homework-5-ku

April 20, 2024

1 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 $A|s$ 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.

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

1.0.1 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.

[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":

> The key difference between the two learning rules is that Q-learning is a model of operant conditioning (reinforcement learning), while R-W is a model of classical conditioning.
>
> Q-learning works by estimating the Q-value, which is the expected future reward of taking action a in state s. * $Q(a|s)$: The current Q-value estimate before the update. * η: The learning rate. * R_t: The immediate reward after taking action a in state s at time t. * $Q^{new}(a|s)$: The updated Q-value estimate.

Rescorla-Wagner learning works by updating the strength of S-R association based on prediction error. * V_T: The current S-R association strength before the update. * η: The learning rate. * R_T: The reward the agent is trying to associate with the stimulus. * V_{t+1}: The updated S-R association strength.

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":

TD(λ) learning works by updating the value of a state based on prediction error (the difference reward received and the expected reward), scaled by the eligibility trace. * V_T: The current estimate of the value of the state at time T. * η: The learning rate. * $E(\lambda)$: The eligibility trace associated with the state. * R_T: The immediate reward received at time T. * γ: The discount factor.

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 Bellman equations. * $Q(A1, S1) = P(S2|S1, A1)V(S2) + P(S3|S1, A1)V(S3)$ * $Q(A1, S1) = P(S2|S1, A2)V(S2) + P(S3|S1, A2)V(S3)$

Where: * $V(S2) = \max(Q(A1, S2), Q(A2, S2))$ * $V(S3) = \max(Q(A1, S3), Q(A2, S3))$

The greedy policy is to choose $A1$ if $Q(A1, S1) > Q(A1, S1)$ and $A2$ otherwise. You can also sample proportial to the Q-values.

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

- Q-learning and R-W learning both have one hyperparameters: the learning rate η.
- TD(0) has two hyperparameters: the learning rate η and discount factor γ.

- TD(λ) has three hyperparameters: the learning rate η, discount factor γ, and 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 $e(a \mid S)$ are updated for each (S, a) pair:

All $Q(a|S)$ 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.

[56]:
```
from scipy.special import softmax

def rl_nll(params, state, choice1, choice2, money, model_based=False):
    """Computes the negative log likelihood of the data.

    Args:
        state: Array <int32>[n_trials] of intermediate states (S2 or S3)␣
↪following choice 1.
            States are zero indexed, so 1 corresponds to S1 and 2 corresponds␣
↪to S3.
            Missed trials will have a value of -1.
        choice1: Array <int32>[n_trials] of choices following the initial state␣
↪S1.
            Actions are binary, 0 corresponds to left and 1 corresponds to␣
↪right.
            Missed trials will have a value of -1.
        choice2: Array <int32>[n_trials] of choices following the intermediate␣
↪states S2 or S3.
            Actions are binary, 0 corresponds to left and 1 corresponds to␣
↪right.
            Missed trials will have a value of -1.
        money: Array <int32>[n_trials] of rewards received at the end of the␣
↪trial.
            Rewards are binary, either 0 or 1. Missed trials will have a value␣
↪of -1.
        model_based: Boolean indicating whether to use a model-based (SARSA) or
```

```
        model-free (Q-learning) algorithm.

    Returns:
        Negative log likelihood of the data given the model.
    """
    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([n_states, n_actions])
        # Every time a state is visited, its eligibility trace becomes 1; at␣
↪every subsequent time point, the eligibility trace is multiplied by a factor

        # Get your current state for the top level (S1)
        S = 0
        A = choice1[t]
        S_next = state[t]
        A_next = choice2[t]

        # Stop if trial was missed. Missed trials will have a value of -1.
        if A == -1:
            continue

        # First level choice likelihood: compute likelihood of choice at the␣
↪first state S1.
        # Your likelihood should be a softmax function.
        p_chosen = softmax(beta * Q[S])[A]
        # Update the log likelihood
        LL += np.log(p_chosen + 1e-9)

        # 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
        E[S, A] = 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
            PE = 0 + Q[S_next, A_next] - Q[S, A]
        else:
            # Implement Q-learning update for model free learning
            PE = 0 + max(Q[S]) - Q[S, A]

        # update Q values according to Q = Q + eta * prediction errror * eligibility
        Q[S, A] = Q[S, A] + eta * PE * E[S, A]

        # Get your current state for the second level (S2 or S3)
        S = state[t]
        A = choice2[t]

        # Stop if trial was missed at the second level. Missed trials will have a value of -1
        if A == -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 = softmax(beta * Q[S])[A]
        # Update your log likelihood
        LL += np.log(p_chosen + 1e-9)

        # 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
        E[S, A] = 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[S, A]
```

```
        # update Q values according to Q = Q + eta * prediction errror *␣
↪eligibility
        Q[S, A] = Q[S, A] + eta * PE * E[S, A]

    return -LL

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

[56]: 1295.684046996923

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

> Both updates are fundemental similar. The goal of both is to update the Q-value based on the observed rewards and transitions. The key difference is how each algorithm calculates the target Q-value. Q-learning is model-free while SARSA is model-based. What this means is that Q-learning does not explicitly model the transition dynamics of the environment – it updates Q-values based on state-action pairs and the rewards received. In contrast, SARSA updates action-values for state-action pairs based on transitions actually experienced, and is therefore model-based.

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

> Q-learning is off-policy while SARSA is on-policy. This is because in SARSA, Q-values are updated based on the actions actually taken by the policy being learned. In contrast, Q-learning updates the Q-value based on the action with the maximum estimated value in the next state, regardless of the action actually taken by the 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.

[70]:
```
# 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:
all_data = []
all_model_based = []
all_model_free = []
```

```
for sub_fn in sub_fns:
    data = np.load(sub_fn)
    all_data.append(data)
    c1 = data['choice1']
    c2 = data['choice2']
    s = data['state']
    m = data['money']
    minimized = minimize(rl_nll, params, (s, c1, c2, m, True), bounds=bounds)
    all_model_based.append(minimized)
    minimized = minimize(rl_nll, params, (s, c1, c2, m, False), bounds=bounds)
    all_model_free.append(minimized)
```

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?

[82]:
```
# Compute BIC for each model and subject:
n_parameters = 3
for i, (data, model_based, model_free) in enumerate(zip(
    all_data, all_model_based, all_model_free
)):
    n_trials = data['choice1'].shape[0]
    model_based_bic = -2 * model_based.fun + np.log(n_trials) * n_parameters
    model_free_bic = -2 * model_free.fun + np.log(n_trials) * n_parameters
    print(f'Subject: {i}')
    print(f'Model-based BIC: {model_based_bic:.5}')
    print(f'Model-free  BIC: {model_free_bic:.5}')
    if model_based_bic > model_free_bic:
        print('Model-based is a better fit.')
    else:
        print('Model-free is a better fit.')
    print('--')
```

```
Subject: 0
Model-based BIC: -1367.7
Model-free  BIC: -1200.8
Model-free is a better fit.
--
Subject: 1
Model-based BIC: -531.36
Model-free  BIC: -531.36
Model-based is a better fit.
--
Subject: 2
Model-based BIC: -532.22
```

```
Model-free  BIC: -1064.1
Model-based is a better fit.
--
Subject: 3
Model-based BIC: -533.77
Model-free  BIC: -533.73
Model-free is a better fit.
--
Subject: 4
Model-based BIC: -521.71
Model-free  BIC: -521.7
Model-free is a better fit.
--
```

For subjects 0, 3, and 4, model-free is a better fit. For subjects 1 and 2, model-based is a better fit.

1.0.2 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?

SARSA is likely to learn the safer, roundabout path that avoids the cliff region. This is because SARSA learns based on the actions it actually takes during exploration. Since SARSA would follow a policy that avoids the cliff, it would experience fewer negative rewards and penalties associated with falling off the cliff. Therefore, it would learn to avoid the cliff region and choose the safer path, even if it's suboptimal in terms of reaching the goal quickly.

Q-learning, being off-policy, would learn the optimal path that leads to the goal with the least cost. This is because Q-learning updates its Q-values based on the maximum Q-value of the next state-action pair, regardless of the actions actually taken. Therefore, even if the agent occasionally falls off the cliff while exploring, Q-learning would learn that the optimal action from the state before the cliff is to move directly toward the goal, as it results in the highest expected return.

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.)

β controls the degree of exploration versus exploitation in the agent's behavior. A higher β encourages more exploration, while a lower one favors exploitation of the learned Q-values. If the environment is stable and the agent has learned accurate Q-values, it is more likely to exploit the learned knowledge and follow the path with the highest expected return. However, if the environment is dynamic or uncertain, or if the agent's

Q-values are inaccurate, it may choose to explore more to discover potentially better paths or to avoid risky regions. If the cliff region poses a significant risk (such as a high penalty for falling off the cliff), and the β is relatively low, the agent may prefer the safer, roundabout path that avoids the cliff. This is because the agent would prioritize exploitation of the learned knowledge and choose the path with the least expected cost. If the environment is stable, the cliff region poses a lower risk, or β is relatively high, the agent may prefer the optimal path that leads directly to the goal with the least cost. This is because the agent would be more inclined to explore alternative paths, even if they involve some risk, in search of higher rewards.

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

On-policy methods directly optimize the policy that is being executed, which can lead to safer and more stable learning. Real-world motor behavior often occurs in dynamic environments where conditions can change rapidly. On-policy methods are well-suited for adapting to such changes because they continuously update the policy based on the most recent experiences.

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.