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 $P(s∈S,a∈A|s)>0$, where S is the possible states and $A|s$ 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.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
%autosave 60
```

```
Autosaving every 60 seconds
```

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.

In [2]:
```
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**: $Q^{new}(a|s) \leftarrow Q(a|s) + \eta * (R_t-Q(a|s))$
>
> **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":

> **Q-learning**: The estimated Q-value $(Q^{new}(a|s))$ for an action (a) in state (s) is updated based on the prediction error between the expected Q-value $(Q(a|s))$ and the actual reward received (R_t), scaled by the learning rate (η). This learning process iteratively refines the estimated Q-value, allowing the agent to evaluate how good it is to take a certain action based on the predicted reward of that action. This is a model of operant conditioning.
>
> - $Q^{new}(a|s)$: The updated Q-value for taking action a in state s.
> - $Q(a|s)$: The current Q-value for taking action a in state s (before the update).
> - η: The learning rate. This influences how much new information is used to update the Q-value.
> - R_t: The actual reward received after taking action a in state s. This is used to compute the difference (i.e. prediction error) between the actual reward received and our current estimate of the Q-value.
>
> **Rescorla-Wagner learning**: The associative strength (V_{t+1}) is updated based on the prediction error between our expected value (V_{t}) and the actual reward received (R_t), scaled by the learning rate (η). This

> learning process allows the agent to gradually converge towards accurate predictions of the reward associated with a given stimulus. This is a model of classical conditioning.
>
> - V_{t+1}: The updated value of associative strength at the next time step.
> - V_{t}: The value of associative strength at the current time step (before the update).
> - η: The learning rate. This influences how much new information is used to update the associative strength.
> - R_t: The actual reward received at the current time step. This represents the true value of the most recently experienced stimulus and is used to compute the difference (i.e. prediction error) between the expected and actual outcome.

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

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 $|cdot$ prediction error $|cdot$ 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":

> **TD(0)**: The predicted value of a state is updated based on the TD error $(R_T+\gamma*V_{T+1}-V_T)$, scaled by the learning rate (η). The TD

> error represents the discrepancy between the agent's expected value of being in a certain state and the actual reward received in that state.
>
> - V_{T+1}: The updated estimate of the value at time $T+1$.
> - V_{T}: The current estimate of the value at time at time T.
> - R_T: The actual reward received at time (T).
> - η: The learning rate. This determines how quickly the value estimates are updated with respect to new information.
> - γ The discount factor for future rewards, indicating how much future rewards are valued compared to immediate rewards.
>
> **TD(λ)**: Same as above, but with the incorporation of memory of past visited states. Memory is operationalized via eligibility traces ($E(\lambda)$), which provide a mechanism to update states based on experience. Here, the TD error is scaled by both the learning rate and the eligibility trace.
>
> - $E(\lambda)$: The eligibility trace, which represents the memory of a particular state. The eligibility trace for a visited state is initially set to 1 and decays exponentially over time based on the parameter λ. If states are revisited their trace is reset to 1.

$Q(A1,S1) = P(S2|S1,A2)V(S2) + P(S3|S1,A2)V(S3)$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?

> Given the current task structure, we know that actions taken at $S1$ influence the probability of being in a future state ($S2$ vs. $S3$). We can look at the probability of moving to state $S2$ or $S3$ from $S1$ if a certain action is taken ($A1$ vs. $A2$), and weigh these probabilities by how valuable states S2 and S3 are expected to be:
>
> - $Q(A1,S1) = P(S2|S1,A1)*V(S2) + P(S3|S1,A1)*V(S3)$
> - $Q(A1,S1) = P(S2|S1,A2)*V(S2) + P(S3|S1,A2)*V(S3)$
>
> where:
>
> - $V(S2) = \max(Q(A1,S2), Q(A2,S2))$
> - $V(S3) = \max(Q(A1,S3), Q(A2,S3))$
>
> Using the above formulae, we can select the action with the higher expected value at the first stage. A greedy policy is to choose the action at the first stage that offers the highest expected value:

> - If $Q(A1,S1) > Q(A1,S1)$, choose action $A1$.
> - Otherwise, choose action $A2$.
>
> Alternatively, we can choose actions proportionally to the estimated Q-values, allowing for some exploration and variation in choices.

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

> - **Q-learning**: One hyperparameter: learning rate(η).
> - **Rescorla-Wagner learning**: One hyperparameter: learning rate (η).
> - **TD(0)**: Two hyperparameters: learning rate(η), discount rate (γ).
> - **TD(λ)**: Three hyperparameters: learning rate(η), discount rate (γ), eligibility decay (λ).

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 $e(a|S)$ are updated for each (S, a) pair:

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

With prediction error $δ(t)$ 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.

In [3]:
```
from scipy.special import softmax

def rl_nll(params, state, choice1, choice2, money, model_based=False):
```

```
    """

    """
    eta, beta, lambd = params
    n_states = 3
    n_actions = 2
    n_trials = state.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 trac
        E = np.zeros([n_states, n_actions])
        # Every time a state is visited, its eligibility trace becomes 1; at e

        # 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 f
        # Your likelihood should be a softmax function.
        p_chosen = softmax(beta * Q[S])[choice1[t]]
        # 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 cu
        # e(a|S) = lambda * e(a|S)        for all other a, S pairs
        E *= lambd
        E[S, choice1[t]] = 1

        # Update prediction error without reward (because we are in the first
        if model_based:
            # Implement SARSA update for model based learning
            # Keep in mind that choosing 1 at the first level (S1) leads to S2
            # and choosing 2 at the first level (S1) leads to S3 in approximat
            if choice1[t]==0:
                p2 = 0.7
                p3 = 0.3
            else:
                p2 = 0.3
                p3 = 0.7
            PE = p2*Q[1].max() + p3*Q[2].max() - Q[S, choice1[t]]
        else:
            # Implement Q-learning update for model free learning
            PE = max(Q[state[t]]) - Q[S, choice1[t]]

        # update Q values according to Q = Q + eta * prediction errror * eligi
        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
        if choice2[t] == -1:
            continue

        # Second level choice likelihood: compute likelihood of choice at the s
        # Your likelihood should be a softmax function.
        p_chosen = softmax(beta * Q[S])[choice2[t]]
        # Update your 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 cu
        # e(a|S) = lambda * e(a|S)        for all other a, S pairs
        E *= lambd
        E[S, choice2[t]] = 1

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

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

    return -LL

# params = [.5, 50, .5]
# rl_nll(params, s, c1, c2, m)
```

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

> - Since there is no future reward to estimate beyond the second state, the prediction error is updated in a similar manner for both approaches.
> - The second stage serves as the terminal state, where no subsequent decisions are made. As a result, no future states are considered in the SARSA method; instead, the prediction error is updated solely based on the immediate reward and the state-action value of the current (and final) choice. The latter aligns with the prediction error update for Q-learning, where no future states are considered.

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

> **Q-learning**: off-policy
>
> - Q-learning is considered an off-policy method because it updates the Q-value solely based on the maximum possible outcome. This means that it calculates the expected value value of a state by assuming that the most optimal (greedy) actions are taken in the future. This is irrespective of the current policy that the agent is actually using to make decisions.
>
> **SARSA**: on-policy

> - SARSA is an on-policy algorithm because it updates Q-value based on the Q-value of the next state and the action taken based on the current policy being followed. SARSA computes the expected value for state-action pairs based on the premise that the agent continues to adhere to its current policy.

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.

In [4]:
```
# 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'

def calculate_bic(ll, n_trials, n_params):
    return -2 * ll + np.log(n_trials) * n_params

# Loop through subjects, load data, and fit models:
results = {}
for sub_fn in sub_fns:
    data = np.load(sub_fn)
    c1 = data['choice1']
    c2 = data['choice2']
    s = data['state']
    m = data['money']
    n_trials = len(c1)

    result_ql = minimize(rl_nll, params, args=(s, c1, c2, m, False), bounds=bou
    bic_ql = calculate_bic(-result_ql.fun, n_trials, len(params))

    result_sarsa = minimize(rl_nll, params, args=(s, c1, c2, m, True), bounds=b
    bic_sarsa = calculate_bic(-result_sarsa.fun, n_trials, len(params))

    results[sub_fn] = {
        'Q-learning': {'params': result_ql.x, 'BIC': bic_ql},
        'SARSA': {'params': result_sarsa.x, 'BIC': bic_sarsa}
    }
```

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?

In [6]:
```
for sub_fn in sub_fns:
    bic_ql = results[sub_fn]['Q-learning']['BIC']
    bic_sarsa = results[sub_fn]['SARSA']['BIC']
    best_fit = 'Q-learning' if bic_ql < bic_sarsa else 'SARSA'
    print(f"{sub_fn}: Best fitting model is {best_fit} with BIC = {min(round(b
```

```
sub-0.npz: Best fitting model is Q-learning with BIC = 1191.91
sub-1.npz: Best fitting model is Q-learning with BIC = 528.78
sub-2.npz: Best fitting model is Q-learning with BIC = 500.06
sub-3.npz: Best fitting model is Q-learning with BIC = 565.31
sub-4.npz: Best fitting model is Q-learning with BIC = 547.0
```

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?

> **Q-learning**: optimal path
>
> - Since Q-values are solely updated based on the highest possible outcome, the agent would likely learn to navigate the optimal path. Since the optimal path (even if it is close to the cliff) offers a faster route to the goal with less total penalty than the safer route, Q-learning will favor this path. More concretely, Q-learning does not factor in the potential risks of exploratory actions because the Q-value is updated solely based on the highest outcome possible.
>
> **SARSA**: roudndabout (safe) path
>
> - Since SARSA directly incorporates the consequences of its exploratory actions during learning, it would likely learn to navigate along a more conservative path. Using SARSA, the agent will occasionally choose non-greedy actions and is therefore likely to experience falling off the cliff during training. As SARSA incorporates the risk of exploration into its learning, it will eventually learn to avoid taking actions that could lead to the large negative reward.

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 behavior and choices of the agent can be significantly influenced by the temperature parameter β. The softmax decision rule offers a means for balancing exploration and exploitation by converting Q values into a probability distribution over actions, where higher Q values lead to proportionally higher probabilities of being chosen. The choice of path will vary depending on β as follows:
>
> - **High β**: The agent is more likely to learn the optimal path. This is because a higher β corresponds to a more greedy policy, meaning that the agent will learn the path with the highest Q-value more consistently.
> - **Low β**: The agent is more likely to learn the safer path. A lower β corresponds to a less greedy (more exploratory) policy, as actions are chosen more uniformaly. This means that the agent is more likely to explore different paths, ultimately favoring the safer path in order to avoid incurring a large negative reward.

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

> For real-world motor behavior (e.g. robotics or autonomous vehicles), executing actions derived from a stable and improving policy can reduce the risk of performing dangerous actions. On-policy methods improve on the curent poicy that is being used to make decisions, ensuring a more predictable and steady performance when executing actions in the real-world. Furthermore, since the real world environments are subject to change, motor tasks also require continuous adaptation. On-policy methods perform updates more regularly, making them well-suited for variable and underpredictable environments.

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.