Comparison of computational models for RTG

Ismail Guennouni

2025-01-24

Simple Q-Learning Model for Trust Game Behavior

This is a “model free” **Q-learning-based computational model** designed to analyze and predict the behavior of participants in a repeated trust game. Participants play the role of the trustee, interacting with varying opponents over multiple rounds. The model assumes that participants learn to make decisions (return amounts) based on the observed investment amounts from their opponents, using a reinforcement learning approach.

Key Features

State Representation

- The state is defined by the **investment amount** from the opponent, discretized into bins (e.g., low, medium, high investment).
- The number of bins is dynamically determined based on the `get_investment_bin` function, making the model adaptable to different binning schemes.

Action Space

- Participants choose a particular return amount. We can then calculate the **return proportion** (e.g., returning 0/30, 10/30, . . . , 30/30 of the tripled 10 investment).
- Return proportions are discretized into bins (e.g., 6 bins: 0-1/6, 1/6-2/6, . . . , 5/6-6/6).

Learning Mechanism

- Participants learn **Q-values** for each state-action pair, representing the expected future rewards.
- Q-values are updated using a **temporal difference (TD) learning rule**, incorporating:
  - **Immediate rewards**: Calculated based on the trustee’s payoff (Trustee assumed to only care about his/her own payoff here)
  - **Future rewards**: Discounted by a factor (`gamma`) to account for the value of future states.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Decision Making

- Participants choose actions probabilistically using a **softmax function**, which balances exploration and exploitation based on a temperature parameter (`temp`).

Model Fitting

- The model is fitted to participant data by maximizing the **log-likelihood** of observed choices under the softmax action selection rule.
- Parameters (`alpha`, `gamma`, `temp`) are estimated using optimization techniques (e.g., L-BFGS-B).

Flexibility

- The model dynamically adapts to the number of investment and return bins, making it robust to changes in the experimental design.
- It supports fitting a single set of parameters across multiple games for each participant.

Example Use Case

In a trust game experiment, participants play multiple rounds against different opponents. This model can be used to:

1. Estimate how quickly participants learn (`alpha`).
2. Measure how much they value future rewards (`gamma`).
3. Assess their tendency to explore versus exploit (`temp`).
4. Compare behavior across different experimental conditions or participant groups.

Output

The model outputs estimated parameters (`alpha`, `gamma`, `temp`) for each participant, along with model fit statistics (e.g., negative log-likelihood, AIC, BIC). These results can be used to compare participants, test hypotheses about learning mechanisms, or predict behavior in new scenarios.

Q learning with opponents states and transition

This model is a reinforcement learning (RL) model with inequality aversion and state transitions. It extends the basic Q-learning framework by incorporating Fehr-Schmidt inequality aversion and a Hidden Markov Model (HMM) inference for opponent state transitions.

Key Features of the RL + Inequality Aversion + HMM Model

State Representation

- The investor's behavior is characterized by **three hidden states**, each associated with an investment amount.
- The trustee infers the current state of the investor based on the observed investment amount (look at the closest State_Investment and assign to that state).

Action Space

- Participants choose from a set of **return amounts** (e.g., returning 0, 1, 2, . . . , up to the tripled investment amount).
- Return amounts are continuous but discretized into 6 bins for computational purposes.

Learning Mechanism

- Participants learn **Q-values** for each state, representing the expected future rewards.
- Q-values are updated using a **temporal difference (TD) learning rule**, incorporating:
  - **Immediate rewards**: Calculated using the **Fehr-Schmidt utility function**, which accounts for inequality aversion (envy and guilt).
  - **Future rewards**: Discounted by the transition probabilities between states, which depend on the trustee's return behavior.

The Q-value is updated as:

$$Q(s) \leftarrow Q(s) + \alpha \left[r_{\mathrm{FS}} + \sum_{s'} P(s'|s,a)Q(s') - Q(s) \right]$$

Where: - $Q(s)$: Current Q-value for state s. - α: Learning rate. - r_{FS}: Immediate reward calculated using the Fehr-Schmidt utility function. - $P(s'|s,a)$: Transition probability to state s' given the current state s and action a. - $\sum_{s'} P(s'|s,a)Q(s')$: Expected future reward, weighted by transition probabilities.

Assumed Investor State Transitions

- The model assumes that the investor's state transitions depend on the **net return** (return amount minus investment) provided by the trustee.
- Transition probabilities are calculated using a **sensitivity parameter**, which determines how strongly the net return affects state changes.

$$P(S_{t+1} = s'|S_t = s, \Delta) = \frac{1}{1 + e^{-\alpha \Delta}} \quad (\Delta = 3p_t I_t - I_t)$$

where p_t is the midpoint of chosen bin a_t

Decision Making

- Participants choose actions probabilistically using a **softmax function**, which balances exploration and exploitation based on a temperature parameter (`temp`).

Model Fitting

- The model is fitted to participant data by maximizing the **log-likelihood** of observed choices under the softmax action selection rule.
- Parameters (`envy`, `guilt`, `temp`, `sensitivity`, `alpha_Q`) are estimated using optimization techniques (e.g., L-BFGS-B).

Hierarchical Approach

- The model supports a **hierarchical Bayesian approach**, where parameters are constrained by literature-based priors (e.g., typical values for envy, guilt, and learning rates).

Q-Learning Update Rules: Our Approach vs. Standard MBRL

Our Approach (Hybrid TD Learning)

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[\underbrace{r}_{\text{Observed Reward}} + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[\max_{a'} Q(s',a') \right] - Q(s,a) \right]$$

- **Key Features**:
  - Uses **observed immediate reward** r from the environment.
  - Infers transitions $P(s'|s,a)$ via asumption of Hidden Markov Model (HMM) opponent.
  - No explicit reward model $R(s,a)$; rewards are stochastic samples.

Standard Model-Based RL (Slide 13)

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[\underbrace{R(s,a)}_{\text{Learned Reward Model}} + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[\max_{a'} Q(s',a') \right] - Q(s,a) \right]$$

- **Key Features**:
  - Uses a **learned reward model** $R(s,a)$, which is the expected reward for (s,a).
  - Transitions $P(s'|s,a)$ are typically learned via counts (e.g., $N(s',s,a)/N(s,a)$).
  - Reward is deterministic (model-based expectation).

```r
# Calculate transition probabilities between investor states
calculate_transition_probs <- function(current_state, investment, return_amount, sensitivity) {
    probs <- rep(0, 3)  # 3 investment bins

    # Calculate net return for investor
    net_return <- return_amount - investment

    if (current_state == 1) {
        p_up <- exp(sensitivity * net_return)
        p_stay <- 1
        norm_const <- p_up + p_stay
        probs[1:2] <- c(p_stay, p_up) / norm_const
    } else if (current_state == 3) {
        p_down <- exp(-sensitivity * net_return)
        p_stay <- 1
        norm_const <- p_down + p_stay
        probs[2:3] <- c(p_down, p_stay) / norm_const
    } else {
        p_up <- exp(sensitivity * net_return)
        p_down <- exp(-sensitivity * net_return)
        p_stay <- 1
        norm_const <- p_up + p_down + p_stay
        probs[1:3] <- c(p_down, p_stay, p_up) / norm_const
```

```
    }

    return(probs)
}

#########################################
# Core Model Function
#########################################

trustee_decision_model <- function(params, data, use_priors = FALSE) {
    # Extract parameters
    envy <- params[1]
    guilt <- params[2]
    temp_game1 <- params[3]
    temp_game2 <- params[4]
    sensitivity_game1 <- params[5]
    sensitivity_game2 <- params[6]
    alpha_Q_game1 <- params[7]
    alpha_Q_game2 <- params[8]

    # Check for missing 'gameNum.f' column
    if (!"gameNum.f" %in% names(data)) {
        stop("'gameNum.f' column is missing in the data.")
    }

    n_trials <- nrow(data)
    log_lik <- 0  # Initialize log likelihood

    # Initialize Q-values for each state-action pair (3 investment bins x 6 action bins)
    Q_values <- matrix(0, nrow = 3, ncol = 6)

    for (t in 1:n_trials) {
        investment <- data$investment[t]
        return_amount <- data$return[t]
        game <- ifelse(data$gameNum.f[t] == "first game", 1, 2)  # Map 'gameNum.f' to numeric game iden

        # Skip trials where investment is 0
        if (investment == 0) {
            # warning(paste("Investment is zero at trial", t, ": skipping."))
            next
        }

        # Calculate return proportion (based on tripled investment)
        return_prop <- return_amount / (3 * investment)

        # Validate return proportion
        if (return_prop < 0 || return_prop > 1) {
            warning(paste("Invalid return proportion at trial", t, ": skipping."))
            next
        }

        # Define current investor state (using binned investment) and action (using binned prop return)
        current_state <- get_investment_bin(investment)
```

```
current_action <- get_return_bin(return_prop)

# Use game-specific parameters
temp <- ifelse(game == 1, temp_game1, temp_game2)
sensitivity <- ifelse(game == 1, sensitivity_game1, sensitivity_game2)
alpha_Q <- ifelse(game == 1, alpha_Q_game1, alpha_Q_game2)

# Calculate utilities for all possible return bins
bin_midpoints <- seq(1 / 12, 11 / 12, length.out = 6)
possible_returns <- bin_midpoints * (3 * investment)  # Scale midpoints to the investment

utilities <- sapply(1:6, function(action_bin) {
    proposed_return <- possible_returns[action_bin]
    payoffs <- calculate_payoffs(investment, proposed_return)
    immediate_utility <- calculate_fs_utility(        # Use Fehr_Schmidt Utility Function
        payoffs$trustee, payoffs$investor, envy, guilt
    )

    # Transition probabilities
    proposed_return_prop <- bin_midpoints[action_bin]  # Use the midpoint directly
    trans_probs <- calculate_transition_probs(
        current_state, investment, proposed_return_prop, sensitivity
    )

    # Expected future value
    future_value <- sum(trans_probs * apply(Q_values, 1, max))

    return(immediate_utility + future_value)
})

# Convert utilities to probabilities using softmax
utilities_scaled <- utilities - max(utilities)
probabilities <- exp(utilities_scaled / temp)
probabilities <- probabilities / sum(probabilities)

# Ensure no zero probabilities
probabilities <- pmax(probabilities, 1e-10)
probabilities <- probabilities / sum(probabilities)

# Update log likelihood
log_lik <- log_lik + log(probabilities[current_action])

# Update Q-values if not last trial
if (t < n_trials) {
    next_investment <- data$investment[t + 1]
    next_state <- get_investment_bin(next_investment)

    # Immediate reward
    payoffs <- calculate_payoffs(investment, return_amount)
    reward <- calculate_fs_utility(payoffs$trustee, payoffs$investor, envy, guilt)

    # Update Q-value
    prediction_error <- reward + max(Q_values[next_state, ]) - Q_values[current_state, current_a
```

```
            Q_values[current_state, current_action] <- Q_values[current_state, current_action] +
                                                        alpha_Q * prediction_error
        }
    }

    # Add prior terms if using hierarchical approach
    if (use_priors) {
        priors <- get_literature_priors()
        prior_terms <- sum(dnorm(
            params,
            mean = priors$means,
            sd = priors$sds,
            log = TRUE
        ))
        return(-(log_lik + prior_terms))  # Return negative log posterior
    }

    # Return negative log-likelihood for optimization
    return(-log_lik)
}
```

Depth-of-Planning Model for Trust Game Behavior

Overview

Here we implement a computational model of strategic decision-making in repeated trust games, where a trustee infers an investor's hidden trust state and plans multi-step returns. The model formalizes how trustees might balance immediate gains against long-term relationship incentives using a partially observable Markov decision process (POMDP) framework with depth-k planning.

Depth-of-Planning Model for Repeated Trust Games

1. State Space and Observations

- **Hidden Investor States**:

$$S_t \in \{1(\text{Low}), 2(\text{Medium}), 3(\text{High})\}$$

- **Investment Emissions**:

$$P(I_t|S_t = s) = \mathcal{N}(\mu_s, \sigma^2), \quad \mu = [4, 11, 17], \sigma = 3$$

2. Trustee Action Space

- **6 Return Proportion Bins** based on ratio $r_t = \frac{\text{returned}_t}{3 \times \text{investment}_t}$:

3. Belief Updates

- **Bayesian Filtering** after observing investment I_t:

$$b_t(s) \propto P(I_t|S_t = s)b_{t-1}(s)$$

- **Assumed Investor State Transitions** after trustee action a_t:

$$P(S_{t+1} = s'|S_t = s, \Delta) = \frac{1}{1 + e^{-\alpha\Delta}} \quad (\Delta = 3p_t I_t - I_t)$$

where p_t is the midpoint of chosen bin a_t

4. Depth-k Planning

Q-value recursion for action selection:

$$Q_k(b_t, I_t, a_t) = \underbrace{3I_t(1 - p_t)}_{\text{Immediate payoff}} + \gamma \underbrace{\mathbb{E}[V_{k-1}(b_{t+1})]}_{\text{Depth-}(k-1)\text{ Value}}$$

Value function:

$$V_k(b_t) = \max_{a_t} Q_k(b_t, I_t, a_t)$$

5. Parameter Estimation

- **Initial Belief**: $b_0 \sim \text{Softmax}(\alpha_{b1}, \alpha_{b2})$
- **Transition Sensitivity**: α (logistic slope)
- **Decision Noise**: β (softmax temperature)
- **Game-Specific Parameters**: $\alpha^{(1)}, \beta^{(1)}$ (first game) vs $\alpha^{(2)}, \beta^{(2)}$ (second game)

Likelihood Function:

$$\mathcal{L}(\theta) = \prod_t P(a_t|b_t, I_t; \theta), \quad \theta = \{\alpha_{b1}, \alpha_{b2}, \alpha^{(1)}, \alpha^{(2)}, \beta^{(1)}, \beta^{(2)}\}$$

Recursive Structure

Depth Level	Calculation	Interpretation
$k = 0$	$V_0(b_t) = \max_a R(a)$	Purely myopic choices
$k = 1$	$Q_1(b_t, a) = R(a) + \gamma V_0(b_{t+1})$	1-step lookahead
$k = 2$	$Q_2(b_t, a) = R(a) + \gamma V_1(b_{t+1})$	2-step strategic planning
$\vdots$	$\vdots$	$\vdots$
$k = n$	$Q_n(b_t, a) = R(a) + \gamma V_{n-1}(b_{t+1})$	n-step forward planning

---

Model Fitting

Estimated via: - Maximum likelihood estimation (MLE) - Nested optimization over planning depths $k \in \{0, 1, 2, 3\}$ - Multi-start L-BFGS-B optimization with parameter constraints

Key Features

- **Partial Observability**: Belief state updates through Bayesian filtering
- **Strategic Depth**: Recursive planning up to $k = 3$ steps ahead
- **Adaptive Beliefs**: Trial-by-trial updating of investor state estimates
- **Individual Differences**: Participant-specific parameters for initial beliefs and decision noise

```
##############################
# 2. Emission Probability
##############################
# Three states => means = c(4, 11, 17), stdev=3
emission_prob <- function(I_obs, state, sigma=3) {
  mu <- c(4, 11, 17)
  dnorm(I_obs, mean=mu[state], sd=sigma)
}


##############################
# 6. Depth-k Q-values
##############################
# We'll do a simpler "average next invest" approach for lookahead
compute_Qvalues_k <- function(b, I_t, H, k, alpha, beta, gamma, sigma_emission=3) {
  # Purpose:
  #   Computes the *Q-values* for each of the 6 possible bins (actions)
  #   given the trustee's current belief 'b', current investor investment I_t,
  #   horizon H (rounds remaining), and planning depth k.
  #
  #   Q(S,a) represents the "value" (expected return) if the trustee chooses action a
  #   in state S at this point, under a depth-k planning approach. The trustee then picks among
  #   these Q values using softmax.


  # If k=0 => myopic:
  if (H<=0 || k<=0) {
    Q <- numeric(length(proportion_midpoints))
    for (a_idx in seq_along(proportion_midpoints)) {
      Q[a_idx] <- immediate_payoff(I_t, a_idx)
    }
    return(Q)
  }

  Q <- numeric(length(proportion_midpoints))
  mu_states <- c(4, 11, 17)  # average invests for s=1..3

  for (a_idx in seq_along(proportion_midpoints)) {
    # immediate payoff
    im_r <- immediate_payoff(I_t, a_idx)

    # next belief
    b_next <- update_belief_after_action(b, I_t, a_idx, alpha)

    # approximate next invest => sum_{s'} b_next[s']* mu[s']
    I_next <- sum(b_next * mu_states)
```

```r
    # belief after seeing I_next
    b_afterI <- update_belief_after_invest(b_next, I_next, sigma_emission)

    # recursion
    Q_next <- compute_Qvalues_k(b_afterI, I_next, H-1, k-1, alpha, beta, gamma, sigma_emission)
    V_next <- max(Q_next)

    Q[a_idx] <- im_r + gamma*V_next
  }
  Q
}

softmax <- function(qvals, beta) {
  ex <- exp(qvals/beta)
  ex / sum(ex)
}
```

```r
##############################
# 7. Neg Log-Likelihood for One Participant + k
##############################
compute_neg_log_lik_for_participant <- function(par, df_sub, k) {
  # Now 'par' has 6 elements:
  #  c(alpha_b1, alpha_b2, alpha_tr_game1, beta_game1, alpha_tr_game2, beta_game2)
  #
  # alpha_b1, alpha_b2 => define initial belief over states, shared across both games.
  # alpha_tr_game1, beta_game1 => logistic transition + softmax temp for game 1
  # alpha_tr_game2, beta_game2 => logistic transition + softmax temp for game 2
  #
  alpha_b1        <- par[1]
  alpha_b2        <- par[2]
  alpha_tr_game1  <- par[3]
  beta_game1      <- par[4]
  alpha_tr_game2  <- par[5]
  beta_game2      <- par[6]

  gamma_          <- 1.0
  sigma_emission  <- 3

  # Convert alpha_b1, alpha_b2 => initial belief distribution b_init(1..3)
  denom <- 1 + exp(alpha_b1) + exp(alpha_b2)
  b_init <- c(exp(alpha_b1)/denom,
              exp(alpha_b2)/denom,
              0)
  b_init[3] <- 1 - b_init[1] - b_init[2]

  neg_log_lik <- 0

  # We'll handle each game separately, but use game-specific alpha_tr / beta
  for (g in unique(df_sub$gameNum.f)) {
    df_game <- df_sub[df_sub$gameNum.f == g, ]
    df_game <- df_game[order(df_game$roundNum), ]
    T_game  <- nrow(df_game)
```

```r
    # Decide which alpha_tr / beta to use based on the game label
    if (g == "first game") {
      alpha_tr <- alpha_tr_game1
      beta_    <- beta_game1
    } else if (g == "second game") {
      alpha_tr <- alpha_tr_game2
      beta_    <- beta_game2
    } else {
      stop(paste("Unrecognized game label:", g))
    }

    # Reset belief to b_init at start of each game
    b_current <- b_init

    # Now loop over rounds for that game
    for (t in seq_len(T_game)) {
      I_t <- df_game$investment[t]

      # 1) Belief update after seeing investor's investment I_t
      b_current <- update_belief_after_invest(b_current, I_t, sigma_emission)

      # 2) Depth-k planning => Q-values for all 6 bins
      H_left <- T_game - t + 1
      Qvals <- compute_Qvalues_k(b_current, I_t, H_left, k,
                                 alpha_tr, beta_, gamma_, sigma_emission)
      p_actions <- softmax(Qvals, beta_)

      # 3) Observed bin in data
      a_obs <- df_game$return_bin[t]

      # If I_t=0 => we expect a_obs=1. If not, small probability => penalize LL
      if (I_t == 0 && a_obs != 1) {
        neg_log_lik <- neg_log_lik - log(1e-12)
      } else {
        p_chosen <- if (a_obs >=1 && a_obs <=6) p_actions[a_obs] else 1e-12
        if (p_chosen < 1e-12) p_chosen <- 1e-12
        neg_log_lik <- neg_log_lik - log(p_chosen)
      }

      # 4) Update belief after the chosen action
      b_current <- update_belief_after_action(b_current, I_t, a_obs, alpha_tr)
    }
  }

  neg_log_lik
}
```

To do:

- Fit a hybrid model-free/model-based RL with a weight parameter w (a la Daw two step task example)
- Model comparison: Look at goodness of fit through generating new predictions for a held-out test set (maybe last 5 rounds of each 25 round game) and compute various error metrics.

- Planning without belief states (Observed planning model)
- Improve the fitting, using Hierarchal fitting procedure consistently for all model parameters (Currently only model based RL does that). Also, currently a lot of params are clustered around the bounds, maybe use Hierarchical Bayesian Models with Stan (potential project for msc student?)-. (Challenge, stan required differentiable code, but we have discrete recursion in Compute_Q_vales_k for planning. . .)