EteRNA-RL: Using reinforcement learning to design RNA secondary structures

Isaac Kauvar [1] **Ethan Richman** [1] **William E Allen** [1]

Abstract

We use reinforcement learning to target the task of designing ribonucleic acid (RNA) sequences that fold into target multi-dimensional structures, which has broad applications in biology. Although we do not successfully solve the complete task, we demonstrate that it is possible to train an agent that gains some level of 'intuition' about the problem.

1. Introduction

Because ribonucleic acid (RNA) is single stranded, it can fold upon itself to generate functional structures similar to proteins. Much work has been done to predict secondary structures given an RNA sequence (Eddy, 2004); however, the inverse problem of selecting a sequence that generates a target structure is not efficiently solved. Applications of RNA sequence design include RNA-guided silencing, genome editing, and protein organization (Anderson-Lee et al., 2016; Lee et al., 2014). In this paper, we investigate using reinforcement learning to solve the problem of selecting a ribonucleic acid (RNA) sequence that folds into a target multi-dimensional structure, as illustrated in Fig. 1.

We were inspired by EteRNA, a computer game which provides a black-box secondary structure predictor and a user interface for mutating bases in a sequence. EteRNA was developed as a web game in order to crowdsource human intuition, pattern recognition, and puzzle-solving skills. Gameplay consists of performing a sequence of actions: at each time step, the user can switch the nucleotide of a base in the sequence. Reward is indicated by how likely the current sequence is to fold correctly into the target structure. A screenshot of the game interface is provided in Fig. 2. Underlying EteRNA is an assumption (or at least, hypothesis) that expert human game-players can develop intuition that enables better and faster sequence design. The EteRNA project additionally added a wet-lab in-the-loop aspect which checked the physical accuracy of gameplayer derived models. For now, we ignore the wet-lab aspect and focus on the question of whether we can use reinforcement learning to train a computer agent to play an EteRNA-esque game. We aim to develop an algorithm capable of meeting and surpassing human level puzzle solving. If we can train a reinforcement learning agent to acquire human level 'intuition', then it may ultimately be useful in discovering or predicting new properties of RNA folding that can be tested in wet-lab experiments.

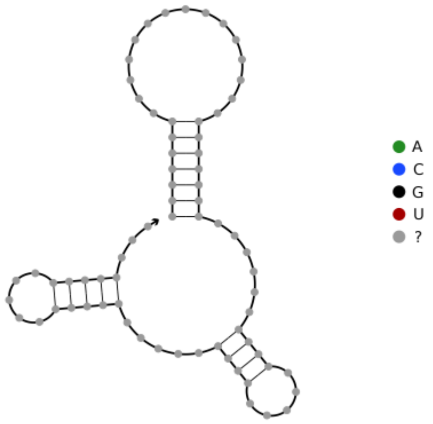

Figure 1. Example RNA secondary structure. The goal is to assign a color to each base so that the specified target structure is the structure most likely to results from RNA folding.

Rather than using EteRNA's secondary structure predictor, we use a predictor provided by the Nucleic Acid Package (NUPACK) (Zadeh et al., 2011). The API provided by NUPACK offers a function to compute the ensemble defect of a given sequence and target structure. The ensemble defect is the average number of incorrectly paired nucleotides at equilibrium over the ensemble of possible secondary structures. Minimizing ensemble defect means that the RNA sequence is more likely to spend time folded into the target structure. The ensemble defect calculation in NUPACK uses thermodynamics-based computation. Although there are limitations inherent to the ensemble defect computation, improvements in the forward model based on wet-lab experimentation can in theory be incorporated into what-

[*]Equal contribution [1]Stanford University, Stanford, CA. Correspondence to: Kauvar I <ikauvar@stanford.edu>.

Stanford CS 234 Final Project, 2017.

ever algorithm we design, simply by modifying the reward function.

Because there are four possible nucleotides for each base, the combinatorial number of options for a sequence of length n is 4^n. For $n = 30$, this is $1e18$ combinations. For $n = 100$, this is $1e60$ combinations. Because of the exponential nature of the problem, it quickly becomes impossible to either enumerate across all possible combinations, or to tabulate the score associated with each combination. Finding the target sequence is made easier by taking a proper sequence of steps - it is sometimes better to be in one state than another state when selecting the next step. Because this game offers a continuously computable score, there are a number of optimization approaches that could apply. However, because the action space is discrete, and humans appear to be effective at solving the game by making a sequence of action decisions, reinforcement learning is potentially a good algorithm to apply. Further, gradient based methods do not necessarily make sense in this setting, and an algorithm, such as a reinforcement learning agent, that can incorporate some intuition about how RNA folds may perform better than a generic global optimization algorithm.

One essential question of our investigation is whether reinforcement learning can train an agent to acquire human-like intuition. Can it learn heuristics that human EteRNA game-players have learned? One such example is 'boosting' loops, which is the use of certain nucleotide combinations at the first unpaired positions of a loop.

We reiterate that it is not clear to us whether a reinforcement learning based approach will actually work better than a generic evolutionary based approach.

It is also worth noting that we are essentially generating a framework to solve the general problem of coloring a graph according to a black-box function that defines the quality of the coloring.

Pair up three more As (yellow) with Us (blue), making 9 pairs total.

Figure 2. Screenshot of EteRNA computer game in which humans can learn and then utilize intuition about how to produce good colorings.

2. Methods

2.1. Problem setup

We will solve the problem for a fixed-sized graph of n elements, where each node is labeled with a color. The graph represents an RNA sequence, where the color of each node represents the nucleotide of that base, and the links in the graph represent which nodes are bonded to one another, as seen in Fig. 3. Even with a fixed size graph as input, we can use the same learned parameters on graphs of fewer elements by simply labeling the extra nodes with a color indicating that they are to be ignored (i.e. setting their value to 0).

State: (adjacency matrix, color vector), with $A \in \{0,1\}^{n(n-1)/2}$ and $C \in \{0,1,2,3,4\}^n$. The ordering of the color vector matters, and indicates the sequence of bases. The adjacency matrix only indicates paired bases, and is vectorized so that the first elements correspond to the pairing of the first base with other bases.

Action: (i, x), indicates change node i to new color x. We enumerate the possible (i, x) pairs by assigning each one a unique integer identifier. We define the deterministic state-action transition function to be

$$f(s, a) : \text{set } s[i] = x \in \{1, 2, 3, 4\} \tag{1}$$

Objective: Minimize ensemble defect.

Reward: Gain 1 point for decreasing ensemble defect by 1. (We will store the ensemble defect of the previous state and then compute the difference). The sum reward at the end is thus just equivalent to the total improvement in the ensemble defect through the episode.

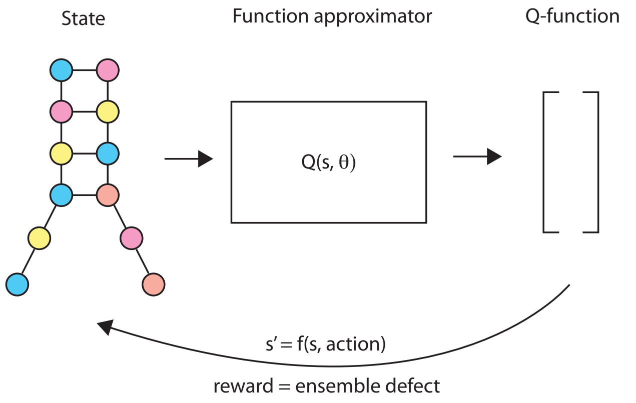

Figure 3. Problem setup.

Our problem is one with a known and deterministic, though non-differentiable, transition model. The transition model, specifically, is that if the specified action is to change the color of a specific node, the game will always change that color with a probability of 1. The reward function, however, is unknown and non-differentiable, but can be simulated with a forward model.

2.2. Algorithm

In this current work, we focus on implementing a value function approximating Deep-Q Network (DQN). Future work would likely benefit significantly from also exploring the efficacy of policy gradient approaches. In the results section we also compare our RL agent to a generic global optimization algorithm that directly optimizes the ensemble defect.

We use the DQN algorithm with memory replay presented in (Mnih et al., 2015). We use temporal difference (TD) learning based gradient updates to approximate the state-action value function Q. We used stochastic gradient descent with Adam update steps.

Algorithm 1 DQN

```
Initialize Q-function parameters $w$
for iter = 1, 2, ... do
    Initialize replay buffer
    for episode = 1, 2, ... do
        Roll-out episode according to current $Q$
        For each step, append (s, a, s', r) to replay buffer
    end for
    for episode = 1, 2, ... do
        Sample (s, a, s', r) from replay buffer
        $L = r + \gamma \max_{a'} Q(s', a', w^{-}) - Q(s, a, w)$
        $\Delta w = \alpha(L) \nabla_w Q(s, w)$
        Update $w$ according to $\Delta w$
    end for
end for
```

3. Implementation

We implemented our Q-function approximator as deep network in tensorflow. We used fully connected as opposed to convolutional networks because there is no obvious space invariance in the state of our environment. We tried a number of different network architectures, although in our experimentation it did not ever become obvious which hyperparameters were clearly better. For the simplest test cases, a linear network or a two-layer network with a ReLu activation actually performed quite well. For the more difficult cases, we instead trained primarily a 4 layer network with 80, 100, 100, and 100 units per layer and ReLu activations. We initially used a learning rate that decreased on a linear schedule from 0.0025 to 0.001. We found that convergence seemed to significantly slow down once the learning rate was at it its minimum value, so we ran some additional experiments with various learning rate schedules, but in the end this was not the fundamental problem with our initial experiments (our definition of the reward function was likely the primary issue). We also used epsilon, the parameter for epsilon-greedy exploration, that decreased from 1 to 0.01 across half of the number of training steps. Training time was slow - primarily, we believe because of the way that we were calling the defect ensemble computation subroutine from the NuPACK package. Unfortunately, the python bindings to NUPACK called the package through the command line, which incurs a decent amount of overhead for each call. It was not obvious to us how to streamline the interface, although we believe this would make a big difference.

4. Results

4.1. Simplest test case: 2 colors, 3 nodes, 1 target structure, simple reward function

We began by training a linear Q-function approximation using Q-learning with a single fixed target sequence of length $n = 3$. Here, we use a simple environment for initial testing, with a prescribed reward function. This simplest test environment accepts sequences of length 3, and is initialized with a target coloring. In this case, positive reward (i.e. +1) is gained only when the state matches the target sequence, and a small negative reward (i.e. -0.01) is incurred for every action taken before reaching the target. This is not as general as the case where reward is measured based on ensemble defect, and training the weights of a policy or value function network only apply to the specific target sequence. However, the agent should be able to reach the target sequence from any starting sequence, and ideally should reach it in the fewest number of steps possible. From the starting sequence coloring [1, 1, 1], only two actions were required to reach the target coloring [0, 1, 0]. Within a few thousand iterations, the Q-function converged and the algorithm consistently achieved the target in the minimum of two actions, as seen in Fig. 4.

4.2. Simplest interesting test case: 4 colors, 3 nodes, 5 target structures, simple reward function

We next tested a more difficult scenario - a sequence of length 3, but now with 4 possible colors for each node, and also 5 potential target structures. The linear network did not work on this more complicated task, and so we increased the complexity of the function approximator to a 2-layer fully connected network with 20 units in each layer and ReLu activations.

We note that in order to achieve convergence for this test case, it was necessary for there to be multiple color sequences that worked for each adjacency matrix. In this

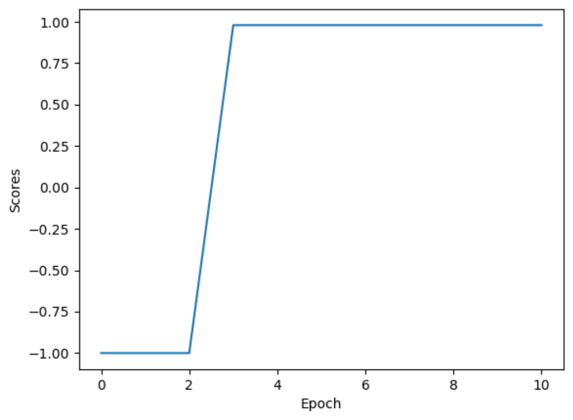

Figure 4. Test environment with linear DQN and fixed target (each epoch is 1000 iterations).

case, for each of the 5 target structures, we defined 10 target color sequences. This was interesting because it demonstrated the importance of the the sparsity level of the reward function. With 4 possible colors and even just 3 nodes, we found that it was difficult during training for the algorithm to find the rewarded states frequently enough. As seen in Fig. 5, however, with enough target sequences, we were able to achieve convergence. Further, as shown in Fig. 6, our agent successfully learned to reach the target coloring in the fewest possible number of actions. A random policy, on the other hand, took significantly more actions to reach the target.

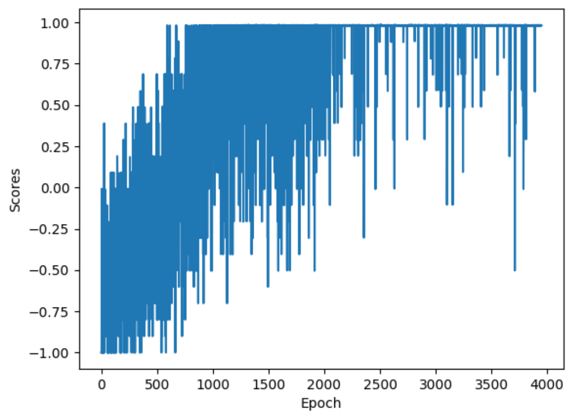

Figure 5. Convergence for training a 2-layer network on 5 target structures with 4 color options and sequences of length 3.

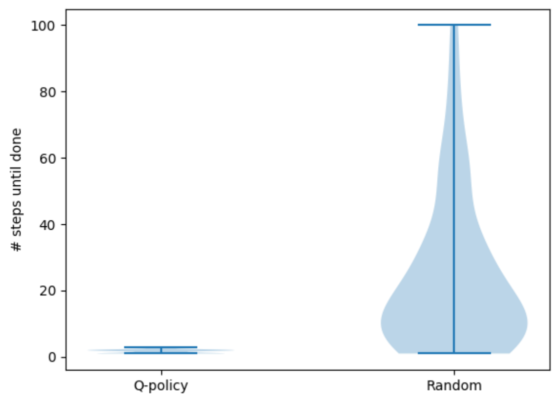

Figure 6. Performance comparison of the trained policy vs. a random policy. The trained policy is able to nearly always successfully reach the target sequence for a given structure in the fewest number of steps possible, whereas the random policy does not.

4.3. Full game: 4 colors, 20 nodes, 12 target structures, defect-ensemble based reward

Given the successful performance of our algorithm on the test cases, we moved on to testing it on the full game. We defined 12 target structures that spanned a spectrum of possible folded structures. Specifically, we focused on hairpin RNA structures with different length bonded and unbonded regions. When resetting the environment, we did not initialize the RNA sequence in a fully random manner, but rather made sure that all paired bases followed Watson-Crick pairing rules. From the outset, the initialized sequence thus performed fairly well, and the goal of the trained agent is to see how much further it could improve the performance. We used a fixed maximum sequence length, but designed the state so that it could account for sequences of a length less than the maximum length. In particular, for any nodes beyond the length of the target sequence, the color was set to zero (whereas for the active nodes the color was in $1, 2, 3, 4$. Further, we defined the state-action transition function such that for nodes whose color was 0, no action could change the color of that node. Further, no action could change the color of a non-zero node to zero. The hope was that the algorithm would learn that for an input state with nodes colored zero, it was futile to take any actions on those nodes.

The biggest decision we had to make was how to define the reward function as a function of the normalized ensemble defect. We note that some colorings for a given structure are 'invalid', and we ascribed these a reward value of -1.0. The best possible reward value is 0.0, for a coloring that is likely to fold into the target structure with a probability of one. We decided that we should return as the reward the defect ensemble at every update step. To us, it seemed as if it would be best to provide the algorithm with the most information possible about each of its actions. With this environmental setup, we used a 4 layer fully-connected network, with 40, 100, and 80 units in the hidden layers, and an output layer corresponding to the 80 possible actions.

Training this algorithm was quite slow because of the calls to the defect ensemble subroutine as described above. However, we kept at the training because it appeared to showing exponentially increasing convergence towards a higher average. The convergence eventually sharply plateaued, as seen in Fig. 7. This occurred around the same time that the learning rate and epsilon reached their minimum value. Before trying again with different hyperparameter schedules, however, we noticed that the policy returned by this algorithm was not quite doing what we wanted it to. In particular, the policy seemed to be choosing actions that either did not change the sequence or that only changed it once or twice throughout all of the steps of an episode. This is evident in Fig. 8, where we see that for the learned policy, the majority of episodes had scores that were extremely similar to the initial score (whereas for the random policy, most of the final scores were much worse than the initial score). We realized that this effect is likely do to the specific of how we were defining the reward. In particular, to reach a potentially better state than the initial state, the agent would likely have to take actions that would pass through some bad states, for example invalid states which would yield a defect ensemble of -1.0. Because the agent computes the expected discounted sum of future rewards, the cost of ending up in a bad state for even a few steps far outweighed the potential increase in reward from, say -0.2 to -0.15. That is, staying at the state with reward -0.2 for the full episode of ten steps would be better than making one major mistake even if the agent ends up at a slightly better state.

There are a couple of potential solutions to this problem: 1) Defining the reward slightly differently: we could return the defect ensemble only every few steps, returning a reward of 0 during the intervening steps. Or, we could only return the reward at the end of the episode. The latter would probably be the best option, however based on our initial experiments on the simple test environments, this would likely be too sparse of a reward for training. 2) In the current experiment, we capped each episode to a maximum of ten steps. We did this both to increase the speed of training, as well as to see if we could successfully train the agent to achieve good scores in only a few steps. If, however, we increased this maximum score, the agent may be able to reap more gain from moving to slightly more rewarding states, if that reward was enjoyed over more steps.

Due to the long training of this initial setup, we did not, however, have time to try many additional experiments. We thus decided to focus on two specific questions. On the one hand, how well could a non-reinforcement learning based algorithm perform at this task? And on the other hand, rather than trying again to train an agent on the full task, could we train an agent that simply learned the distinction valid vs. invalid structures? We decided to take this approach because it also seemed like first pre-training a network to 'understand' what makes a *good* sequence could help when it is trying to take a sequence of steps to achieve the *best* sequence.

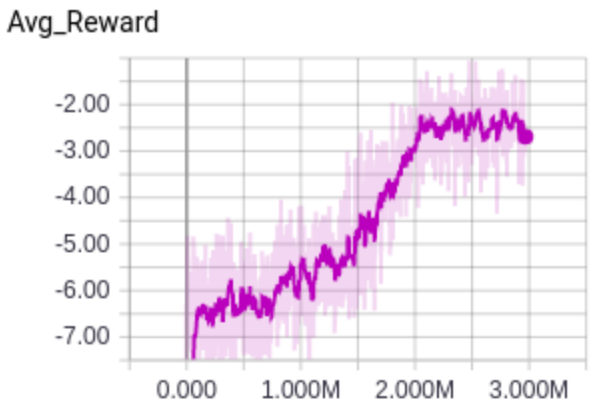

Figure 7. Convergence of training a 4 layer fully-connected network on 12 target structures using the ensemble defect at each step as the reward. The state was initialized to a fairly good, though not optimal, coloring.

4.4. Training an agent that can produce valid sequences: 4 colors, 20 nodes, defect-ensemble based reward

With the insight from the previous experiment that we likely defined our reward function incorrectly, we decided to take a step back and investigate a simpler scenario. In particular, we wanted to see whether we could train an agent that was capable of learning *any* of the structure of the game. We thus focused on whether we could train an agent to convert an invalid coloring into a valid coloring for a specified target structure. Instead of initializing the coloring to a reasonably good state, we selected an initialization coloring that was invalid (and thus had a reward of -1.0) and that was specifically chosen to be one mutation away from being a valid coloring (that is, changing the color of just minimum one node would yield a valid coloring). We also let the agent play for only one step per episode. The goal was thus that the agent should learn 'what makes a valid structure', and learn how to convert an invalid structure into a valid one. We trained the same 4 layer fully-connected

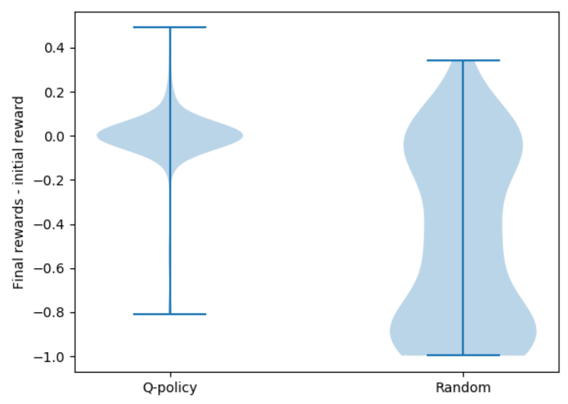

Figure 8. Final reward minus initial reward across 100 games with either the learned policy or a random policy. These results demonstrate that there was a subtle error in the way that we defined the reward function, which led the agent to learn that it was better off selecting actions that did not change the current coloring as opposed to risking the punishment associated with moving through a bad state. The random policy, on the other hand, moves into a bad state with high frequency.

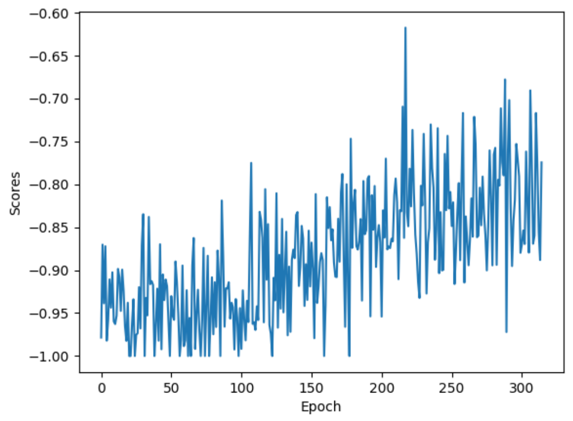

Figure 9. Convergence (average evaluation score) during training of a 4-layer Q-network which is allowed to take one action per trial with the goal of turning an invalid structure into a valid structure.

network architecture as in the previous experiment.

The result, as seen in Fig. 10 (with convergence plotted in 9), is that our network was indeed able to successfully learn some of the structure of the environment. In comparison to a random policy, which would select a proper action some small probability of the time, the learned policy selected an action which led to a valid structure on a large proportion of the test trials.

We note that training a network on this task may serve as good pre-training before moving onto the more challenging situation of taking many steps to improve an already reasonable coloring.

4.5. Direct ensemble defect optimization with evolutionary algorithm: 4 colors, 20 nodes, defect-ensemble based reward

The final comparison that we wanted to make was with the performance of a generic optimization algorithm that did not learn about any of the structure of the environment. Whereas we were trying to train a task-specific agent using reinforcement learning that was able to harness ‘intuition’ about the environment in order to efficiently improve the score, algorithms blind to the specific task structure have the potential to work well.

For this, we turned to a non-gradient based global optimizer using the differential evolution algorithm and implemented as part of scipy. We directly optimized the defect ensemble. We ran the optimization for ten different target structures, initializing to a valid coloring, and running 100 iterations. As seen in Fig. 11, where each line represents the change in ensemble defect from the initial state to the final optimized state. Whereas the initial defect had a mean value of 0.05 (standard deviation 0.03), the final defect had a mean value of 0.004 (standard deviation 0.02). Thus, in just 100 iterations, this generic global optimization algorithm was able to, for a wide variety of different target structures, perform extremely well.

It is likely, however, that for larger sequence lengths, this would run into more issues as the dimensionality of the problem becomes larger. In particular, a significantly larger number of iteration would likely be required, and each iteration step would also take longer.

5. Discussion

Here, we began development of a reinforcement learning based algorithm for optimizing the design of an RNA sequence that folds into a target secondary structure. We were inspired to apply reinforcement learning to this problem because of the internet-based game EteRNA, in which humans can gain intuition through trial and error about tropes and strategies that yield high performing sequences for a given target structure. We hypothesized that a reinforcement learning agent would be able to similarly acquire ‘intuition’ about what defines a good sequence for a given structure.

It is worth noting that the true impact of human intuition

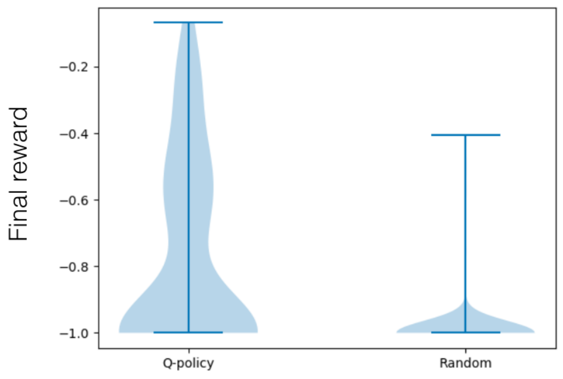

Figure 10. The distribution of final scores for 100 test trials using either the learned Q-network based policy or a random policy, in the case when the state is initialized to an invalid coloring, and the agent is allowed to take only one action to convert the state into a valid coloring. Whereas a random policy only takes the correct action a small percentage of trials, the learned Q-policy demonstrates that it has learned about the structure of the environment such that it is capable of taking a correct action on a large proportion of trials.

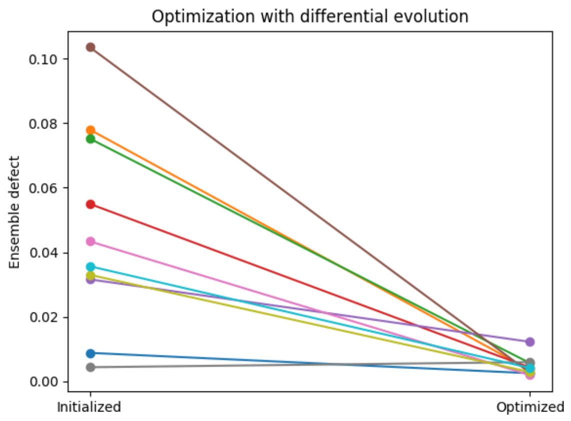

Figure 11. Improvement in defect ensemble from direct optimization with an evolutionary algorithm, for 100 steps and for 10 target structures. Each line represents the improvement in ensemble defect from the initialized coloring to the optimized coloring for a different target structure.

in the EteRNA games is relevant when there is a mismatch between the simulated forward model of how RNA sequences fold, and the wet-lab experiments that demonstrate how a given RNA sequence *actually* folds. The hope with EteRNA was that human players would be able to synthesize information both from game-play and from wet-lab experiments that together would produce better models of RNA folding. We hoped to develop an agent that could perform the first part: gaining intuition about game-play. With that in hand, you would hopefully then be able to train the agent based on data from wet-lab experiments.

We demonstrate that our algorithm is successful in a number of simpler cases. In these simpler test cases, we found that learning was unlikely to converge if rewards were too sparse. In the process of trying to train an agent on more complicated cases, we also discovered the overwhelming importance of the precise definition of the reward function. By defining a reward function in a manner that made the punishment for exploration through bad states too large relative to the potential gain in ultimately reaching good states, our agent learned that it was better to just stay where it was rather than explore. Potential improvements in future work could include: adjusting the reward function to enable ‘exploratory excursions’ without impact the reward, and adding an optimistic exploration bonus. Additionally, pre-training on simpler cases so as to teach the agent the general principles of the environment will likely be beneficial. Further, using a policy gradient approach instead of a Q-learning approach will likely yield improvements, since the action space of our environment is relatively large.

For the relatively short sequences that we investigated here, we found that direct optimization of the ensemble defect using an evolutionary global optimization algorithm was was actually quite effective. We hypothesize that for larger sequences, however, this generic approach will become less effective, and there will be more room for improvement offered by a learned optimization algorithm which possesses some knowledge of the structure of the environment.

Finally, it is worth discussing, in retrospect, whether the game we chose to solve is in fact a good application of reinforcement learning. There a number of characteristics that make this application play less to the strengths of reinforcement learning: the state-action transition function is deterministic, only the final sequence coloring is important as opposed to the action sequence used to reach that coloring (even if it may be possible to more efficiently reach that target coloring by taking a specific sequence of actions), the environment offers a black-box computation of the quality of each state, and every state is reachable from every other state. For these reasons, it is not *impossible* to apply conventional global optimization algorithms. However, where reinforcement learning may offer an advantage is in developing a non-gradient based optimization approach which incorporates intuition about which sequences of actions will yield the largest future expected discounted sum of rewards.

In sum, we demonstrate that it is possible for a DQN to learn some level of intuition about RNA folding, at least insofar as how to turn an invalid coloring into a valid coloring. This intuition may be prove valuable for the difficult optimizations of long RNA sequences.

References

Anderson-Lee, Jeff, Fisker, Eli, Kosaraju, Vineet, Wu, Michelle, Kong, Justin, Lee, Jeehyung, Lee, Minjae, Zada, Mathew, Treuille, Adrien, and Das, Rhiju. Principles for predicting rna secondary structure design difficulty. *Journal of molecular biology*, 428(5):748–757, 2016.

Eddy, Sean R. How do rna folding algorithms work? *Nature biotechnology*, 22(11):1457–1458, 2004.

Lee, Jeehyung, Kladwang, Wipapat, Lee, Minjae, Cantu, Daniel, Azizyan, Martin, Kim, Hanjoo, Limpaecher, Alex, Gaikwad, Snehal, Yoon, Sungroh, Treuille, Adrien, et al. Rna design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences*, 111(6):2122–2127, 2014.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Zadeh, Joseph N, Wolfe, Brian R, and Pierce, Niles A. Nucleic acid sequence design via efficient ensemble defect optimization. *Journal of computational chemistry*, 32 (3):439–452, 2011.