponding structure in dot-bracket notation will end with ‘))...’. The ‘#GC’-tag ensures a GC content of 0.5 for the designed candidates with the provided tolerance. To be more explicit, an example call of libLEARNA from the command line (if the constraints of the example above were saved into a file named example.input) to design RNAs of a maximum length of 50 nucleotides with the provided constraints would look as follows:

```
liblearna  --input_file example.input --max_length 50
```

An example candidate then could look as follows:

```
AUUUAGUUAAAAAGGGACUCUU
....((((.......))))...
```

3.2.3 Description of the Program Options

The basic program options of libLEARNA are the same as for the other algorithms of the learna_tools package. You can run

```
liblearna -h
```

to get a full list of available options. We note that libLEARNA does not support the --*hamming_tolerance* <*natural number*> option provided for the other algorithms of the learna_tools package. However, as we describe in detail in the following, libLEARNA – by design – allows to define RNA design spaces that include more than one target structure.

Since libLEARNA can design RNAs of different lengths, the algorithm has two additional options, --*min_length* and --*max_length*, for setting the length borders of a design space. Furthermore,

libLEARNA allows to specify a desired GC-content with a certain tolerance for the design with the --*desired_gc* and --*gc_tolerance* options (see also Section 3.2.2 and Section 3.2.7).

3.2.4 Description of the Output

The output of libLEARNA is in tabular format like the output of the other algorithms. However, the fields of the table differ slightly:

- An unnamed field to index the solutions.
- **Id**: The Id of the target.
- **time**: The time required to find the solution.
- **reward**: The reward received for the designed candidate solution.
- **sequence**: The sequence of the candidate solution.
- **structure**: The predicted secondary structure of the candidate solution in dot-bracket format.
- **GC-content**: The G and C nucleotide ratio of the designed candidate sequence.
- **length**: The length of the designed solution.

libLEARNA allows saving the output to a file, using the same options as described for the other algorithms of the learna_tools package, including changing the output format with the --*output_format* option, or saving structure and logo plots.

3.2.5 Inverse RNA Folding with libLEARNA

The libLEARNA approach allows to design RNAs from provided motifs. Consequently, libLEARNA can be applied to inverse RNA folding when given a single structure motif, a target secondary structure in dot-bracket format. For the “Frog Foot” example mentioned before, the input file for libLEARNA would look as follows:

```
>Frog Foot
#seq NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
#str ..........((((....))))((((....))))((((...))))
```

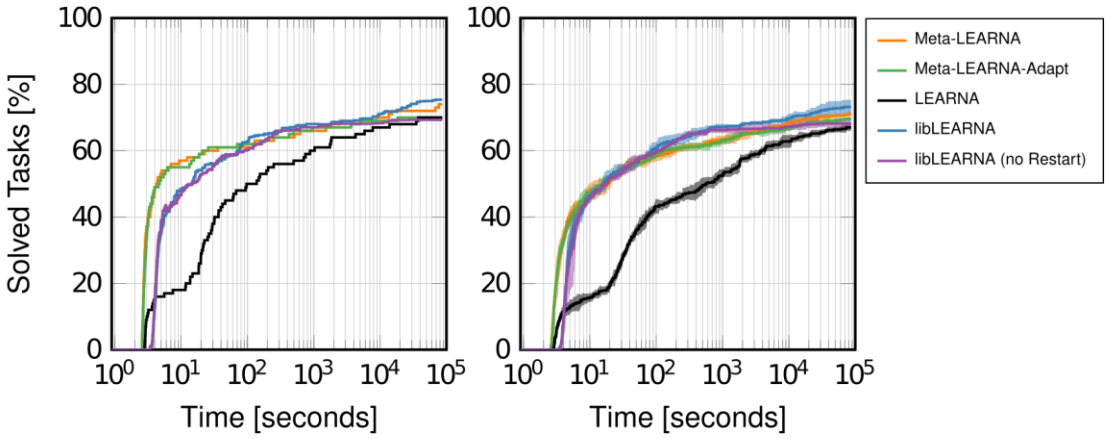
The input examples for the “Frog Foot” task are provided with the source code in the examples directory. A call to libLEARNA then could e.g., look as follows:

```
liblearna --input_file examples/if_frog_foot_example_liblearna.input
--timeout 30 --num_solutions 10
```

The task of inverse RNA folding allows to compare the different approaches of the learna_tools package to deepen the understanding and intuition of the different algorithms. We use the 100 tasks of version 2 of the Eterna100 benchmark***(48)*** for our comparison. Figure 3 shows the results of the evaluation using five independent runs with each program with the commonly used timeout of 24 hours.

We observe that all algorithms with a pretrained policy outperform LEARNA on this challenging benchmark, with libLEARNA showing the best performance, followed by Meta-LEARNA and Meta-LEARNA-Adapt. This is an interesting result; in contrast to the other algorithms, libLEARNA is not specifically trained for the task of inverse RNA folding. We discuss the training regimen of the algorithms in note 6, highlighting potential benefits of the training procedure of libLEARNA compared to the other algorithms. However, for this experiment we run libLEARNA with the exact same setup as Meta-LEARNA-Adapt***(27)*** to get comparable results. In particular, we use the *--restart_timeout* option to reset all weights of the algorithms every 1800 seconds. As discussed in note 4, this option can have substantial impact on the resulting candidate sequences, essentially preventing the algorithms from getting stuck in local minima of the design space. For Meta-LEARNA, this option is not available since the algorithm is not updating weights at all, i.e., it does not adapt to the given task at hand. We, therefore, run libLEARNA twice, with and without restarting, and observe

that the performance clearly decreases without the restarting procedure. In fact, the performance seems to plateau right at the time where the first restart would have been performed. Since the common evaluation protocol for the Eterna100 benchmark uses a long runtime of 24 hours on each task, it is not very surprising that preventing local minima via the *--restart_timeout* option can have a strong impact on the performance when running an algorithm that adapts to a given task. We, therefore, recommend using this option when the expected runtime is high, or to use Meta-LEARNA if adaptation to the task at hand is not required.



[Figure 3] Results of the comparison of LEARNA, Meta-LEARNA, Meta-LEARNA-Adapt and libLEARNA on the tasks of the Eterna100 benchmark version 2. The plot on the left side shows the accumulated number of solved tasks across five evaluation runs for each algorithm. The plot on the right shows the average number of solved tasks with the standard deviation around the mean. libLEARNA was run twice, once with restarting the algorithm every 1800 seconds, once without the restarting procedure.

3.2.6 Constrained Inverse RNA Folding with libLEARNA

In contrast to LEARNA, Meta-LEARNA, and Meta-LEARNA-Adapt, libLEARNA can include sequence constraints using IUPAC notation. Furthermore, libLEARNA allows placing unconstrained sites in the structure using the letter ‘N’. An example input file for RNA design with sequence constraints and unconstrained positions in the structure input could e.g., look as follows:

```
>partially constrained Frog Foot example
```

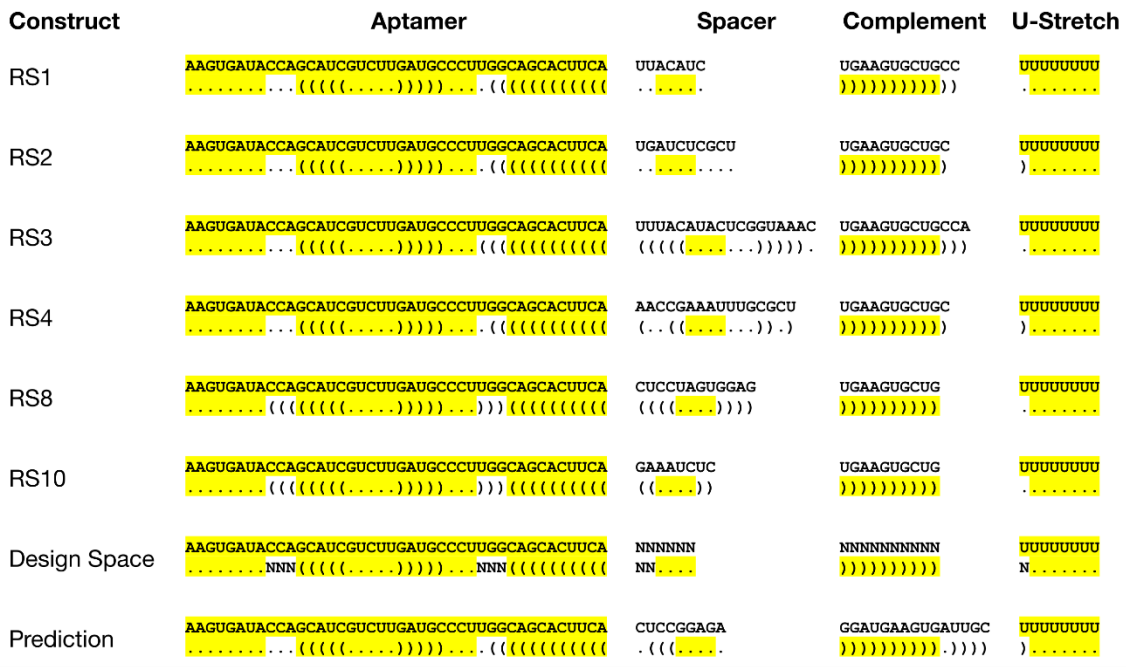
```
#seq NNNAGNNNNNNNNNNNNNNNNNNNNYNNNNNNNNNMKNNNNNNNNNN
#str ..........((((....))))NN((....))NNNN((...))))
```

The provided structure constraints of the example reveal another unique feature of libLEARNA which makes it different compared to other RNA design algorithms: There is no need to define specific pairs in advance, i.e. libLEARNA can handle any structure information using the symbols ‘)’, ‘.’, ‘(‘, ‘N’, including unbalanced brackets. This allows for more flexibility during the definition of RNA design tasks (see note 5). The partially constrained example is provided with the source code in the example directory. You can run libLEARNA on the example with:

```
liblearna --input_file
examples/cif_frog_foot_example_liblearna.input --timeout 30 --
num_solutions 10
```

3.2.7 Automated Design of Riboswitches

While libLEARNA allows the design of RNAs from single motifs like for inverse RNA folding, it further offers the opportunity to define more complex design spaces. To demonstrate such a more advanced use case of libLEARNA, we set up a partially restricted search space for the design of riboswitch-like constructs that we recently used to design massive amounts of theophylline riboswitch candidates***(28)***, following a previously published protocol***(37)***. We will guide through the definition of the design space step-by-step, using the protocol and the constructs proposed in ***(37)*** as templates. We then show how to define desired GC-contents as part of a task and compare the results with the original design procedure of ***(37)***.



[Figure 4] Design Space of libLEARNA for the design of theophylline riboswitch constructs. The constructs RS1, RS2, RS3, RS4, RS8 and RS10 were proposed by ***(37)***. Highlighted regions mark parts that are shared across all the original riboswitch constructs. These regions are used to construct a design space for libLEARNA. At the bottom, the figure shows an example prediction of libLEARNA for the given design space.

Originally, ***(37)*** construct riboswitch candidates from (1) the TCT8-4 theophylline aptamer sequence and structure, (2) a spacer sequence of 6 to 20 nucleotides (nt), (3) a sequence of 10nt to 21nt complementary to the 3′-end of the aptamer, and (4) a U-stretch of 8nt at the 3′-end of the construct. The total length of the riboswitch candidates within the given restrictions is 66 to 91 nucleotides. In the following, we will set up a design space consisting of a motif for each of the described parts, interleaved with regions for exploration. The resulting search space allows designing large amounts of candidates with a single run of libLEARNA. We show all proposed constructs of ***(37)*** (RS1-10), as well as the final design space that we use for the design of riboswitches and an example prediction in Figure 4. Highlighted regions mark parts that are shared across all constructs. We will use these shared motifs as positional constraints for the formulation of the design space.

We start with the definition of the aptamer motif. The sequence of the theophylline aptamer is fixed to ‘AAGUGAUACCAGCAUCGUCUUGAUGCCCUUGGCAGCACUUCA’ for all constructs

proposed by ***(37)***. We use this motif to define the sequence constraints. However, the structures show slight differences (see Figure 4, Aptamer column). To gather all information from the six constructs into a single structure representation, we use the shared regions across all constructs to define the positional structure constraints of the aptamer motif. All other positions are filled with 'N's, marking unconstrained positions in the structure space.

(37) use a variable-length spacer region of 6 to 20 nucleotides. The sequences and structures of the proposed constructs differ in their length and composition for this motif. However, we find a shared structure motif of four unpaired nucleotides ('....') across all structures with at least two positions upstream that are either paired or unpaired (see Spacer column in Figure 4). For the design space of libLEARNA, we use this shared motif to define the positional structure constraints of the spacer motif, while keeping the sequence of the motif unconstrained. We note again that all positions between the different motifs (Aptamer, Spacer, Complement, and U-stretch in Figure 4) are regions for exploration of the design space by libLEARNA, i.e., these regions can be filled with arbitrary sequence and structure parts. It is thus sufficient to define the spacer region with six nucleotides only, to ensure a minimum length of six.
We will define the length borders of the design space at the end, setting the *--min_length* and *--max_length* options to 66 and 91, respectively, according to the original setup in ***(37)***.

The third motif defines the terminator hairpin of the riboswitch. This region should pair with the 3'-end of the aptamer and ***(37)*** design this region by placing complementary nucleotides to the 3'-end of the aptamer, starting at different positions. Similar to the spacer region, the complementary regions can be of different lengths (see column Complement in Figure 4). The shared structure motif across the different constructs are ten consecutive closing brackets ('))))))))))'), which we use to define the structure part for libLEARNA. For the sequence part, however, we decide to not use the shared motif of complementary nucleotides to the 3'-end of the aptamer but keep this part unconstrained to allow for more exploration during the design. For example, note that the provided example prediction in

Figure 4 (bottom row) contains Wobble base pairs ('G–U'), which are not part of the design space of the original design procedure by ***(37)***.

For the last motif, the 8-U-stretch, all constructs share the same patterns in the sequence and the structure except for the first position of the structure part. We take on the positional constraints to define the design space for this motif and leave the first position of the structure part unconstrained. Our final search space for the design of theophylline riboswitch constructs with libLEARNA is then given as:

```
>theophylline riboswitch example
#seq AAGUGAUACCAGCAUCGUCUUGAUGCCCUUGGCAGCACUUCA NNNNNN UGAAGUGCUG
UUUUUUUU
#str ........NNN(((((.....)))))...NNN((((((((( NN.... ))))))))))
N.......
```

The input file for the riboswitch design task can be found in the examples directory (riboswitch_design_example.input).

Setting the length constraints from 66 to 91 nucleotides, we can run libLEARNA on the given input as follows:

```
liblearna --input_file examples/riboswitch_design_example.input --
timeout 300 --num_solutions 10 --min_length 66 --max_length 91
```

libLEARNA essentially samples a task from the search space at each iteration, seeking to solve it with a single shot. Consequently, the policy is optimized across all tasks that are encompassed in the provided design space. Depending on the search space definition, a single run of libLEARNA, therefore, might require some time. However, once libLEARNA found a good region of the search

space, it will sample massive amounts of different constructs of different lengths and sequence composition.

The ratio of G and C nucleotides can influence the function of an RNA drastically***(46)(47)***. libLEARNA, therefore, allows the design of RNAs with a desired GC content. The GC-content of the designed candidates can be adjusted with the *--desired_gc* and the *--gc_tolerance* options. The GC-content option can be used on top of each libLEARNA command described before. For example, to design riboswitch constructs with a GC content of 0.5, one can use the following command:

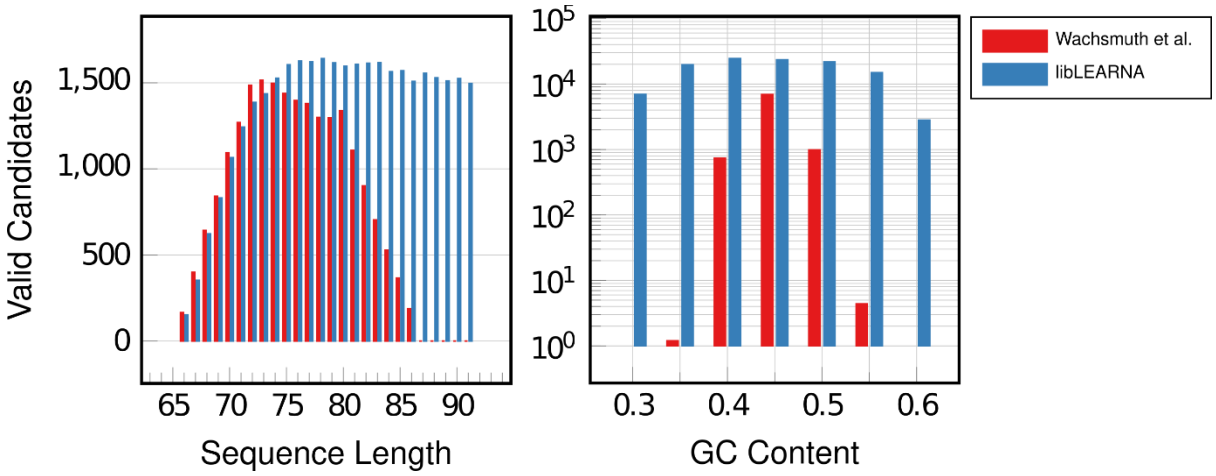
.

```
liblearna --input_file examples/riboswitch_design_example.input --
timeout 1000 --num_solutions 10 --min_length 66 --max_length 91 --
desired_gc 0.5 --gc_tolerance 0.01
```

We discuss the design with desired GC-contents in note 7.

However, the design space for theophylline riboswitch constructs is a complex task for libLEARNA. To compare against the original design procedure proposed by ***(37)***, we design a total of 50000 candidates with each approach using five different random seeds. We evaluate both approaches using the original evaluation scheme, described in ***(37)***. However, we skip the scoring of the candidates since we are not interested in the actual design but the generation of large amounts of candidates for potential wet-lab screens. The exact evaluation procedure is as follows: after dropping duplicates, the designed sequences are folded using RNAfold and verified for the existence of the two desired hairpins. No pairing is allowed within the last seven nucleotides of the 8-U-stretch as well as between the spacer and the aptamer in a sequence of folding steps, simulating a co-transcriptional folding process using a fixed elongation speed of five. The left plot in Figure 5 shows a comparison of the average number of valid candidates for each length. The distribution of candidates designed with libLEARNA appears nearly uniform across the different lengths, while the original protocol favors shorter solutions. In a second experiment, we additionally specify desired GC-contents for the design

of libLEARNA and again analyze the designed candidates. The results are shown in the right plot of Figure 5. libLEARNA generates much more valid candidates for each of the desired GC-contents compared to the original random procedure described in ***(37)***. Together, these results indicate that libLEARNA can be a valuable tool for large-scale design of candidates with desired properties. In particular, libLEARNA can be very useful in situations where fast screening methods of the generated candidates are available to determine the value of a given design. In such cases, libLEARNA can rapidly design a library of candidate sequences that can be experimentally screened for the desired features.



[Figure 5] Results of the design of theophylline riboswitch constructs using libLEARNA. We compare the designed candidates of libLEARNA with the original design procedure. The left plot shows the distribution of the number of designed candidates across different lengths. On the right, we plot results for the design of candidates with desired GC-contents. All plots show average numbers across five runs with different random seeds.

4 Notes

In this section, we discuss some aspects of the algorithms of the learna_tools package that might lead to undesired outputs, comment on some specifics of the individual programs, and provide further guidance on important aspects that users should consider when designing RNAs with the learna_tools package. We conclude this chapter with final remarks and an outlook to what we expect to be the future of machine learning for RNA design.

1. **Search Space Definition.** The definition of an appropriate search space is a crucial step when designing RNAs with libLEARNA. The concept of partial RNA design is very powerful, but also imposes challenges to the designer. For example, consider the problem of designing RNAs for the following two different target structures with a single design space formulation:

   `..((((...))))...` and `.((((...))))....`

   A joint RNA design space for libLEARNA could e.g., be defined as follows:

   ```
   >joint design space
   #seq NNNNNNNNNNNNNNNN
   #str .N(((N..N)))N...
   ```

   However, when running libLEARNA on this task, the output might not reflect the desired design goals. The reason is that the search space, besides the two target structures, also contains other structures. For example, '`..((((..))))....`', '`..(((....)))....`', and '`.((((((..))))))...`' are all valid structures for the provided space but do not match the two target structures.

   In such a case, we recommend running libLEARNA once on a multitarget input file:

   ```
   >target 1
   #seq NNNNNNNNNNNNNNNN
   #str ..((((...))))...
   >target 2
   #seq NNNNNNNNNNNNNNNN
   #str .((((...))))....
   ```

   With this approach, libLEARNA generates candidates for each target individually.

   In general, we recommend considering the size of the final solution space when defining a design space for libLEARNA. The riboswitch design task described in Section 3.2.7, for example, could be defined without any restrictions on the sequence and the structure:

   ```
   >unrestricted riboswitch design task
   ```

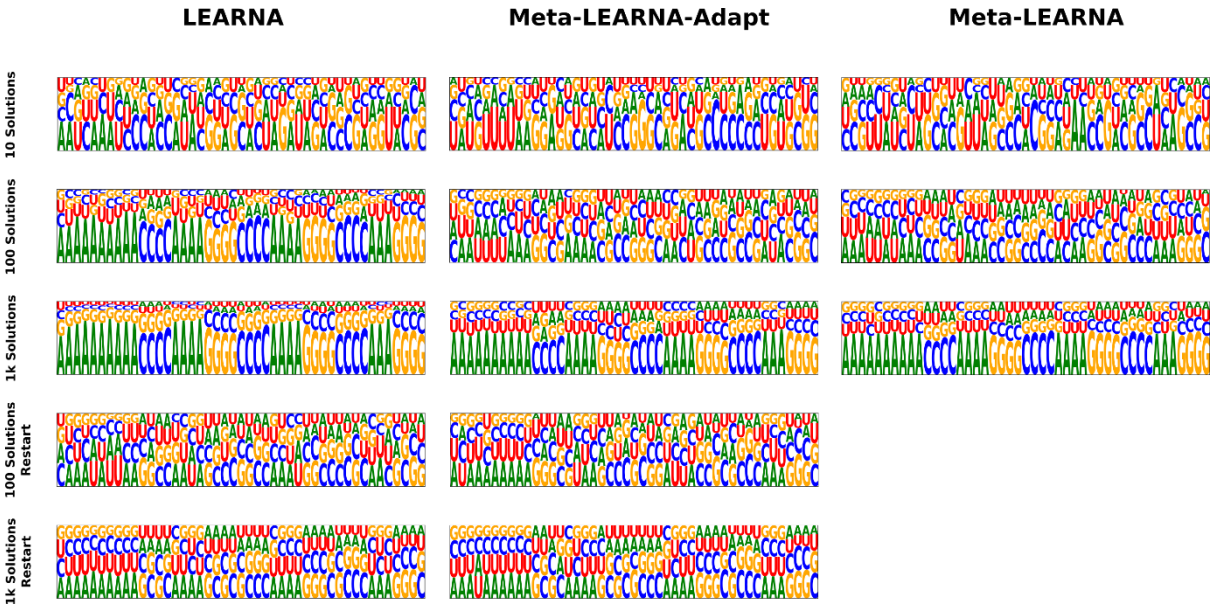
```
#seq X
#str X
```

However, while this search space includes all the riboswitch constructs when considering a length limit of 66 to 91 nucleotides, the space is likely defined too general to generate reasonable candidates. Similarly, for search spaces that only allow for very little exploration, libLEARNA might not always be the best choice and sometimes the definition of multiple tasks that are processed sequentially could be beneficial. Another critical aspect of the design space definition relates to the design of RNAs with a desired GC-content. Depending on the provided sequence constraints and the length restrictions of the design space, the requested GC-content might not be reachable at all. Without a provided timeout, libLEARNA would run infinitely long, trying to find solutions for the given task.

2. **Meta-Optimization Process**. For some users, it might also be interesting to run the automated reinforcement learning pipeline using BOHB. We provide the respective scripts in the github repository at https://github.com/automl/learna_tools. You can follow the instructions provided in the repository. However, we note that the optimization is a resource intensive process, and we recommend using a computer cluster.
3. **Plotting Secondary Structures.** If you experience any problems during plotting of the secondary structures with VARNA, please consider checking the following details. The learna_tools interface to VARNA expects an installation via conda. The call for VARNA might differ if it was installed without conda e.g. using the binaries provided with the .jar file at the official download section under https://varna.lisn.upsaclay.fr/index.php?lang=en&page=downloads&css=varna. If this is the case, we recommend to either change the call to VARNA in the source code (script learna_tools/visualization.py), or to install VARNA via conda. A second problem that we sometimes experience when using VARNA is that, depending on the structure and the desired resolution of the plots, VARNA might run out of memory. If you experience memory issues, consider changing the resolution for plotting with VARNA, or increase the available memory

for java on your system. You can adjust the resolution using the *--resolution <floating point number>* option available for all learna_tools algorithms. The default resolution is set to 8.0.

4. **Sequence Diversity.** We recall that all algorithms of the learna_tools package either optimized a policy across thousands of different RNA design tasks (Meta-LEARNA) or continue optimizing their design policies at test time to adapt to a given design task at hand (LEARNA, Meta-LEARNA-Adapt and libLEARNA). As a result, it is very likely that the algorithms converge to a good solution over time, resulting in similar sequences. For LEARNA, Meta-LEARNA-Adapt and libLEARNA, this clearly has the advantage that they can provide solutions even for tasks that are further away from the training data distribution, however, the diversity of sequences decreases over time. To mitigate this problem, all algorithms except Meta-LEARNA have a *--restart-timeout <natural number>* parameter that controls a reset of the algorithms. The restart parameter controls the time after which the weights of the algorithms are reset to the initial values, meaning the algorithms start from scratch again on the provided task. Figure 6 shows logo plots to visualize the distribution of nucleotides at a given position. We show results for the "Frog Foot" example from the Eterna100 benchmark mentioned before. In each row, the figure shows a run of either LEARNA, Meta-LEARNA, or Meta-LEARNA-Adapt with different numbers of solutions requested. At the bottom, we generate 100 and 1000 solutions with LEARNA and Meta-LEARNA-Adapt using the restart timeout option to reset the algorithms every 5 seconds (option not available for Meta-LEARNA). We observe that all algorithms converge to a similar major nucleotide pattern, mainly predicting 'C' and 'G' nucleotides for paired positions and 'A' for unpaired positions of the task, with increasing number of solutions requested. However, the restarting procedure rescues this behavior for both, LEARNA and Meta-LEARNA-Adapt, and the resulting logos show a high diversity, like the case when the algorithms did not yet adapt strongly to the target (when requesting 10 solutions only). However, this diversity comes at the cost of longer runtimes.



[Figure 6] Logo plots of LEARNA, Meta-LEARNA, and Meta-LEARNA-Adapt for different numbers of designed candidates for the Frog Foot example of the Eterna100 benchmark. Each column shows the logo plots for one of the algorithms for different numbers of requested solutions. The two bottom rows show the logo plots for LEARNA and Meta-LEARNA-Adapt when restarting the algorithms every 5 seconds for the design of 100 and 1000 (1k) candidates. While all algorithms seem to converge to a similar major design pattern, the restart option seems to help the algorithms to provide more diverse solutions.

5. **Flexible Structure Definitions.** libLEARNA implements the new RNA design paradigm, partial RNA design, which allows the user to provide arbitrary RNA sequence and structure motifs to define a design space. Typically, RNA design algorithms require an entire secondary structure as input. All pairing positions have to be defined in advance and the input is only valid if the numbers of opening and closing brackets are equal when using the dot-bracket notation. It is, therefore, noteworthy that there is no such restriction on the structure input for the motifs nor the entire structure in libLEARNA. In contrast, the designer can freely define the requirements of a given design task. For example, the following inputs are considered valid definitions of the sequence and structure dimensions of search spaces for libLEARNA:

```
>unbalanced example 1
```

```
#seq NNNNNNNNNN
#str NNNNNNN)))
>unbalanced example 2
#seq NNN NNN
#str NN( ))N
>unbalanced example 3
#seq XN
#str X)
```

Example 1 considers a design space of 10 nucleotides length that contains structures that end with three closing brackets (‘)))’). Example 2 defines a space from two motifs, while position 3 defines the opening position of a base pair. The provided candidates of libLEARNA will fold into structures that end with a ‘))N’ pattern, where ‘N’ can either be a dot or a closing bracket (since the structure cannot end with an opening bracket). Finally, example 3 is a very unrestricted search space where the only restriction is that the resulting candidates fold into structures that have a closing bracket at the last position.

As a result of these flexible task definitions, libLEARNA cannot take similar advantage of known base pairs as LEARNA, Meta-LEARNA, or Meta-LEARNA-Adapt. While these algorithms automatically place complementary pairing partners at positions that ought to be paired in the input structure (following a Watson-Crick base pair scheme), for libLEARNA the exact pairing positions might be unknown. However, libLEARNA leverages a similar approach in case the pairing partner is clearly distinguishable, i.e., there is no unrestricted structure position between the current opening bracket and the next closing bracket. One advantage of libLEARNA, therefore, is that it can generate sequences with non-Watson-Crick base-pairing patterns.

However, libLEARNA optionally allows to indicate matching pairs within the task description using a unique identifier, a natural number following directly after the respective

paired position to index the pair. As an example, the following task description is valid for libLEARNA:

```
>explicit pairing example
#seq NNN NNN
#str (1(N N)1N
```

The structure part defines a search space, where position 1 of the first motif and position 2 of the second motif should be paired. During the design, these explicit inputs are treated as soft constraints: The likelihood of the two positions to pair is increased by placing complementary Watson-Crick pairs at the explicit positions, however, the pairing is not enforced during the design process. It is also valid to provide multiple indices in a single design space using unique identifiers, or to leave some of the pairs without indices. We note that libLEARNA currently only supports the design for nested RNA structures. However, non-nested indexing will not result in an error but might not lead to the desired outcome.

Sometimes, the extension with arbitrary structure symbols might not be desired. For example, designing candidates with a variable length of 30 to 50 nucleotides that contain a single hairpin of five to ten nucleotides with a GNRA tetra-loop appears as a valid design goal. However, when indicating the regions for extension with whitespaces (or X), the resulting candidates might contain more than one hairpin. libLEARNA, therefore, allows to define regions for extensions with dots (‘.’) only. These regions can be defined with the letter ‘O’. An input file to the hairpin design approach described above could then e.g. look as follows:

```
>Single hairpin design example
#seq XNNNNNNNNNNGNRANNNNNNNNNNX
#str ONNNNN((((....))))NNNNNO
```

6. **Unique Features of libLEARNA.** Besides the definition of tasks, libLEARNA shows some differences compared to the other algorithms of the learna_tools package. It uses a completely new training regimen, inspired by a masked prediction task that was e.g. used to train the language model Bert***(49)***. While Meta-LEARNA and Meta-LEARNA-Adapt are purely trained on tasks of the inverse RNA folding problem, given only a structure in dot-bracket notation, the training tasks of libLEARNA are more diverse and might contain sequence constraints. Generally, libLEARNA employs the same data pipeline as the other two algorithms. However, in an additional preprocessing step, roughly two third of the training data points are masked such that sequence and structure constraints alternate, roughly 20 percent are masked at random and the remaining samples are used as inverse RNA folding tasks. Table 1 shows examples for each of these classes of training samples. The general task during training of libLEARNA can be interpreted as filling the missing parts of the sequence and the structure input.

[Table 1] Examples of different tasks of the training data of libLEARNA.

Task Description	**Task Space**	**Example**	**Percentage of Data**
Inverse RNA Folding	Sequence	`NNNNNNNNNNNN`	11,5
	Structure	`...(((...)))`	
Alternating Constraints	Sequence	`NNAUGNNNCCNN`	66,7
	Structure	`..NNN(..NN))`	
Random Masking	Sequence	`NNANNGANNNNA`	21,8
	Structure	`N..NN(..NNN)`	

The inclusion of sequence constraints also affects the state composition of libLEARNA. In contrast to LEARNA, Meta-LEARNA, and Meta-LEARNA-Adapt, which only use the structure input for providing states to the agent, libLEARNA uses the structure and the sequence inputs to define the states. The state space of libLEARNA, thus, is much larger but

also allows for more flexible task definitions. The sliding window approach, however, remains the same as for the other methods.

As a result of the changes, the policy learned by libLEARNA seems to be less biased towards specific nucleotide compositions. There are several reasons that might explain this observation. Firstly, the large state space of libLEARNA makes it harder to learn specific patterns for a given state. Also, the same state is less likely to appear more than once during training due to the random masking procedure and the additional sequence space. Secondly, the masking leads to more ambiguous data and a broader solution space, making it less attractive to optimize for a single solution for a given task. Finally, the masking procedure often enforces the prediction of single nucleotides rather than Watson-Crick base pairs at a given paired position. The algorithm thus must be able to find solutions with predictions of single nucleotides only. This leads to general differences in the learning procedure, reduces biases and might lead to an increased base pairing repertoire, e.g., libLEARNA can predict solutions with Wobble-pairs ('G-U') which the other algorithms cannot.

7. **RNA Design with Desired GC-Contents.** When designing RNAs with desired GC-contents using libLEARNA, we would like to note that libLEARNA was not trained to generate sequences with certain GC-contents. The algorithm optimizes the design for the tasks contained in a given design space to maximize its reward. The GC-content is a term in the reward function of libLEARNA that is set on top of the structure-based reward score that it was trained on. However, libLEARNA is quite sensitive to this change in the reward function and adapts quickly to the design of sequences with a specific GC content. Nevertheless, depending on the search space, the adaptation might require more time than running libLEARNA without GC-content control.

4.1 Concluding Remarks

The learna_tools Python package provides advanced machine learning algorithms for the design of RNAs. In particular, the most recent algorithm, libLEARNA, hits novel ground with its ability to design RNAs with desired properties from provided motifs. We will continue the development of libLEARNA to exploit its full potential in the future. However, machine learning methods, and in particular deep learning approaches, for biological applications recently got into the focus of the deep learning community. We expect this development to continue and anticipate that several deep learning methods for biological applications will be developed in the future, including methods related to the field of RNA structural biology. The first steps in this direction can already be seen today. For example, recent work uses deep learning to design RNAs including 3D structure information***(50)(51)***, an interesting approach that will likely get off the ground with an ever growing number of 3D RNA structure data being available in the future. Further, the development of methods that deal with structural biology problems across different molecule classes***(52)***, including proteins, RNAs, or small organic compounds, could have the potential to revolutionize structure prediction and design in general. In this context, generative methods based on transformers***(53)*** and diffusion models***(54)*** could play a key role. We believe that deep learning has the potential to have a lasting impact on the field of biology and are therefore looking to the future with excitement.

5 Acknowledgements

This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant number 417962828. The authors further acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG.

6 References

[1] Jumper J et al (2021) Highly accurate protein structure prediction with AlphaFold. *Nature* 596(7873):583-589.

[2] Lin Z et al (2023) Evolutionary-scale prediction of atomic-level protein structure with a language model. *Science* 379(6637):1123-1130.

[3] Watson JL et al (2023) De novo design of protein structure and function with RFdiffusion. *Nature* 620(7976):1089-1100.

[4] LeCun Y, Bengio Y, Hinton GE (2015) Deep learning. *Nature* 521(7553):436-444.

[5] Hornik, K, Stinchcombe M, White H (1989) Multilayer feedforward networks are universal approximators. *Neural networks* 2.5:359-366.

[6] Cybenko, G (1989) Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* 2.4:303-314.

[7] Silver, D et al (2016) Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587):484-489.

[8] Vinyals, O et al (2019) Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575(7782):350-354.

[9] Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25.

[10] Singh J et al (2019) RNA secondary structure prediction using an ensemble of two-dimensional deep neural networks and transfer learning. *Nature communications* 10.1:5407.

[11] Sato K, Akiyama M, Sakakibara Y (2021) RNA secondary structure prediction using deep learning with thermodynamic integration. *Nature communications* 12.1:941.

[12] Franke JKH, Runge F, Hutter F (2022) Probabilistic Transformer: Modelling Ambiguities and Distributions for RNA Folding and Molecule Design. *Advances in Neural Information Processing Systems* 35:26856-26873.

[13] Townshend RJL et al (2021) Geometric deep learning of RNA structure. *Science* 373(6558):1047-1051.

[14] Valeri JA et al (2020) Sequence-to-function deep learning frameworks for engineered riboregulators. *Nature communications* 11.1:5058.

[15] Tuerk C, Gold L (1990) Systematic evolution of ligands by exponential enrichment: RNA ligands to bacteriophage T4 DNA polymerase. *science* 249(4968):505-510.

[16] Im J, Park B, Han K (2019) A generative model for constructing nucleic acid sequences binding to a protein. *BMC genomics* 20.13:1-13.

[17] Iwano N et al (2022) Generative aptamer discovery using RaptGen. *Nature Computational Science* 2.6:378-386.

[18] Di Gioacchino A et al (2022) Generative and interpretable machine learning for aptamer design and analysis of in vitro sequence selection. *PLoS computational biology* 18.9:e1010561.

[19] Andress C et al (2023) DAPTEV: Deep aptamer evolutionary modelling for COVID-19 drug design. *PLOS Computational Biology* 19.7:e1010774.

[20] Riley AT, Robson JM, Green AA (2023) Generative and predictive neural networks for the design of functional RNA molecules. *bioRxiv* doi: https://doi.org/10.1101/2023.07.14.549043.

[21] Goodfellow I et al (2014) Generative adversarial nets. *Advances in neural information processing systems* 27.

[22] Green AA et al (2014) Toehold switches: de-novo-designed regulators of gene expression. *Cell* 159.4:925-939.

[23] Shi J, Das R, Pande VS (2018) SentRNA: Improving computational RNA design by incorporating a prior of human design strategies. *arXiv preprint arXiv:1803.03146*.

[24] Koodli RV et al (2019) EternaBrain: Automated RNA design through move sets and strategies from an Internet-scale RNA videogame. *PLoS computational biology* 15.6:e1007059.

[25] Lee J et al (2014) RNA design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences* 111.6:2122-2127.

[26] Eastman P et al (2018) Solving the RNA design problem with reinforcement learning. *PLoS computational biology* 14.6:e1006176.

[27] Runge F et al (2018) Learning to Design RNA. *International Conference on Learning Representations*.

[28] Runge F, Franke JKH, Hutter F (2023). Towards Automated Design of Riboswitches. The 2023 ICML Workshop on Computational Biology.

[29] Sutton RS, Barto AG (2018) Reinforcement learning: An introduction. MIT press, 2018.

[30] Schulman J et al (2017) Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[31] Henderson P et al (2018) Deep reinforcement learning that matters. *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. No. 1. 2018.

[32] Parker-Holder J et al (2022) Automated reinforcement learning (autorl): A survey and open problems. *Journal of Artificial Intelligence Research* 74:517-568.

[33] Hutter F, Kotthoff L, Vanschoren J (2019). *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[34] Hofacker IL et al (1994) Schnelle Faltung und Vergleich von Sekundärstrukturen von RNA. *Monatshefte für Chemie/Chemical Monthly* 125:167-188.

[35] Hamming RW (1950) Error detecting and error correcting codes. *The Bell system technical journal* 29.2:147-160.

[36] Runge F et al (2023). Partial RNA Design. *bioRxiv*, 2023-12, doi: https://doi.org/10.1101/2023.12.29.573656.

[37] Wachsmuth M et al (2013) De novo design of a synthetic riboswitch that regulates transcription termination. *Nucleic acids research* 41.4:2541-2551.

[38] Li M et al (2018) In vivo production of RNA nanostructures via programmed folding of single-stranded RNAs. *Nature communications* 9.1:2196.

[39] Nozawa Y et al (2019) Rational design of an orthogonal pair of bimolecular RNase P ribozymes through heterologous assembly of their modular domains. *Biology* 8.3:65.

[40] Mironov AS et al (2002) Sensing small molecules by nascent RNA: a mechanism to control transcription in bacteria. *cell* 111.5:747-756.

[41] Shine J, Dalgarno L (1975) Determinant of cistron specificity in bacterial ribosomes. *Nature* 254(5495):34-38.

[42] Falkner S, Klein A, Hutter F (2018) BOHB: Robust and efficient hyperparameter optimization at scale. *International conference on machine learning*. PMLR, 2018.

[43] Darty K, Denise A, Ponty Y (2009) VARNA: Interactive drawing and editing of the RNA secondary structure. *Bioinformatics* 25.15:1974.

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

[45] Tareen A, Kinney JB (2020) Logomaker: beautiful sequence logos in Python. *Bioinformatics* 36.7:2272-2274.

[46] Isaacs FJ, Dwyer DJ, Collins JJ (2006) RNA synthetic biology. *Nature biotechnology* 24.5:545-554.

[47] Chan CY et al (2009) A structural interpretation of the effect of GC-content on efficiency of RNA interference. *BMC bioinformatics* 10:1-7.

[48] Koodli RV et al (2021) Redesigning the EteRNA100 for the Vienna 2 folding engine. *BioRxiv* (2021): 2021-08 doi: https://doi.org/10.1101/2021.08.26.457839.

[49] Devlin J et al (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[50] Yesselman JD et al (2019) Computational design of three-dimensional RNA structure and function. *Nature nanotechnology* 14.9:866-873.

[51] Joshi CK et al (2023) Multi-State RNA Design with Geometric Multi-Graph Neural Networks. *arXiv preprint arXiv:2305.14749*.

[52] GoogleDeepMind (2023) Deep Mind Blog. https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/a-glimpse-of-the-next-generation-of-alphafold/alphafold_latest_oct2023.pdf. Accessed 21 Dec 2023

[53] Vaswani A et al (2017) Attention is all you need. *Advances in neural information processing systems* 30.

[54] Yang L et al (2023) Diffusion models: A comprehensive survey of methods and applications. *ACM Computing Surveys* 56.4:1-39.