

# 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  $P(s \in S, a \in A | s) > 0$ , where  $S$  is the possible states and  $A \forall s$  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  $1 - \epsilon$  – a small number  $\epsilon$ ). However, with probability  $\epsilon$  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.

```
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](https://github.com/nivlab/jspsych-demos/tree/main/tasks/two-step). 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).

$$\text{Q-learning: } Q^{new}(a \vee s) \leftarrow Q(a \vee s) + \eta * (R_t - Q(a \vee s))$$

$$\text{Rescorla-Wagner learning: } V_{t+1} \leftarrow V_t + \eta * (R_t - V_t)$$

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

*In the Q-learning equation, on the right,  $Q(a/s)$  represents the value of an action in a state. The second component on the right side of the equation is the reward at timepoint  $t$ , minus that value  $Q(a/s)$ , scaled by a learning rate  $\eta$ . On the left, the  $Q^{new}(a/s)$  variable represents the output of the equation; it's the new value of a certain action given a particular state. It's a function of the sum of  $Q(a/s)$  and the difference between reward and value at that timepoint scaled by the learning rate. The Rescorla-Wagner learning equation is very similar to Q-learning, but in terms of the value of the state instead of the action. Here, you get  $V_{t+1}$  (the value of the next state) by summing the value of the current state and that value minus the reward (scaled by learning rate) at that timepoint. These learning rules work by updating the value of either the action or the state at each timepoint as a function of learning rate and received reward.*

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.

$$\text{TD}(0): V_t \leftarrow V_t + \eta * (R_t + \gamma V_{t+1} - V_t)$$

Here,  $R_t=0$  because the first state doesn't yield rewards and  $\gamma$  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  $\text{TD}(\lambda)$  instead of  $\text{TD}(0)$  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.

$$\text{TD}(\lambda): V_{t+1} = V_t + \eta * E(\lambda) * (R_t + \gamma V_{t+1} - V_t)$$

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

*The goal of model free learning is for the agent to learn to maximize the amount of received cumulative reward over time without knowing the structure of the environment/task. The first equation represents learning at timepoint zero (in the first*

state) in which there is no reward information yet ( $R_t = 0$ ). It tries to estimate/predict the value in trial  $T$  based on past reinforcements and values. It includes gamma, which is the temporal discount parameter of future rewards which is multiplied by the temporal difference prediction error:  $V_{t+1} - V_t$ . The reward variable  $R_t$  would be added to this, but here it = 0. Like with Rescorla-Wagner, this is all scaled by the learning rate  $\alpha$ . The second equation aims to increase the learning efficiency of the first equation by using  $TD(\lambda)$  instead of  $TD(0)$ . This adds in the eligibility trace ( $E(\lambda)$ ) to the equation, which allows the algorithm to learn more quickly because the error signal can propagate backwards to update the value. The  $\lambda$  parameter controls how quickly the eligibility of past states/actions decays over time to be (or not to be, over time) factored in to current choices, because it is used in the equation to scale the updates to the value function.

For **model-based learning**, let's begin by assuming that transition model (i.e. the probabilities of going from  $S_1$  to  $S_2$  or  $S_3$  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?

*Since you know the transition probabilities given certain choices, but not the reward model, in order to plan and make choices at the first stage you would have to do a lot of trial and error to figure out which choices lead to more reward in the long run. In model based learning, you can use the structure of the task/environment (e.g. transition probabilities) to learn the values at each second-stage state. You can then choose which first stage choices (based on the task structure) lead to the most high value second-stage states and use that to guide sequential decisions. To implement this model, you would need to create a model that predicts the next state and reward given the current state and action of the agent, with known task structure. The environment would need to have known transition probabilities, states, actions, and rewards. For each run of the model, the agent would need to update its policy based on reward outcome and taking into account the probabilities of the first to second stage transitions. The policy updates could happen through some method of a Q-learning algorithm. In order to improve performance on the task, you would need to run many iterations.*

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

*In the Q-learning model, the parameter is the (1) the learning rate  $\alpha$ . In the R-W model, the parameter is (1) the learning rate  $\alpha$ . In the model-free TD learning model, the parameters are (1) the learning rate  $\alpha$  and (2) the discount factor gamma. For the  $TD(\lambda)$  model, the parameters are (1) the learning rate  $\alpha$ , (2) the discount factor gamma, and (3) the eligibility trace with trace decay rate  $\lambda$ .*

Now we'll implement and fit the following models. Implement  $TD(\lambda)$  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,  $\beta$  is an inverse-temperature parameter that we'll optimize. If you're using constrained optimization, fix  $\beta$  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  $Q(a \mid S)$  are updated according to:

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

The parameter  $\eta$  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):
    """
    """
    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 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.
    p_chosen = (np.exp(Q[S, choice1[t]] * beta)) /
np.sum(np.exp(Q[S, :]*beta))
    # 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 = lambda * E
    E[S,choice1[t]] = 1

    # 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
        Q_2 = Q[1,:]
        Q_3 = Q[2,:]
        if choice1[t] == 0:
            prob_2 = 0.7
            prob_3 = 0.3
        else:
            prob_2 = 0.3
            prob_3 = 0.7

        PE = (prob_2 * Q_2.max()) + (prob_3 * Q_3.max()) - Q[S,
choice1[t]]
    else:
        # Implement Q-learning update for model free learning
        PE = Q[state[t]].max() - Q[S, choice1[t]]

    # update Q values according to  $Q = Q + \eta * \text{prediction error}$ 
    * 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.
        p_chosen = (np.exp(Q[state[t], choice2[t]] * beta)) /
np.sum(np.exp(Q[state[t],:] * beta))
        # 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 current state (S)
        #  $e(a|S) = \lambda * e(a|S)$  for all other a, S pairs
        E = lambda * E
        E[state[t],choice2[t]] = 1

        # 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  $Q = Q + \eta * \text{prediction error}$ 
        # * eligibility
        Q = Q + eta * PE * E

    return -LL

```

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

*The prediction error in the second stage is fundamentally different between Q-learning and SARSA. To update the current action-value function, Q-learning uses the maximum estimated action-value of the state in the next timepoint. On the other hand, SARSA uses the action actually taken at the state in the next timepoint to update the value of the current state-action pair.*

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

*SARSA is considered on-policy and Q-learning is off-policy. This is because Q-learning learns the optimal action-value function to optimize cumulative reward regardless of the policy being followed. Q-learning can learn from actions taken from a greedy, exploratory policy. SARSA learns about the policy as it follows and updates its action-value estimates based on the actions of the policy 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 = []
q_learn = []
trials = []
for subject in sub_fns:
    data = np.load(subject)
    choice1 = data['choice1']
    choice2 = data['choice2']
    state = data['state']
    money = data['money']

    fit_sarsa = minimize(rl_nll, params, args = (state, choice1,
choice2, money, True), method = method, bounds=bounds).fun
    sarsa.append(fit_sarsa)
    fit_q = minimize(rl_nll, params, args = (state, choice1, choice2,
money, False), method = method, bounds=bounds).fun
    q_learn.append(fit_q)
    trials.append(len(choice1[(choice1 != -1) & (choice2 != -1)]))
```

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

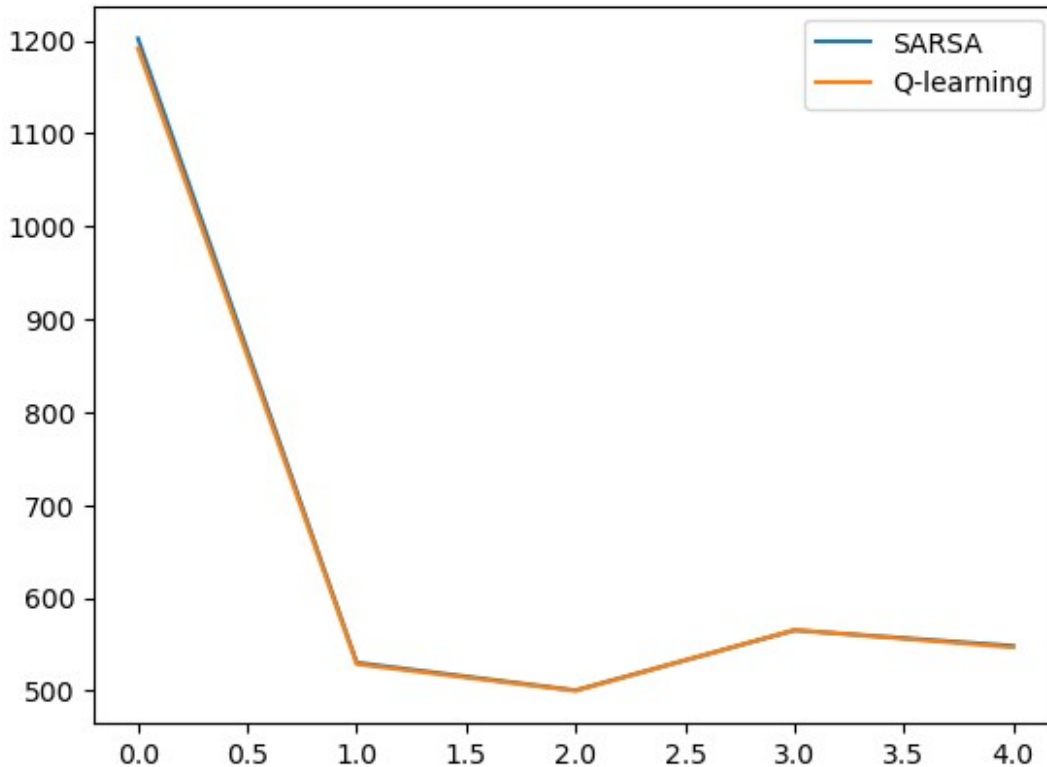
$$BIC = -2 * \log\text{-likelihood} + \ln(\text{number of trials}) * \text{number of parameters}$$

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:
sarsa_BIC = []
q_BIC = []
for i in range(len(sub_fns)):
    sarsa_BIC.append((-2 * sarsa[i] * -1) + (np.log(trials[i]) * 3))
    q_BIC.append((-2 * q_learn[i] * -1) + (np.log(trials[i]) * 3))
```

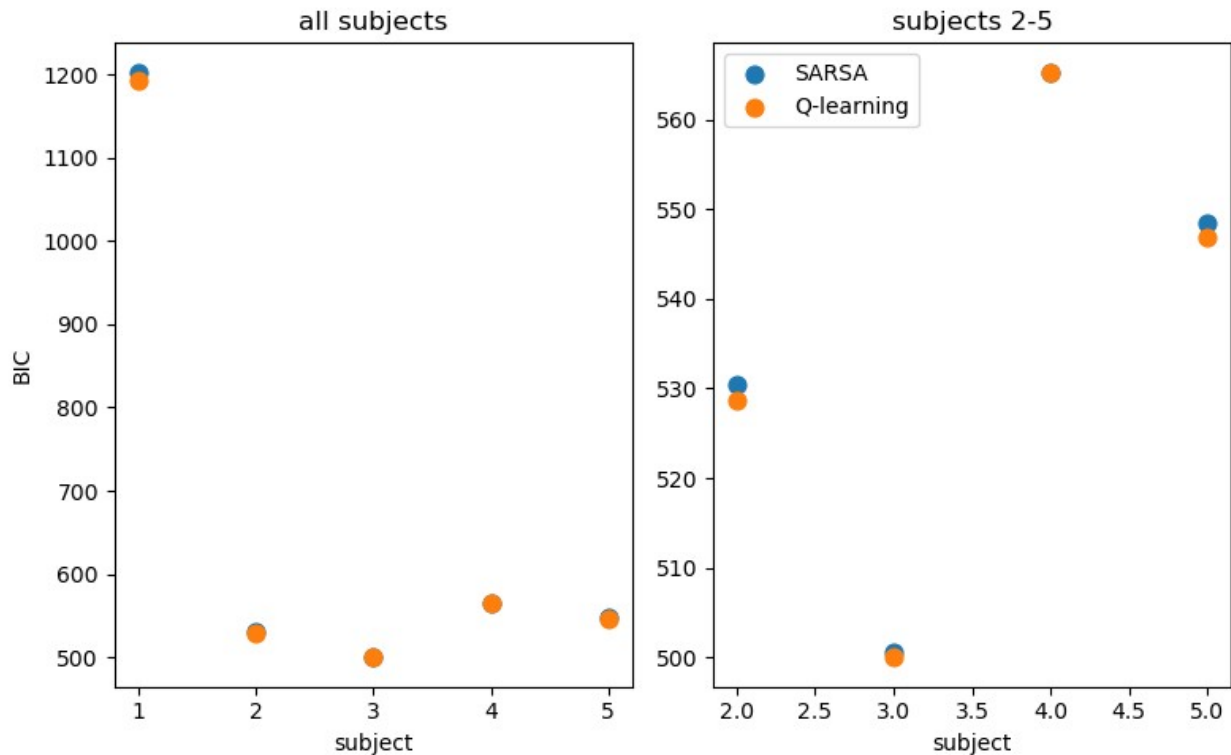
```
plt.plot(sarsa_BIC, label = 'SARSA')
plt.plot(q_BIC, label = 'Q-learning')
plt.legend()

<matplotlib.legend.Legend at 0x7f3cc2d5f2d0>
```



*Oop they look exactly the same to me but i probably did something wrong.. but they both did much better for subjects 2-5 and not very good for subject 1. From looking at the actual values, it looks like Q-learning fits the behavior slightly better for all of the subjects which you can see a little better with these scatter plots.*

```
x = np.arange(1,6)
fig,axs = plt.subplots(1,2,figsize = (8,5))
axs[1].scatter(x[1:],sarsa_BIC[1:], s = 60, label = 'SARSA')
axs[1].scatter(x[1:],q_BIC[1:], s = 60, label = 'Q-learning')
axs[1].set_xlabel('subject')
axs[0].set_ylabel('BIC')
axs[0].scatter(x,sarsa_BIC, s = 60, label = 'SARSA')
axs[0].scatter(x,q_BIC, s = 60, label = 'Q-learning')
axs[0].set_xlabel('subject')
axs[1].set_title('subjects 2-5')
axs[0].set_title('all subjects')
plt.legend()
plt.tight_layout()
```



## 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 would learn the optimal path, while SARSA would learn the safe path. This is because SARSA is on-policy and learns the action policies: it will update its learning rules taking into account the penalty of walking off the cliff so will take the safe route. On the other hand, Q-learning updates its action-values based on the optimal outcome without taking into account other outcomes (like walking off the cliff). Therefore, Q-learning would take the optimal, but more high risk, path.*

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

*When behaving according to the softmax of the learned Q values, the agent will choose a path based on  $\beta$ , which is the inverse temperature. The value of  $\beta$  reflects the stability of the task/environment in terms of what a certain choice will lead to different outcomes. If the environment is stable, the optimal path is preferable because there is*

*a lower chance of randomly walking off of the cliff. And if the environment is unstable the safe path would be preferred by the agent in order to avoid walking off of the cliff. This is reflected in the value of  $\beta$  because when  $\beta$  is high, the choice of the agent has high weight in determining the outcome (more stable environment) and therefore the agent can take the optimal path without falling off the cliff. The opposite is true when the value of  $\beta$  is low and choice is relatively random - here the safe path would be preferred because the environment is less stable.*

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

*On-policy methods might be better for learning real-world motor behavior because these methods would be better equipped to learn motor sequences and complex behaviors. This is because on-policy methods are more conservative in their updates and learn about the policy being followed. It learns about the value of actions based on the actual actions, which would be important in learning a complex motor sequence, like making a sandwich for example. It learns about the actions at each step that lead to the higher value outcome.*

## 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.