Project description for course CS-E4880
(Machine Learning in Bioinformatics)
RNA Folding Using Reinforcement Learning

April 10, 2019

Instructor: Mehdi Saman Booy (`mehdi.samanbooy@aalto.fi`)

We are trying to implement a simple RNA folding predictor in a Reinforcement Learning framework. (I would recommend to work on folding due to its simplicity in comparison with the inverse problem. However, if anyone wants to work on the inverse folding, please drop me an email). Meanwhile, we are tying to explore this field a little bit together. Then, please be creative; share and test your ideas (but have your own implementation of course). RL is a flexible, yet powerful framework! Please read the instruction twice, because classes are related to each other in a cyclic way and it's hard to explain them in a linear format :).

Short Version

> Design a simple RL framework for RNA folding to fold sequences with length 10. It must be more general, but we are working with predefined length (don't use 10 as a hard-code number in your code). The goal is finding the structure with maximum number of pairings w.r.t Watson-Crick base-pairing only ($A-U/G-C$).

In the following, you can find more description about different classes and modules. It's recommended to check simple examples for an environment (like from OpenAI's Gym) and simple policy networks.

1 Environment

First element of an RL framework is the environment which depend on the problem should be implement or used. There are plenty of toy-example for envs (e.g. OpenAI Gym). In our case, we should implement the env manually. You can see gym's envs to have some ideas about major functions in an env (however, we don't want to implement a general env like gym examples). Important point about implementing (designing) an environment for an RL framework is defining the **states**, the **actions** and the **reward function**. Think about them and also proper way to represent them.

Let say `EnvRNA` is our env. It must have two major functions:

1.1 `reset`

Reset the initial state of the agent.
Hint1: One good way to implement and use the `reset` function is to initialize what you want, then call it in your constructor (`__init__`).
Hint2: You can ignore this function in some cases, but in some learning modes it's better to have reset (like when you want to start an epoch for a new sequence)

1.2 `step`

Function to do an action (step is a standard name for this function using by OpenAI Gym). Input is `action` and the outputs are `observation`, `reward`, `done`, and `info` (more info here).

As you can see, `step` function doesn't need *state* as input. Therefore, `EnvRNA` must keep current state (or even more). In fact, it needs something like an object from `RNA` class.

2 RNA

As you may notice, the core of our env is RNA. It's possible to handle functions like `pairing` in `EnvRNA` itself (feel free to do that), but I think having another class for `RNA` provides better coupling. Previous functions were typical elements of each env but RNA is what you should think about its implementation a bit more. Then, choose proper representation of sequence and structure (pairing list, dot matrix, or ...). Representation matters due to its effect on functionality. Also, you can have multiple representations and sync them. All in all, your `RNA` class must contains followings:

- `sequence`: An RNA sequence which only has $\{A, U, G, C\}$. If you want to use other representation instead of string, keep the sequence also for logging purposes (printing for example).
- `structure`: Current structure (state) in proper format(s).
- `pairing`: It is a function to pair to nucleotides together, hence has two inputs. For simplicity, we are focusing on secondary structure so before do pairing, check the crossing with new pairs. If it makes a crossing, do nothing; else connect them (don't forget to update all of your representations of structure if you have more than one). Also, check if two bases are already connected to each other. It doesn't change our simple implementation, but we can unpair two already connected bases (it prevents the model to choose same action again and again).

  Note: Pairing is an one-to-one relationship; hence, if base i connected to j, j should connect to i.
- `free_energy`: It is a measure for current structure given the input sequence. For now, consider number of connected bases as the energy value (reward). Maybe I will provide some ready-to-use function to have more realistic free energy function using Nupack or ViennaRNA.

After implementing your environment, try to test it with some simple examples to make sure it works fine. Start with a sequence, call `pairing` and see what will happen. I will provide a function to visualize the arc-diagram for a structure. Also, for visualizing the dot matrix, you can use `matshow` function from Matplotlib library. After finishing the env which works fine, you have passed a crucial step.

3 Agent (Policy) and Learning

```python
policy = Policy()
optimizer = optim.Adam(policy.parameters(), lr=1e-2)

running_reward = 0
MAX_ITER = 300
N = 10
alpha = .95

for i_episode in count(1):
    seq = generate_random_seq(N)
    state = env.reset(seq)
    best_reward = 0

    for t in range(MAX_ITER):  # Don't use infinite loop while learning
        rewards = []
        action = select_action(policy, state)
        state, reward, done, _ = env.step(action)

        rewards.append(reward)
        reward = max(reward, best_reward) # keep the best reward in the episode
        if done:
            break

    # You can update it in different ways! using best_reward or some other things
    running_reward = running_reward * alpha + best_reward * (1-alpha)
    # Dynamic Programming / Monte Carlo / Temporal Difference for updating the policy
    finish_episode(optimizer, rewards)
```

Figure 1: General idea of the agent

The general idea of the agent is like figure 1. It's recommended in tabular RL to see each (state, action) pairs several times, although we know it's not possible when the search space becomes huge! However, try to do it many times and check the running reward to see if it's getting better or not after several episodes. Therefore, we are doing the following several times:

We have a policy (NN or any other things. Even a table contains best action for each state). Then for some iterations we are tying to choose an action and do it (interaction with our env). If we didn't find the solution (sometime we don't know the solution at all so that try to define a higher bound as the solution! e.g. in our simple example the higher bound is when all bases are paired.), we will stop after `MAX_ITER` iterations. Finally, we should update the policy (here with optimizer due to their connection in the computational graph in Pytorch) with respect to our information from the current episode. What you see in the aforementioned code is not complete, hence you need some other data to update the policy after each episode. It's highly recommended to read a bit about *Monte Carlo (MC)* and *Temporal Difference (TD)* to understand their different approaches for updating the policy (model).

Think about these questions

- What are the states and actions?
- What is the goal? so that what can be a proper reward function?
- Should I use MC or TD for updating the model?

- How does the model choose an action? (e.g. choosing the action with the highest probability or sampling from the action space or ...)