NEU502B Homework 5

Due April 15, 2024

Submission instructions: First, rename your homework notebook to include your name (e.g. `homework-5-nastase.ipynb`); keep your homework notebook in the `homework` directory of your clone of the class repository. Prior to submitting, restart the kernel and run all cells (see *Kernel > Restart Kernel and Run All Cells...*) to make sure your code runs and the figures render properly. Only include cells with necessary code or answers; don't include extra cells used for troubleshooting. To submit, `git add`, `git commit`, and `git push` your homework to your fork of the class repository, then make a pull request on GitHub to sync your homework into the class repository.

In the first homework assignment, we explored how a system can extract latent structure in sensory stimuli (e.g. natural scenes) using unsupervised learning algorithms like Hebbian learning. Our model was shown a set of images with no final goal specified, nor any expectations with which to compare its performance throughout learning. Now, we're interested in how a system can learn to reach a goal through interactions with its environment, by maximizing rewards or minimizing penalties.

Reinforcement learning (RL) models solve problems by maximizing some operationalization of reward. These models use goal-directed learning to solve closed-loop problems: present actions influence the environment, thus changing the circumstances of future actions toward the same goal. In RL, we hope to discover the actions that increase chances of rewards within specific states in the environment.

An agent must be able to sense the state of the environment either fully or partially, and its actions must be able to change this state. Consider the following example from Sutton and Barto (1992):

> "Phil prepares his breakfast. Closely examined, even this apparently mundane activity reveals a complex web of conditional behavior and interlocking goal-subgoal relationships: walking to the cupboard, opening it, selecting a cereal box, then reaching for, grasping, and retrieving the box. Other complex, tuned, interactive sequences of behavior are required to obtain a bowl, spoon, and milk jug. Each step involves a series of eye movements to obtain information and to guide reaching and locomotion. Rapid judgments are continually made about how to carry the objects or whether it is better to ferry some of them to the dining table before obtaining others. Each step is guided by goals, such as grasping a spoon or getting to the refrigerator, and is in service of other goals, such as having the spoon to eat with once the cereal is prepared and ultimately obtaining nourishment."

At each point in time, there is a state-action pair. Some of them fall under sub-goals, while others could ultimately be a state where there is a high chance of reward, fulfilling the goal of feeding. To be able to model this process, we have to break it down into its interacting components:

- The agent has a policy, the map between perceived states and the actions taken. We can think of it as a set of stimulus-response rules or associations that determine behavior given a state and a goal within the environment. It can be implemented through the probabilities of taking specific actions given a state.
- This set of rules should serve to maximize the reward signal in the short and/or long term.
- Environmental states are evaluated through a value function, which provides a measure of the expected rewards that can be obtained moving forward from a specific state. Grabbing a bowl might not feed you immediately, yet it has high value as it will lead you to a state in which you can feed yourself some cereal without spilling milk all over the table. Would grabbing a shallow dish instead of a bowl have the same value? Actions are taken based on these value judgements.
- The agent could have the ability for foresight and planning if it has a model of the environment. This means it can have a model of how the environment reacts to its behavior, from which to base its strategies and adjustments.

At each decision, the agent has a choice to either exploit the actions it has already tested to be effective, or it can explore the action-state space to find new routes to optimal rewards. Exploration is risky, yet under some circumstances it will pay off in the long run. Finding the balance between the two would be the optimal solution in uncertain environments. Different methods can be employed to deal with this duality:

- On-policy methods improve the policy that is used to make decisions. This policy is generally soft (probabilistic), as $P(s \in S, a \in A \mid s) > 0$, where S is the possible states and $A \vee s$ is the possible actions given a state. The probability is gradually shifted to a deterministic optimal policy with each update. For example, $\epsilon - greedy$ policies choose an action that has maximal expected value most of the time (with probability 1 – a small number ϵ). However, with probability ϵ the agent will choose an action at random. The agent will try to learn values based on subsequent optimal behavior, yet it has to behave non-optimally (choosing random actions) in order to explore and find the optimal actions. This means the agent has to learn about the optimal policy while behaving according to an exploratory policy. On-policy can be thought of as a compromise, where values are learned for a near optimal policy that still explores.
- Another approach is to use two policies, a target policy and a behavior policy. The first one is modified based on the information that is collected through the behaviors generated by the second. This approach is termed off-policy, as learning occurs based on behaviors generated off the policy being developed. The benefit here is that the target policy can be deterministic (i.e. greedy), while the behavior policy can continue to explore without limits.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
```

Problem 1: Fitting RL models to data

First, familiarize yourself with the two-step RL task (Daw et al., 2011). Visit this website to play through an example of the two-step RL task: https://nivlab.github.io/jspsych-demos/tasks/two-step/experiment.html. If you're interested, the Python code for the task can be found at: https://github.com/nivlab/jspsych-demos/tree/main/tasks/two-step.

The data from the two-step task are structured as follows:

- **choice1**: your choices at the first level (1 or 2)
- **choice2**: your choices at the second level (1 or 2)
- **state**: which second level game you were offered on this trial
  - choice1 = 1 at the first level (S1) leads to S2 in approximately 70% of the trials
  - choice1 = 2 at the first level (S1) leads to S3 in approximately 70% of the trials
- **money**: did you get a reward on each trial or not (0 or 1)

Note that missed trials will have a 0 in the choice; trials can be missed either at the first or second level. When you write your code (later on), make sure to deal separately with missed trials as this is a common source of discrepancies while fitting the models. Below is a schematic representation of the task structure:

	A_1	A_2
S_1	$Q_{S1,A1}$	$Q_{S1,A2}$
S_2	$Q_{S2,A1}$	$Q_{S2,A2}$
S_3	$Q_{S3,A1}$	$Q_{S3,A2}$

The schematic does not map to the colors used in the actual task. S1 refers to the state at the top (first) level, where you will be shown two distinct rocket ships. You will have to choose one of the two (represented by action | state in the schematic). One of the rockets, let's say A1|S1 will have 70% chance of transferring you to S2 (one of the possible states at the bottom level), and a 30% chance of getting you to S3. This is represented by the thickness of the arrows. For A2|S1, the chances are inverted. At the bottom (second) level, you can be at either of two distinct states (S2 or S3). You will need to choose between two aliens at each state, with gradually drifting chances of getting a reward once a decision is made. For example, A1|S2 might start with higher chances than A2|S2. These probabilities will change gradually with time and at some point, the chances might be reversed. There is no implicit relationship between what happens in S2 and S3. You will have to learn, with each experience, which choices lead you to better rewards.

Let's load in the data for one of our subjects. Make sure you understand what each variable contains.

```
data = np.load('sub-0.npz')

c1 = data['choice1']
c2 = data['choice2']
s = data['state']
m = data['money']
```

At the bottom level, learning can be modeled with Q-learning or Rescorla-Wagner learning, as there's no future state. Note that these learning rules are identical if you treat each option as an action (in Q-learning) or as a state (the state of the chosen stimulus, in Rescorla-Wagner).

> **Q-learning**: $Q^{new}(a \vee s) \leftarrow Q(a \vee s) + \eta * (R_t - Q(a \vee s))$
>
> **Rescorla-Wagner learning**: $V_{t+1} \leftarrow V_T + \eta * (R_T - V_T)$

Question: Describe in words the variables in each equation how each the learning rule "works":

> *The Q-learning rule uses the current value of an action at a particular state ($Q(a \vee s)$), along with the learning rate η and reward of the current action R_t to update the next value of this action and state ($Q^{new}(a \vee s)$). It works by scaling the difference between the reward and value of the current action at a particular state (the reward prediction error) by the learning rate (which determines how much the model learns from the reward prediction error, in a sense), and adding it to the current value to get the next value of that action at that state.*
>
> *The Rescorla-Wagner learning rule is the same, except it uses the value of the current state V_T, along with the learning rate η and reward of the current state R_T to update the value of the next state. It again works by scaling the difference between the reward and value of the current state (the reward prediction error) by the learning rate (which determines how much the model learns from the reward prediction error, in a sense), and adding it to the value of the current state to get the next value of that state.*

At the top level, learning can be modeled in several different ways. We'll consider two: (1) **model-free learning** and (2) **model-based learning**.

For **model-free learning**, we'll start with the temporal difference (TD) learning rule.

> **TD(0)**: $V_T \leftarrow V_T + \eta * (R_T + \gamma * V_{T+1} - V_T)$

Here, $R_T = 0$ because the first state doesn't yield rewards and γ is the temporal discount parameter of future rewards—this allows us to adjust the first-stage actions by taking into account the result of the second-stage action.

One way to make learning more efficient is to use TD(λ) instead of TD(0) learning. In this case, we add an additional memory variable associated with each state to serve as an "eligibility trace". You can think of it as a "memory" that a particular state has been visited, which decays (e.g. exponentially) over time. Every time a state is visited, its eligibility trace becomes 1; at every subsequent time point, the eligibility trace is multiplied by a factor $0 < \lambda \leq 1$. At the end of a trial or episode, all eligibility traces become 0.

All states are updated according to *learning rate* · *prediction error* · *eligibility trace*. This will automatically update all the states visited in this episode (i.e. all the states "eligible" for updating), doing so for the most recently visited states to a greater extent. Write the updated equation.

> **TD(λ)**: $V_{T+1} = V_T + \eta * E(\lambda) * (R_T + \gamma V_{T+1} - V_T)$

Question: Again, describe the variables in these equations and how the learning rules "work":

> *The TD learning rule uses the value of the current state V_T, the learning rate η, eligibility trace $E(\lambda)$, the reward at the current state R_T, and the next value multiplied by a temporal discount parameter to scale it (γV_{T+1}) to update the next value of the state V_{T+1}. It works by taking the reward prediction error (reward of the current state minus the value of the current state), adding a scaled version of the next value to take the next stage into account, scaling that entire value by the eligibility trace (memory of visiting this state) and learning rate like before, and finally adding this to the value of the current state to get the next value of the current state.*

For **model-based learning**, let's begin by assuming that transition model (i.e. the probabilities of going from $S1$ to $S2$ or $S3$ given choice1) is known from the start—while the reward model is not known.

Question: How can you use the transition probabilities and the learned values at the second-stage states to plan and make choices at the first stage? How would you implement this model?

> *You can use the transition probabilities and learned values at the second-stage states to find an expected value for each choice in the first stage, and then you can plan the first-stage choice based on which gives you the highest expected learned value at the second stage. You can implement this by summing the products of the transition probabilities and learned values corresponding to each first stage choice.*

Question: How many parameters do each the four models have?

> *The Q-learning model has 1 parameter, the Rescola-Wagner model has 1 parameter, the TD(0) model has 2 parameters, and the TD(λ) model has 3 parameters.*

Now we'll implement and fit the following models. Implement TD(λ) using the Q-learning and State-Action-Reward-State-Action (SARSA) algorithms. Some pseudocode is provided to get you started. These algorithms use state-action value predictions (Q values) to choose actions. In state S, the algorithm chooses an action according to softmax Q values.

Here, β is an inverse-temperature parameter that we'll optimize. If you're using constrained optimization, fix β to be in the range [0, 100].

Update the eligibility traces. Recall that the eligibility traces are values corresponding to each state and action pair, and are set to zero at the beginning of the trial. Upon taking action a to leave state S for state S^{new} and receiving reward r, the eligibility traces $e(a \mid S)$ are updated for each (S, a) pair:

All $Q(a \vee S)$ are updated according to:

With prediction error $\delta(t)$ being:

The parameter η is a step-size or learning-rate parameter in the range (0,1].

Reset the eligibility traces to 0 at the end of each round.

```
def rl_nll(params, state, choice1, choice2, money, model_based=False):
    """
    Use either SARSA or Q-learning to update the negative log-
likelihood over all trials
    """
    eta, beta, lambd = params
    n_states = 3
    n_actions = 2
    n_trials = s.shape[0]

    # Initialize an array to store Q-values that is size n_states x
n_actions
    Q = np.zeros((n_states, n_actions))

    # Initialize log-likelihood
    LL = 0

    for t in range(n_trials):

        # Create an n_states x n_actions matrix to store your
eligibility traces for the current trial
        E = np.zeros_like(Q)

        # Get your current state for the top level (S1)
        S = 0

        # Stop if trial was missed. Missed trials will have a value of
-1.
        if choice1[t] == -1:
            continue

        # First level choice likelihood: compute likelihood of choice
at the first state S1.
        # Your likelihood should be a softmax function.
        Q_a_s = Q[S, choice1[t]]
        Q_aprime_s = Q[S, :]
        p_chosen = np.exp(beta * Q_a_s) / np.sum(np.exp(beta *
Q_aprime_s))

        # Update the log likelihood
        LL += np.log(p_chosen)

        # Learning at first level: update your eligibility trace
according to
        # e(a|S) = 1                       for the chosen action (a)
in the current state (S)
```

```
        # e(a|S) = lambda * e(a|S)         for all other a, S pairs

        E = lambd * E # update all first
        E[S, choice1[t]] = 1 # then update chosen action/state
differently


        # Update prediction error without reward (because we are in
the first level)
        if model_based:
            # Implement SARSA update for model based learning
            # Keep in mind that choosing 1 at the first level (S1)
leads to S2 in approximately 70% of the trials
            # and choosing 2 at the first level (S1) leads to S3 in
approximately 70% of the trials
            if choice1[t] == 0:
                PE = 0.7 * np.max(Q[1, :]) + 0.3 * np.max(Q[2, :]) -
Q_a_s
            else:
                PE = 0.3 * np.max(Q[1, :]) + 0.7 * np.max(Q[2, :]) -
Q_a_s
        else:
            # Implement Q-learning update for model free learning
            PE = np.max(Q[state[t]]) - Q[S, choice1[t]]

        # update Q values according to Q = Q + eta * prediction errror
* eligibility
        Q = Q + eta * PE * E

        # Get your current state for the second level (S2 or S3)
        S = state[t]

        # Stop if trial was missed at the second level. Missed trials
will have a value of -1
        if choice2[t] == -1:
            continue

        # Second level choice likelihood: compute likelihood of choice
at the second state (S2 or S3).
        # Your likelihood should be a softmax function.
        Q_a_s = Q[S, choice2[t]]
        Q_aprime_s = Q[S, :]
        p_chosen = np.exp(beta * Q_a_s) / np.sum(np.exp(beta *
Q_aprime_s))

        # Update the log likelihood
        LL += np.log(p_chosen)

        # Learning at second level: update your eligibility trace
```

```
according to
        # e(a|S) = 1                       for the chosen action (a)
in the current state (S)
        # e(a|S) = lambda * e(a|S)         for all other a, S pairs
        E = lambd * E # update all first
        E[S, choice2[t]] = 1 # then update chosen action/state
differently

        # Update the prediction error with reward because we are in
the second level
        # NOTE: This update IS NOT dependent on the next state because
we are in the final state
        PE = money[t] - Q[state[t], choice2[t]]

        # update Q values according to Q = Q + eta * prediction errror
* eligibility
        Q = Q + eta * PE * E

    return -LL
```

Question: Is the prediction error (δ) update in the second stage fundamentally similar or different between Q-learning and SARSA? Explain your answer.

> *The prediction error update is the same between Q-learning and SARSA because the secoond stage is the last stage, so we are not including the term corresponding to the future stage in our prediction error equation and the two become the same.*

Question: Which of these two algorithms is considered on-policy, which is off-policy, and why?

> *An on-policy algorithm is one in which the participant is trying to learn the same model that they are using to select their actions. SARSA would therefore be considered on-policy, since we are using transition probabilities and the predicted values at the second stage to inform our choice at the first stage, essentially updating the model based on future predicted values of the same model. An off-policy algorithm, on the other hand, is one in which the participant is using a different model to make their choices than the one they're trying to learn. Q-learning would be considered off-policy since we are updating the model based on the maximum Q-value in the new state instead of future predictions from the same model itself.*

For each subject, load in their data as described at the beginning of the assignment (`sub-0.npz` to `sub-4.npz`). The `sub-0.npz` file contains sample data, while the rest are experimental data collected from other students at PNI. Use SciPy's `minimize` function (imported at the beginning of the problem set) to fit the two models to each of the subjects. You may also want to keep of the number of trials completed by each subject.

```
# Set some parameters
np.random.seed(1312)
params = [.5, 50, .5]
bounds = [(0, 1), (0, 100), (0, 1)]
sub_fns = ['sub-0.npz', 'sub-1.npz', 'sub-2.npz',
           'sub-3.npz', 'sub-4.npz']

# Example solver method for SciPy's minimize
method = 'TNC'

# Loop through subjects, load data, and fit models:
sarsa_nlls = []
qlearn_nlls = []
num_trials_completed = []
for sub in sub_fns:

    # Load data
    data = np.load(sub)
    c1 = data['choice1']
    c2 = data['choice2']
    s = data['state']
    m = data['money']

    # Fit SARSA model
    sarsa_nll = minimize(rl_nll, params, args = (s, c1, c2, m, True), 
method = method, bounds = bounds).fun
    sarsa_nlls.append(sarsa_nll)

    # Fit Q-learning model
    qlearn_nll = minimize(rl_nll, params, args=(s, c1, c2, m, False), 
method=method, bounds=bounds).fun
    qlearn_nlls.append(qlearn_nll)

    # Keep track of the number of trials completed by this subject
    # Get the length of c1 (or any of the data variables) without the 
missed/incomplete trials
    ntrials = len(c1[(c1 != -1) & (c2 != -1)])
    num_trials_completed.append(ntrials)
```

Use Bayesian information criterion (BIC) to compare which is the best-fitting model for each subject. Compute BICs using the following formula:

$$BIC = -2 * \text{log-likelihood} + \ln(\text{number of trials}) * \text{number of parameters}$$

where $\ln()$ is the natural logarithm. BIC is defined here on the deviance scale, which means that lower values are better. **Question**: Which model fits each subject's behavior best?

```
# Compute BIC for each model and subject:
for sub in range(len(sub_fns)):
    sarsa_ll = -1*sarsa_nlls[sub]
    qlearn_ll = -1*qlearn_nlls[sub]
    n_trials = num_trials_completed[sub]

    sarsa_bic = -2*(sarsa_ll) + np.log(n_trials) * 3
    qlearn_bic = -2*(qlearn_ll) + np.log(n_trials) * 3
    print(f'Subject {sub}:')
    print(f'SARSA BIC: {sarsa_bic}')
    print(f'Q-learning BIC: {qlearn_bic}\n')

Subject 0:
SARSA BIC: 1202.562236030643
Q-learning BIC: 1191.6859014140534

Subject 1:
SARSA BIC: 530.4206871391603
Q-learning BIC: 528.7173802767219

Subject 2:
SARSA BIC: 500.62679205168524
Q-learning BIC: 499.9937000439658

Subject 3:
SARSA BIC: 565.2977132218048
Q-learning BIC: 565.2816069501259

Subject 4:
SARSA BIC: 548.3911469989902
Q-learning BIC: 546.8618098603524
```

Though the results are very close, the Q-learning model BIC is a bit lower for every subject, meaning that it fits each subject's behavior best.

Problem 2: Cliff walking

Consider the grid world shown below. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked "The Cliff." Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

Two paths are marked: an optimal path which incurs the least costs on the way to the goal, and a roundabout (but safe) path that walks farthest from the cliff.

Question: Which algorithm, SARSA or Q-learning, would learn either path, and why?

I think that the SARSA algorithm would learn the safe path, since it uses predicted values at the next state to make each current state's choice, so the higher expected values assoociated with being near the cliff would push it away from the cliff and onto the safe path. The Q-learning algorithm would likely learn the optimal path however, since it uses the maximum value for each state to make each choice, so it will learn the most cost-effective, optimal path.

Question: When behaving according to the softmax of the learned Q values, which path would an agent prefer? (Consider the parameter β and the stability of the environment.)

> *The equation for the softmax of the learned Q values takes into account the optimal Q values for all actions in a given state, so based on this algorithm I think that an agent would prefer the optimal path. The β parameter determines the degree to which value estimates influence choice, and so I think that this will also further push towards a greedy, value-maximizing path like the optimal path.*

Question: Can you explain why on-policy methods might be superior for learning real-world motor behavior?

> *On-policy methods would be better for the real-time prediction error calculations and subsequent adjustments needed for motor behavior in the real world, which is why such methods would be superior for this application. This is because on-policy methods are more flexible and take into account sudden changes like obstacles or external forces better than off-policy, which is really important for motor control.*

References

- Daw, N. D., Gershman, S. J., Seymour, B., Dayan, P., & Dolan, R. J. (2011). Model-based influences on humans' choices and striatal prediction errors. *Neuron*, 69(6), 1204–1215. https://doi.org/10.1016/j.neuron.2011.02.027

- Sutton, R. S., & Barto, A. G. (1992). Reinforcement Learning: An Introduction. MIT Press.