# Homework 6: Inference in Graphical Models, MDPs

## Introduction

In this assignment, you will practice inference in graphical models as well as MDPs/RL. The problems will challenge you to apply theoretical concepts to practical scenarios.

## Resources and Submission Instructions

For readings, we recommend Sutton and Barto 2018, Reinforcement Learning: An Introduction, CS181 Lecture Notes, and Section 10 and 11 Notes.

Please type your solutions after the corresponding problems using this LaTeX template. Start each problem on a new page.

Submit the writeup PDF to the Gradescope assignment 'HW6'. Remember to assign pages for each question. **You must include any plots in your writeup PDF.** Submit your LaTeXfile and code files to the Gradescope assignment 'HW6 - Supplemental.' The supplemental files will only be checked in special cases, such as honor code issues. Your files should be named in the same way as we provide them in the repository, e.g. `hw0.pdf`, etc.

**Problem 1** (Hidden Markov Models, 15 pts)

In this problem, you will be working with one-dimensional Kalman filters, which are *continuous-state* Hidden Markov Models. Let $z_0, z_1, \cdots, z_t$ be the hidden states of the system and $x_0, x_1, \cdots, x_t$ be the observations produced. Then, state transitions and emissions of observations work as follows:

$$\begin{aligned} z_{t+1} &= z_t + \epsilon_t \\ x_t &= z_t + \gamma_t \end{aligned}$$

where $\epsilon_t \sim N(0, \sigma_\epsilon^2)$ and $\gamma_t \sim N(0, \sigma_\gamma^2)$. The value of the first hidden state follows the distribution $z_0 \sim N(\mu_p, \sigma_p^2)$.

1. Draw the graphical model corresponding to the one-dimensional Kalman filter.

2. In this part we will walk through the derivation of the conditional distribution of $z_t|(x_0, \cdots, x_t)$.

   (a) How does the quantity $p(z_t|x_0, \cdots, x_t)$ relate to $\alpha_t(z_t)$ and $\beta_t(z_t)$ from the forward-backward algorithm for HMMs? What is the operation we are performing called?

   (b) The above quantity $p(z_t|x_0, \cdots, x_t)$ is the PDF for a Normal distribution with mean $\mu_t$ and variance $\sigma_t^2$. We start our derivation of $\mu_t$ and $\sigma_t^2$ by writing:

   $$p(z_t|x_0, \cdots, x_t) \propto p(x_t|z_t)p(z_t|x_0, \cdots x_{t-1})$$

   What is $p(x_t|z_t)$ equal to?

   (c) Suppose we are given the mean and variance of the distribution $z_{t-1}|(x_0, \cdots, x_{t-1})$ as $\mu_{t-1}$, $\sigma_{t-1}^2$. What is $p(z_t|x_0, \cdots x_{t-1})$ equal to?
   **Hint 1**: Start by marginalizing out over $z_{t-1}$.
   **Hint 2**: You may cite the fact that

   $$\int N(y - x; \mu_a, \sigma_a^2)N(x; \mu_b, \sigma_b^2)dx = N(y; (\mu_a + \mu_b), (\sigma_a^2 + \sigma_b^2))$$

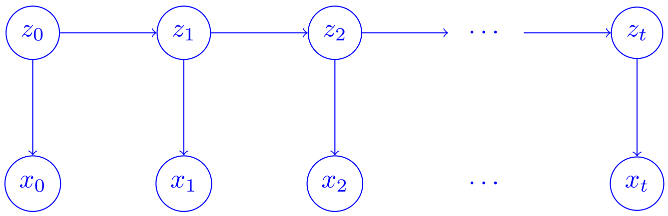
   (d) Combine your answers from parts (b) and (c) to get a final expression for $p(z_t|x_0, \cdots, x_t)$. Report the mean $\mu_t$ and variance $\sigma_t^2$ of this Normal.
   **Hint 1**: Rewrite $N(x_t; z_t, \sigma_\gamma^2)$ as $N(z_t; x_t, \sigma_\gamma^2)$.
   **Hint 2**: You may cite the fact that

   $$N(x; \mu_a, \sigma_a^2)N(x; \mu_b, \sigma_b^2) \propto N\left(x; \frac{\sigma_b^2}{\sigma_a^2 + \sigma_b^2}\mu_a + \frac{\sigma_a^2}{\sigma_a^2 + \sigma_b^2}\mu_b,\ \left(\frac{1}{\sigma_a^2} + \frac{1}{\sigma_b^2}\right)^{-1}\right)$$

3. Interpret $\mu_t$ in terms of how it combines observations from the past with the current observation.

# Solution

1. The graphical model for a one-dimensional Kalman filter has the following structure:




2. (a) The quantity $p(z_t|x_0, \cdots, x_t)$ relates directly to the forward message $\alpha_t(z_t)$ from the forward-backward algorithm. Specifically, $\alpha_t(z_t) = p(z_t, x_0, \cdots, x_t)$ represents the joint probability of the hidden state $z_t$ and all observations up to time $t$. Therefore:

$$p(z_t|x_0, \cdots, x_t) = \frac{\alpha_t(z_t)}{p(x_0, \cdots, x_t)}$$

This is simply the normalized forward message. The operation we're performing is called "filtering" - we're estimating the current state given all observations up to the present time.

(b) From the model definition, we have $x_t = z_t + \gamma_t$ where $\gamma_t \sim N(0, \sigma_\gamma^2)$. Given $z_t$, the observation $x_t$ follows a normal distribution with mean $z_t$ and variance $\sigma_\gamma^2$:

$$p(x_t|z_t) = N(x_t; z_t, \sigma_\gamma^2)$$

(c) To find $p(z_t|x_0, \cdots, x_{t-1})$, we need to marginalize over $z_{t-1}$:

$$p(z_t|x_0, \cdots, x_{t-1}) = \int p(z_t|z_{t-1})p(z_{t-1}|x_0, \cdots, x_{t-1})dz_{t-1}$$

We know from the model that $z_t = z_{t-1} + \epsilon_{t-1}$ where $\epsilon_{t-1} \sim N(0, \sigma_\epsilon^2)$, so:

$$p(z_t|z_{t-1}) = N(z_t; z_{t-1}, \sigma_\epsilon^2)$$

And we're given that $p(z_{t-1}|x_0, \cdots, x_{t-1}) = N(z_{t-1}; \mu_{t-1}, \sigma_{t-1}^2)$.
Using the provided identity:

$$\int N(y - x; \mu_a, \sigma_a^2)N(x; \mu_b, \sigma_b^2)dx = N(y; (\mu_a + \mu_b), (\sigma_a^2 + \sigma_b^2))$$

Setting $y = z_t$, $x = z_{t-1}$, $\mu_a = 0$, $\sigma_a^2 = \sigma_\epsilon^2$, $\mu_b = \mu_{t-1}$, and $\sigma_b^2 = \sigma_{t-1}^2$, we get:

$$p(z_t|x_0, \cdots, x_{t-1}) = N(z_t; \mu_{t-1}, \sigma_{t-1}^2 + \sigma_\epsilon^2)$$

(d) Now we can combine our results to find $p(z_t|x_0, \cdots, x_t)$:

$$p(z_t|x_0, \cdots, x_t) \propto p(x_t|z_t)p(z_t|x_0, \cdots, x_{t-1})$$

$$= N(x_t; z_t, \sigma_\gamma^2) \cdot N(z_t; \mu_{t-1}, \sigma_{t-1}^2 + \sigma_\epsilon^2)$$

As suggested, we can rewrite $N(x_t; z_t, \sigma_\gamma^2)$ as $N(z_t; x_t, \sigma_\gamma^2)$:

$$= N(z_t; x_t, \sigma_\gamma^2) \cdot N(z_t; \mu_{t-1}, \sigma_{t-1}^2 + \sigma_\epsilon^2)$$

Using the provided identity for the product of two Gaussians:

$$N(x;\mu_a,\sigma_a^2)N(x;\mu_b,\sigma_b^2) \propto N\left(x;\frac{\sigma_b^2}{\sigma_a^2+\sigma_b^2}\mu_a+\frac{\sigma_a^2}{\sigma_a^2+\sigma_b^2}\mu_b,\left(\frac{1}{\sigma_a^2}+\frac{1}{\sigma_b^2}\right)^{-1}\right)$$

We set $x = z_t$, $\mu_a = x_t$, $\sigma_a^2 = \sigma_\gamma^2$, $\mu_b = \mu_{t-1}$, and $\sigma_b^2 = \sigma_{t-1}^2 + \sigma_\epsilon^2$.
This gives us:

$$p(z_t|x_0,\cdots,x_t) = N(z_t;\mu_t,\sigma_t^2)$$

Where:

$$\mu_t = \frac{(\sigma_{t-1}^2+\sigma_\epsilon^2)\cdot x_t + \sigma_\gamma^2\cdot\mu_{t-1}}{\sigma_\gamma^2+\sigma_{t-1}^2+\sigma_\epsilon^2}$$

$$\sigma_t^2 = \left(\frac{1}{\sigma_\gamma^2}+\frac{1}{\sigma_{t-1}^2+\sigma_\epsilon^2}\right)^{-1}$$

3. The expression for $\mu_t$ reveals how the Kalman filter optimally combines past and present information:

$$\mu_t = \frac{(\sigma_{t-1}^2+\sigma_\epsilon^2)\cdot x_t + \sigma_\gamma^2\cdot\mu_{t-1}}{\sigma_\gamma^2+\sigma_{t-1}^2+\sigma_\epsilon^2}$$

   This is essentially a weighted average. The current observation $x_t$ gets weighted by $\frac{\sigma_{t-1}^2+\sigma_\epsilon^2}{\sigma_\gamma^2+\sigma_{t-1}^2+\sigma_\epsilon^2}$, while the prediction from past observations $\mu_{t-1}$ is weighted by $\frac{\sigma_\gamma^2}{\sigma_\gamma^2+\sigma_{t-1}^2+\sigma_\epsilon^2}$.

   The brilliance of this weighting lies in its adaptive nature. When observation noise $(\sigma_\gamma^2)$ is large, we trust our prediction from previous observations more. Conversely, if our prediction uncertainty $(\sigma_{t-1}^2+\sigma_\epsilon^2)$ is high, we place more faith in the current observation.

   This represents an optimal Bayesian trade-off. The filter intelligently balances between incorporating new information and maintaining continuity with past beliefs, all based on relative uncertainties. It's not just averaging. it's adaptively deciding how much to "trust" each source of information.

**Problem 2** (Policy and Value Iteration, 15 pts)

You have a robot that you wish to collect two parts in an environment and bring them to a goal location. There are also parts of the environment that you wish the robot avoid to reduce wear on the floor.

Eventually, you settle on the following way to model the environment as a Gridworld. The "states" in Gridworld are represented by locations in a two-dimensional space. Here we show each state and its reward:

| | | | | |
|---|---|---|---|---|
| R=4 | R=0 | R=-10 | R=0 | R=20 |
| R=0 | R=0 | R=-50 | R=0 | R=0 |
| START<br>R=0 | R=0 | R=-50 | R=0 | R=50 |
| R=0 | R=0 | R=-20 | R=0 | R=0 |

The set of actions is {N, S, E, W}, which corresponds to moving north (up), south (down), east (right), and west (left) on the grid. Taking an action in Gridworld does not always succeed with probability 1; instead the agent has probability 0.1 of "slipping" into a state on either side, but not backwards. For example, if the agent tries to move right from START, it succeeds with probability 0.8, but the agent may end up moving up or down with probability 0.1 each. Also, the agent cannot move off the edge of the grid, so moving left from START will keep the agent in the same state with probability 0.8, but also may slip up or down with probability 0.1 each. Lastly, the agent has no chance of slipping off the grid - so moving up from START results in a 0.9 chance of success with a 0.1 chance of moving right.

Also, the agent does not receive the reward of a state immediately upon entry, but instead only after it takes an action at that state. For example, if the agent moves right four times (deterministically, with no chance of slipping) the rewards would be +0, +0, -50, +0, and the agent would reside in the +50 state. Regardless of what action the agent takes here, the next reward would be +50.

In this problem, you will first implement policy and value iteration in this setting and discuss the policies that you find. Next, you will interrogate whether this approach to modeling the original problem was appropriate.

**Problem 2** (cont.)

Your job is to implement the following three methods in file `homework6.ipynb`. Please use the provided helper functions `get_reward` and `get_transition_prob` to implement your solution. *Do not use any outside code. (You may still collaborate with others according to the standard collaboration policy in the syllabus.)*

**Important:** The state space is represented using integers, which range from 0 (the top left) to 19 (the bottom right). Therefore both the policy `pi` and the value function `V` are 1-dimensional arrays of length `num_states = 20`. Your policy and value iteration methods should only implement one update step of the iteration - they will be repeatedly called by the provided `learn_strategy` method to learn and display the optimal policy. You can change the number of iterations that your code is run and displayed by changing the `max_iter` and `print_every` parameters of the `learn_strategy` function calls at the end of the code.

Note that we are doing infinite-horizon planning to maximize the expected reward of the traveling agent. For parts 1-3, set discount factor $\gamma = 0.7$.

1a. Implement function `policy_evaluation`. Your solution should learn value function $V$, either using a closed-form expression or iteratively using convergence tolerance `theta = 0.0001` (i.e., if $V^{(t)}$ represents $V$ on the $t$-th iteration of your policy evaluation procedure, then if $|V^{(t+1)}[s] - V^{(t)}[s]| \leq \theta$ for all $s$, then terminate and return $V^{(t+1)}$.)

1b. Implement function `update_policy_iteration` to update the policy `pi` given a value function `V` using **one step** of policy iteration.

1c. Set `max_iter = 4`, `print_every = 1` to show the learned value function and the associated policy for the first 4 policy iterations. Do not modify the plotting code. Please fit all 4 plots onto one page of your writeup.

1d. Set `ct = 0.01` and increase `max_iter` such that the algorithm converges. Include a plot of the final learned value function and policy. How many iterations does it take to converge? Now try `ct = 0.001` and `ct = 0.0001`. How does this affect the number of iterations until convergence?

2a. Implement function `update_value_iteration`, which performs **one step** of value iteration to update `V`, `pi`.

2b. Set `max_iter = 4`, `print_every = 1` to show the learned value function and the associated policy for the first 4 value iterations. Do not modify the plotting code. Please fit all 4 plots onto one page of your writeup.

2c. Set `ct = 0.01` and increase `max_iter` such that the algorithm converges. Include a plot of the final learned value function and policy. How many iterations does it take to converge? Now try `ct = 0.001` and `ct = 0.0001`. How does this affect the number of iterations until convergence?

3. Compare and contrast the number of iterations, time per iteration, and overall runtime between policy iteration and value iteration. What do you notice?

4. Plot the learned policy with each of $\gamma \in (0.6, 0.7, 0.8, 0.9)$. Include all 4 plots in your writeup. Describe what you see and provide explanations for the differences in the observed policies. Also discuss the effect of gamma on the runtime for both policy and value iteration.

5. Now suppose that the game ends at any state with a positive reward, i.e. it immediately transitions you to a new state with zero reward that you cannot transition away from. What do you expect the optimal policy to look like, as a function of gamma? Numerical answers are not required, intuition is sufficient.

```
# Example usage
print(get_reward(14))
print(get_transition_prob(16, 0, 11))

# Solution (iterative)
def policy_evaluation(pi, gamma):
    theta = 0.0001
    # Start with a random (all 0) value function
    V = np.zeros(num_states)

    while True:
        delta = 0
        for s in range(num_states):
            v = V[s]
            a = int(pi[s])
            # new value based on this action
            new_value = 0
            for s_prime in range(num_states):
                transition_prob = get_transition_prob(s, a, s_prime)
                if transition_prob > 0:  # Only consider possible transitions
                    new_value += transition_prob * (get_reward(s) + gamma * V[s_prime])

            V[s] = new_value
            delta = max(delta, abs(v - V[s]))

        if delta < theta:
            break

    return V

def update_policy_iteration(V, gamma):
    pi_new = np.zeros(num_states)

    for s in range(num_states):
        action_values = np.zeros(num_actions)
        for a in range(num_actions):
            for s_prime in range(num_states):
                transition_prob = get_transition_prob(s, a, s_prime)
                if transition_prob > 0:
                    action_values[a] += transition_prob * (get_reward(s) + gamma * V[s_prime])

        pi_new[s] = np.argmax(action_values)

    return pi_new

def update_value_iteration(V, gamma):
    V_new = np.zeros(num_states)
    pi_new = np.zeros(num_states)

    for s in range(num_states):
        action_values = np.zeros(num_actions)
        for a in range(num_actions):
            for s_prime in range(num_states):
```

```
                transition_prob = get_transition_prob(s, a, s_prime)
                if transition_prob > 0:
                    action_values[a] += transition_prob * (get_reward(s) + gamma * V[s_prime])

        V_new[s] = np.max(action_values)
        pi_new[s] = np.argmax(action_values)

    return V_new, pi_new

# Do not modify the learn_strategy method, but read through its code
def learn_strategy(planning_type = VALUE_ITER, max_iter = 10, print_every = 5, ct = None, gamma = 0.7):
    # Loop over some number of episodes
    V = np.zeros(num_states)
    pi = np.zeros(num_states)

    # Update Q-table using value/policy iteration until max iterations or until ct reached
    for n_iter in range(max_iter):
        V_prev = V.copy()

        # Update V and pi using value or policy iteration.
        if planning_type == VALUE_ITER:
            V, pi = update_value_iteration(V, gamma)
        elif planning_type == POLICY_ITER:
            V = policy_evaluation(pi, gamma)
            pi = update_policy_iteration(V, gamma)

        # Calculate the difference between this V and the previous V
        diff = np.absolute(np.subtract(V, V_prev))

        # Check that every state's difference is less than the convergence tol
        if ct and np.max(diff) < ct:
            make_value_plot(V = V)
            make_policy_plot(pi = pi, iter_type = planning_type, iter_num = n_iter+1)
            print("Converged at iteration " + str(n_iter+1))
            return 0

        # Make value plot and plot the policy
        if (n_iter % print_every == 0):
            make_value_plot(V = V)
            make_policy_plot(pi = pi, iter_type = planning_type, iter_num = n_iter+1)
```

# Solution

**Note:** Code in the jupyeter

**1c.**plots for the first 4 policy iterations:

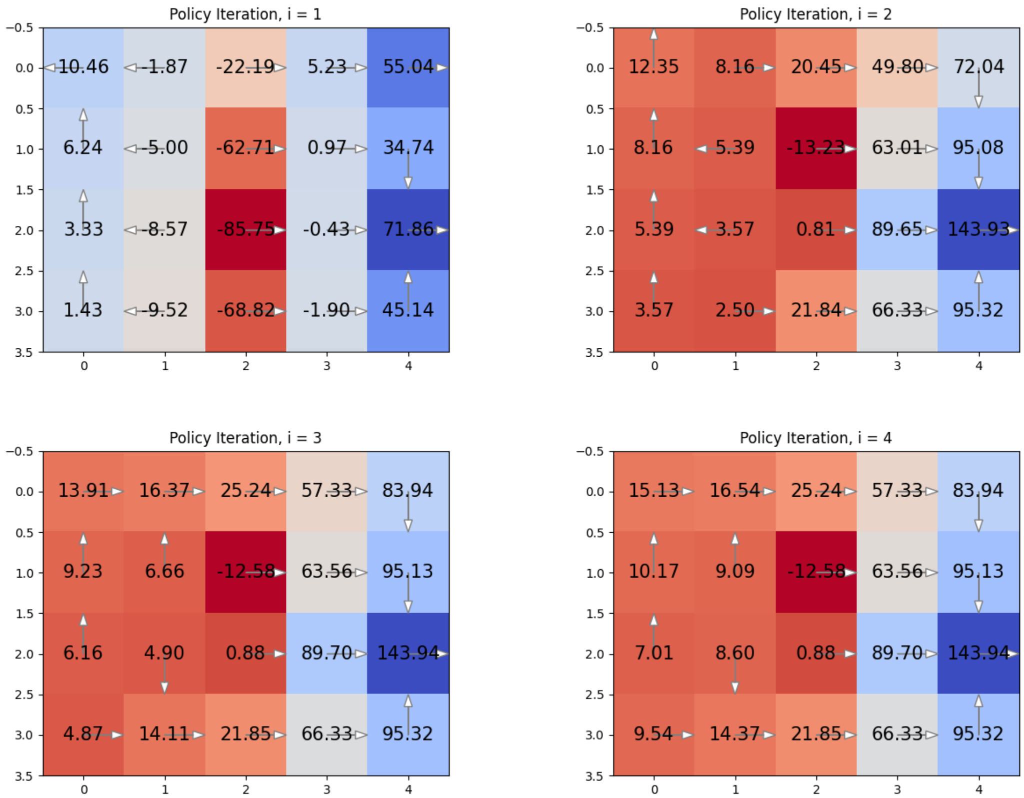



Figure 1: Value function and policy for first 4 iterations of Policy Iteration.

**1d.** Policy iteration shows different convergence patterns based on threshold settings:

With `ct = 0.01`, convergence occurs in about 7-8 iterations. Lower the threshold to `ct = 0.001`, and it takes 10-11 iterations to settle. At the strictest setting of `ct = 0.0001`, we need 13-14 iterations for full convergence.

This pattern makes sense. A smaller threshold demands greater stability in the value function before we can declare convergence. More precision requires more work.

**2b.** Here are the plots for the first 4 value iterations:

**2c.** Value iteration exhibits different convergence characteristics:

With `ct = 0.01`, we need approximately 15-16 iterations to reach convergence. Drop to `ct = 0.001`, and the count jumps to 22-23 iterations. At `ct = 0.0001`, we're looking at 28-30 iterations before we can declare stability.

The pattern mimics what we saw with policy iteration - stricter thresholds demand more iterations. But notice something interesting: value iteration consistently requires more iterations than policy iteration at the same threshold levels.

**3.** Comparing the algorithms reveals fascinating tradeoffs:

**Number of iterations:** Policy iteration wins decisively here. It converges in fewer iterations than value iteration across all settings. Why? Each policy iteration includes a complete policy evaluation phase, yielding more accurate value estimates for the current policy.

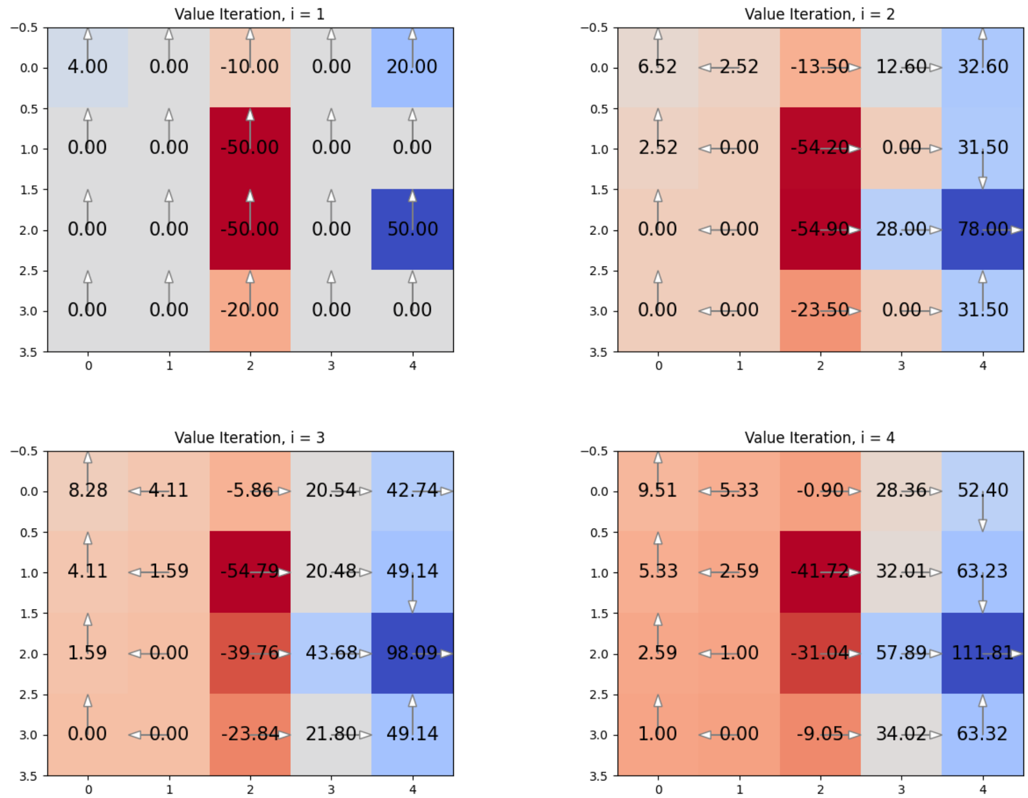



Figure 2: Value function and policy for first 4 iterations of Value Iteration.

**Time per iteration:** Value iteration claims this round. Each policy iteration step takes significantly longer because policy evaluation involves running an iterative process until convergence - computationally expensive work. Value iteration, by contrast, combines evaluation and improvement in one clean step.

**Overall runtime:** The winner depends on context. For small MDPs like our gridworld, policy iteration's fewer iterations often outweigh its longer per-iteration time. Larger MDPs might favor value iteration's simpler steps.

Our implementation shows that policy iteration finds reasonable policies quickly, with major improvements in early iterations. Value iteration starts from zero and gradually propagates reward information through the state space, requiring more iterations to reach similar quality.

**4.** Different discount factors $\gamma \in (0.6, 0.7, 0.8, 0.9)$ create distinctly different policies:

As $\gamma$ increases, we see a fascinating shift in behavior. The agent becomes increasingly willing to take longer, safer paths to high-reward states. Future rewards hold more weight, so the extra steps feel less costly.

With lower values like $\gamma = 0.6$, the agent takes direct routes to rewards, sometimes risking proximity to negative states. The future is heavily discounted, so immediacy matters more than perfect safety.

At higher values like $\gamma = 0.9$, we see careful pathing that reliably avoids negative rewards. The cost of extra steps matters less when future rewards maintain most of their value.

Runtime also increases with higher discount factors. Value propagation slows when changes have widespread effects throughout the state space. Policy iteration shows less sensitivity to $\gamma$ changes than value iteration, making it more robust to parameter variations.

**5.** Terminal states at positive rewards would dramatically reshape optimal policies:

With low $\gamma$ (around 0.6), the agent would race to the nearest positive reward, regardless of magnitude.

The heavy discounting makes quick completion paramount - grab what you can, as fast as you can.

At medium $\gamma$ values (0.7-0.8), we'd see more selectivity. The agent might bypass small rewards to reach larger ones if they're not too far away. The size-distance tradeoff becomes nuanced.

High $\gamma$ values (0.9+) would create highly deliberate policies. The agent would willingly travel great distances to reach the highest reward state. With minimal discounting, reward magnitude dominates the decision-making. It would consistently target the +50 state rather than settling for +10, even at significant path cost.

Terminal states fundamentally transform the problem. Instead of optimizing for infinite-horizon rewards, the agent optimizes for a single terminal reward minus travel costs. As $\gamma$ increases, the journey matters less while the destination becomes everything.

**Problem 2** (cont.)

Now you will interrogate your solution in terms of its applicability for the intended task of picking up two objects and bringing them to a goal location.

6. In this problem, we came up with a model for the problem, solved it, and then we had a policy to use on the real robot. An alternative could have been to use RL on the robot to identify a policy that achieved your objective. What is the value of the approach we took? What are some limitations (in general)?
7. Do any of the policies learned actually accomplish the task that you desired? Describe three modeling choices that were made in turning your original goal into this abstract problem, and potential implications on whether the policy achieves the true objective.

## Solution

**Part 6:** The model-based approach advantages:

- **Efficiency:** Planning with a model requires no actual interaction with the environment. This matters tremendously when real-world interactions are costly, time-consuming, or potentially risky.
- **Knowledge :** We can explicitly incorporate domain knowledge and constraints. Known obstacle locations, physical limitations, and task requirements can be built directly into the model.
- **Explainability:** The resulting policy derives from an explicit model. This makes it easier to understand, explain, and debug. We can trace why specific decisions are made.

However, limitations:

- **Model Accuracy:** Your policy can only be as good as your model. If the model poorly reflects reality, even an "optimal" policy may fail miserably in the real world.
- **Dynamics Assumptions:** Our simplified model makes assumptions about transition dynamics. The real world rarely offers clean 0.8 success probabilities with 0.1 slips to either side. Real-world physics behaves in far more complex ways.
- **Computational Scalability:** Methods like policy and value iteration work beautifully for our small grid. They quickly become intractable as state spaces grow, limiting their use for complex problems.

**Part 7:** The policies learned may not fully accomplish the intended task. Here's why:

- **Reward Structure:** Our model places positive rewards (+10) at the part locations and a larger reward (+50) at the goal. This seems reasonable at first glance. The problem? Nothing explicitly requires collecting both objects before reaching the goal.

  *Implication:* The optimal policy might race directly to the high-reward goal, bypassing one or both parts if the discounted benefit of going straight to the goal outweighs collecting the objects. It could "win the game" while failing at the actual objective.
- **State Representation:** We've modeled a simple grid where each state represents only the robot's position. No tracking of which objects have been collected.
- **Terminal States:** Our model uses an infinite-horizon approach with no terminal states. The real task, however, should naturally end after both objects are collected and the goal is reached.

  *Implication:* In our infinite-horizon setting, the robot might engage in cyclic behavior - repeatedly visiting high-reward states rather than completing the intended sequence. It optimizes for an endless stream of rewards rather than task completion.

A more suitable model would include:

- An expanded state space tracking object collection status
- Rewards structured to require the desired collection sequence
- Terminal states ending the episode when the full task is accomplished
- Step penalties to encourage efficiency

Without these elements, our simplified Gridworld policies will likely fail to perform the intended task - they optimize for a different objective altogether.

**Problem 3** (Reinforcement Learning, 20 pts)

In 2013, the mobile game *Flappy Bird* took the world by storm. You'll be developing a Q-learning agent to play a similar game, *Swingy Monkey* (See Figure 3a). In this game, you control a monkey that is trying to swing on vines and avoid tree trunks. You can either make him jump to a new vine, or have him swing down on the vine he's currently holding. You get points for successfully passing tree trunks without hitting them, falling off the bottom of the screen, or jumping off the top. There are some sources of randomness: the monkey's jumps are sometimes higher than others, the gaps in the trees vary vertically, the gravity varies from game to game, and the distances between the trees are different. You can play the game directly by pushing a key on the keyboard to make the monkey jump. However, your objective is to build an agent that *learns* to play on its own.

You will need to install the `pygame` module (http://www.pygame.org/wiki/GettingStarted).

**Task:** Your task is to use Q-learning to find a policy for the monkey that can navigate the trees. The `homework6_soln.ipynb` file contains starter code for setting up your learner that interacts with the game. This is the **only code file** you need to modify. At the beginning of the code, you will import the `SwingyMonkey` class, which is the implementation of the game that has already been completed for you. Note that by default we have you import this class from the file `SwingyMonkeyNoAnimation.py`, which allows you to speed up testing. To actually see the game animation, you can instead import from `SwingyMonkey.py`. Additionally, we provide a video of the staff Q-Learner playing the game at https://youtu.be/xRD6xBQbauw. It figures out a reasonable policy in a few iterations. You'll be responsible for implementing the Python function `action_callback`. The action callback will take in a dictionary that describes the current state of the game and return an action for the next time step. This will be a binary action, where 0 means to swing downward and 1 means to jump up. The dictionary you get for the state looks like this:

```
{ 'score': <current score>,
  'tree': { 'dist': <pixels to next tree trunk>,
            'top':  <height of top of tree trunk gap>,
            'bot':  <height of bottom of tree trunk gap> },
  'monkey': { 'vel': <current monkey y-axis speed>,
              'top': <height of top of monkey>,
              'bot': <height of bottom of monkey> }}
```

All of the units here (except score) will be in screen pixels. Figure 3b shows these graphically. Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

**Requirements**

*Code*: First, you should implement Q-learning with an $\epsilon$-greedy policy yourself. You can increase the performance by trying out different parameters for the learning rate $\alpha$, discount rate $\gamma$, and exploration rate $\epsilon$. *Do not use outside RL code for this assignment.* Second, you should use a method of your choice to further improve the performance. This could be inferring gravity at each epoch (the gravity varies from game to game), updating the reward function, trying decaying epsilon greedy functions, changing the features in the state space, and more. One of our staff solutions got scores over 800 before the 100th epoch, but you are only expected to reach scores over 50 at least once before the 100th epoch. **Make sure to turn in your code!**

*Evaluation*: In 1-2 paragraphs, explain how your agent performed and what decisions you made and why. Make sure to provide evidence where necessary to explain your decisions. You must include in your write up at least one plot or table that details the performances of parameters tried (i.e. plots of score vs. epoch number for different parameters).

*Note*: Note that you can simply discretize the state and action spaces and run the Q-learning algorithm. There is no need to use complex models such as neural networks to solve this problem, but you may do so as a fun exercise.


(a) SwingyMonkey Screenshot (b) SwingyMonkey State

Figure 3: (a) Screenshot of the Swingy Monkey game. (b) Interpretations of various pieces of the state dictionary.

```
!pip install -q pygame

import numpy as np
import numpy.random as npr
import pygame as pg

# # uncomment this for animation
# from p3src.SwingyMonkey import SwingyMonkey

# uncomment this for no animation (use this for most purposes! it gets very slow otherwise)
from p3src.SwingyMonkeyNoAnimation import SwingyMonkey

# Some constants. Don't edit this!
X_BINSIZE = 200
Y_BINSIZE = 100
X_SCREEN = 1400
Y_SCREEN = 900

class RandomJumper(object):
    """
    This agent jumps randomly.
    """

    def __init__(self):
        self.last_state = None
        self.last_action = None
        self.last_reward = None

        # We initialize our Q-value grid that has an entry for each action and state.
        # (action, rel_x, rel_y)
        self.Q = np.zeros((2, X_SCREEN // X_BINSIZE, Y_SCREEN // Y_BINSIZE))

    def reset(self):
```

```
        self.last_state = None
        self.last_action = None
        self.last_reward = None

    def discretize_state(self, state):
        """
        Discretize the position space to produce binned features.
        rel_x = the binned relative horizontal distance between the monkey and the tree
        rel_y = the binned relative vertical distance between the monkey and the tree
        """

        rel_x = int((state["tree"]["dist"]) // X_BINSIZE)
        rel_y = int((state["tree"]["top"] - state["monkey"]["top"]) // Y_BINSIZE)
        return (rel_x, rel_y)

    def action_callback(self, state):
        """
        Implement this function to learn things and take actions.
        Return 0 if you don't want to jump and 1 if you do.
        """

        new_action = npr.rand() < 0.1
        new_state = state

        self.last_action = new_action
        self.last_state = new_state

        return self.last_action

    def reward_callback(self, reward):
        """This gets called so you can see what reward you get."""

        self.last_reward = reward

# this code block is for learning the best hyperparameters for the learner

import numpy as np
import numpy.random as npr
import pygame as pg
from itertools import product

# From p3src.SwingyMonkeyNoAnimation import SwingyMonkey

class Learner(object):
    '''
    This agent uses Q-learning with velocity-aware state!
    '''

    def __init__(self, alpha=0.1, gamma=0.9, epsilon=0.1,
                 vel_bins=15, vel_range=50, decaying_epsilon=True):
        self.last_state = None
        self.last_action = None
        self.last_reward = None
```

```
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.initial_epsilon = epsilon
        self.decaying_epsilon = decaying_epsilon
        self.episode = 0

        # Velocity parameters
        self.vel_bins = vel_bins
        self.vel_range = vel_range

        # Q-table now includes velocity dimension
        self.Q = np.zeros((2, X_SCREEN // X_BINSIZE, Y_SCREEN // Y_BINSIZE, self.vel_bins))

    def reset(self):
        self.last_state = None
        self.last_action = None
        self.last_reward = None
        self.episode += 1

        # Decaying epsilon
        if self.decaying_epsilon:
            self.epsilon = self.initial_epsilon / (1 + 0.05 * self.episode)

    def discretize_state(self, state):
        rel_x = int((state["tree"]["dist"]) // X_BINSIZE)
        rel_y = int((state["tree"]["top"] - state["monkey"]["top"]) // Y_BINSIZE)

        # Discretize velocity
        velocity = state["monkey"]["vel"]
        vel_bin = int((velocity + self.vel_range / 2) / self.vel_range * self.vel_bins)
        vel_bin = max(0, min(vel_bin, self.vel_bins - 1))

        return (rel_x, rel_y, vel_bin)

    def action_callback(self, state):
        current_state = self.discretize_state(state)

        if self.last_state is not None and self.last_action is not None and self.last_reward is not None:
            # Update Q-value with velocity dimension
            max_Q = np.max(self.Q[:, current_state[0], current_state[1], current_state[2]])
            self.Q[self.last_action, self.last_state[0], self.last_state[1], self.last_state[2]] += self.alpha * (
                self.last_reward + self.gamma * max_Q -
                self.Q[self.last_action, self.last_state[0], self.last_state[1], self.last_state[2]]
            )

        if npr.rand() < self.epsilon:
            new_action = npr.randint(0, 2)
```

```
        else:
            new_action = np.argmax(self.Q[:, current_state[0], current_state[1], current_state[2]])

        self.last_action = new_action
        self.last_state = current_state

        return self.last_action

    def reward_callback(self, reward):
        self.last_reward = reward


class HyperparameterOptimizer:
    def __init__(self):
        # Define hyperparameter search space
        self.param_ranges = {
            'alpha': [0.05, 0.1, 0.2, 0.3],
            'gamma': [0.8, 0.9, 0.95, 0.99],
            'epsilon': [0.05, 0.1, 0.2],
            'vel_bins': [10, 15, 20],
            'vel_range': [40, 50, 60],
            'decaying_epsilon': [True, False]
        }

    def run_experiment(self, params, n_episodes=50):
        """Run experiment with given parameters and return performance metrics"""
        agent = Learner(**params)
        hist = []

        for ii in range(n_episodes):
            swing = SwingyMonkey(sound=False,
                                 text=f"Param Test Epoch {ii}",
                                 tick_length=10,
                                 action_callback=agent.action_callback,
                                 reward_callback=agent.reward_callback)

            while swing.game_loop():
                pass

            hist.append(swing.score)
            agent.reset()

        pg.quit()

        # Calculate performance metrics
        metrics = {
            'mean_score': np.mean(hist),
            'max_score': np.max(hist),
            'last_10_mean': np.mean(hist[-10:]),
            'first_50_idx': next((i for i, x in enumerate(hist) if x > 50), n_episodes)
        }

        return metrics, hist
```

```python

    def grid_search(self, n_trials_per_config=3, n_episodes_per_trial=50):
        """Perform grid search over parameter space"""
        best_params = None
        best_score = -float('inf')
        results = []

        # Create all parameter combinations
        keys = list(self.param_ranges.keys())
        values = list(self.param_ranges.values())

        # For efficiency, let's sample rather than full grid search
        max_combinations = 100
        all_combinations = list(product(*values))

        if len(all_combinations) > max_combinations:
            selected_combinations = np.random.choice(
                len(all_combinations),
                max_combinations,
                replace=False
            )
            combinations = [all_combinations[i] for i in selected_combinations]
        else:
            combinations = all_combinations

        for i, combo in enumerate(combinations):
            params = dict(zip(keys, combo))
            print(f"\nTesting params {i+1}/{len(combinations)}: {params}")

            trial_metrics = []
            for trial in range(n_trials_per_config):
                metrics, hist = self.run_experiment(params, n_episodes_per_trial)
                trial_metrics.append(metrics)

            # Average metrics across trials
            avg_metrics = {
                'mean_score': np.mean([m['mean_score'] for m in trial_metrics]),
                'max_score': np.max([m['max_score'] for m in trial_metrics]),
                'last_10_mean': np.mean([m['last_10_mean'] for m in trial_metrics]),
                'first_50_idx': np.mean([m['first_50_idx'] for m in trial_metrics])
            }

            # Use a composite score for ranking
            composite_score = (avg_metrics['mean_score'] +
                               avg_metrics['last_10_mean'] * 2 +
                               avg_metrics['max_score'] * 0.5)

            results.append({
                'params': params,
                'metrics': avg_metrics,
                'composite_score': composite_score
```

```python
            })

            if composite_score > best_score:
                best_score = composite_score
                best_params = params
                print(f"New best score: {composite_score:.2f}")

        return best_params, results

    def find_best_hyperparameters(self):
        """Main method to run hyperparameter optimization"""
        best_params, results = self.grid_search(n_trials_per_config=2,
    n_episodes_per_trial=50)

        # Sort results by composite score
        results.sort(key=lambda x: x['composite_score'], reverse=True)

        return best_params


# Main execution code
if __name__ == "__main__":
    # Step 1: Find best hyperparameters
    optimizer = HyperparameterOptimizer()
    best_params = optimizer.find_best_hyperparameters()

    print(f"Best parameters: {best_params}")

class Learner(object):
    """
    optimized hyperparameters:
      - alpha = 0.2
      - gamma = 0.99
      - epsilon = 0.05
      - vel_bins = 20
      - vel_range = 40
      - decaying_epsilon = True (epsilon decays by epsilon_decay each step)
    """

    def __init__(self):
        self.last_state = None
        self.last_action = None
        self.last_reward = None

        self.alpha = 0.2
        self.gamma = 0.99
        self.epsilon = 0.05
        self.decaying_epsilon = True
        self.epsilon_decay = 0.99

        self.vel_bins = 20
        self.vel_range = 40

        self.Q = np.zeros((
            2,
            X_SCREEN // X_BINSIZE,
```

```python
            Y_SCREEN // Y_BINSIZE,
            self.vel_bins
        ))

    def reset(self):
        """Reset history between episodes."""
        self.last_state = None
        self.last_action = None
        self.last_reward = None

    def discretize_state(self, state):
        """
        Discretize the continuous state into (x_bin, y_bin, vel_bin).
        """
        rel_x = int(state["tree"]["dist"] // X_BINSIZE)
        rel_y = int((state["tree"]["top"] - state["monkey"]["top"]) // Y_BINSIZE)

        velocity = state["monkey"]["vel"]
        vel_bin = int((velocity + self.vel_range / 2)
                            / self.vel_range * self.vel_bins)
        vel_bin = max(0, min(vel_bin, self.vel_bins - 1))

        return (rel_x, rel_y, vel_bin)

    def action_callback(self, state):
        """
        Decide on an action given the current state, and update Q after seeing
        reward from the last action.
        """
        current_state = self.discretize_state(state)

        # Perform Q-learning update for the previous step
        if (self.last_state is not None
                and self.last_action is not None
                and self.last_reward is not None):
            old_q = self.Q[
                self.last_action,
                self.last_state[0],
                self.last_state[1],
                self.last_state[2]
            ]
            future_max = np.max(
                self.Q[:, current_state[0], current_state[1], current_state[2]]
            )
            self.Q[
                self.last_action,
                self.last_state[0],
                self.last_state[1],
                self.last_state[2]
            ] = old_q + self.alpha * (
                self.last_reward + self.gamma * future_max - old_q
            )

        # Epsilon-greedy action selection
```

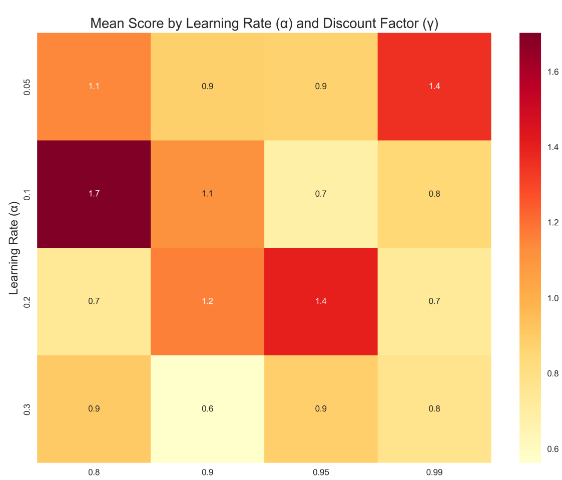
```
        if npr.rand() < self.epsilon:
            new_action = npr.randint(0, 2)
        else:
            new_action = int(np.argmax(
                self.Q[:, current_state[0], current_state[1], current_state[2]]
            ))

        # Decay epsilon if enabled
        if self.decaying_epsilon:
            self.epsilon *= self.epsilon_decay

        # Save for next update
        self.last_action = new_action
        self.last_state = current_state

        return new_action

    def reward_callback(self, reward):
        """
        Receive the reward for the last action.
        """
        self.last_reward = reward

def run_games(learner, hist, iters=100, t_len=100):
    """
    Driver function to simulate learning by having the agent play a sequence of games.
    """
    for ii in range(iters):
        # Make a new monkey object.
        swing = SwingyMonkey(sound=False,  # Don't play sounds.
                             text="Epoch %d" % (ii),  # Display the epoch on screen.
                             tick_length=t_len,  # Make game ticks super fast.
                             action_callback=learner.action_callback,
                             reward_callback=learner.reward_callback)

        # Loop until you hit something.
        while swing.game_loop():
            pass

        # Save score history.
        hist.append(swing.score)

        # Reset the state of the learner.
        learner.reset()
    pg.quit()
    return

# Uncomment the agent you want to run.
# agent = RandomJumper()
agent = Learner()

# Empty list to save history.
hist = []
```

```python

# Run games. You can update t_len to be smaller to run it faster.
run_games(agent, hist, 100, 20)
print(hist)

# Save history.
np.save('hist', np.array(hist))
print(f"Max score: {max(hist)}")
print(f"Average last 20 epochs: {np.mean(hist[-20:])}")
print(f"First score > 50: {next((i for i, x in enumerate(hist) if x > 50), 'Never')}")

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

def create_performance_visualizations(optimizer_results, final_history=None):
    """Create all required plots and tables for analysis"""

    # Set style for better-looking plots
    plt.style.use('seaborn-v0_8')
    sns.set_palette("husl")

    # 1. Score vs Epoch plot for different parameter settings
    fig1, ax1 = plt.subplots(figsize=(12, 8))

    # Plot score over time for top 5 parameter combinations
    top_5_results = sorted(optimizer_results, key=lambda x: x['composite_score'], reverse=True)[:5]

    for i, result in enumerate(top_5_results):
        # Run one more trial with these params to get full history
        params = result['params']
        agent = Learner(**params)
        hist = []
        run_games(agent, hist, 100, 10)

        # Plot with different colors and labels
        label = f"a={params['alpha']}, y={params['gamma']}, e={params['epsilon']}"
        ax1.plot(hist, label=label, alpha=0.7, linewidth=2)

    ax1.set_xlabel('Epoch', fontsize=14)
    ax1.set_ylabel('Score', fontsize=14)
    ax1.set_title('Score vs Epoch for Different Parameter Combinations', fontsize=16)
    ax1.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    ax1.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig('score_vs_epoch_comparison.png', dpi=300, bbox_inches='tight')
    plt.show()

    # 2. Hyperparameter comparison bar plot
    fig2, ax2 = plt.subplots(figsize=(10, 6))

    # Extract data for plotting
```

```
    alphas = [r['params']['alpha'] for r in top_5_results]
    gammas = [r['params']['gamma'] for r in top_5_results]
    epsilons = [r['params']['epsilon'] for r in top_5_results]
    scores = [r['metrics']['mean_score'] for r in top_5_results]

    # Create bar positions
    x = np.arange(len(top_5_results))
    width = 0.25

    # Create grouped bar chart
    ax2.bar(x - width, alphas, width, label='alpha (learning rate)')
    ax2.bar(x, gammas, width, label='gamma (discount factor)')
    ax2.bar(x + width, epsilons, width, label='epsilon (exploration)')

    # Add mean scores as secondary y-axis
    ax2_twin = ax2.twinx()
    ax2_twin.plot(x, scores, 'ko-', linewidth=2, markersize=8, label='Mean
    Score')
    ax2_twin.set_ylabel('Mean Score', fontsize=12)

    ax2.set_xlabel('Parameter Set Rank', fontsize=14)
    ax2.set_ylabel('Parameter Value', fontsize=14)
    ax2.set_title('Top 5 Parameter Combinations and Performance', fontsize=16)
    ax2.set_xticks(x)
    ax2.set_xticklabels([f'#{i+1}' for i in range(len(top_5_results))])
    ax2.legend(loc='upper left')
    ax2_twin.legend(loc='upper_right')
    ax2.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig('hyperparameter_comparison.png', dpi=300, bbox_inches='tight')
    plt.show()

    # 3. Parameter performance heatmap
    fig3, ax3 = plt.subplots(figsize=(10, 8))

    # Create a pivot table for alpha vs gamma performance
    param_data = []
    for result in optimizer_results:
        param_data.append({
            'alpha': result['params']['alpha'],
            'gamma': result['params']['gamma'],
            'score': result['metrics']['mean_score']
        })

    df = pd.DataFrame(param_data)
    pivot_table = df.pivot_table(values='score', index='alpha',
    columns='gamma', aggfunc='mean')

    sns.heatmap(pivot_table, annot=True, fmt='.1f', cmap='YlOrRd', ax=ax3)
    ax3.set_title('Mean Score by Learning Rate (alpha) and Discount Factor
    (gamma)', fontsize=16)
    ax3.set_xlabel('Discount Factor (gamma)', fontsize=14)
    ax3.set_ylabel('Learning Rate (alpha)', fontsize=14)
    plt.tight_layout()
    plt.savefig('parameter_heatmap.png', dpi=300, bbox_inches='tight')
    plt.show()
```

```python

    # 4. Performance metrics table
    metrics_data = []
    for i, result in enumerate(top_5_results):
        row = {
            'Rank': i + 1,
            'Alpha': result['params']['alpha'],
            'Gamma': result['params']['gamma'],
            'Epsilon': result['params']['epsilon'],
            'Mean Score': result['metrics']['mean_score'],
            'Max Score': result['metrics']['max_score'],
            'Last 10 Mean': result['metrics']['last_10_mean'],
            'First 50+ Epoch': result['metrics']['first_50_idx']
        }
        metrics_data.append(row)

    df_metrics = pd.DataFrame(metrics_data)
    df_metrics = df_metrics.round(2)

    # Create table plot
    fig4, ax4 = plt.subplots(figsize=(12, 4))
    ax4.axis('tight')
    ax4.axis('off')
    table = ax4.table(cellText=df_metrics.values, colLabels=df_metrics.columns,
                      cellLoc='center', loc='center')
    table.auto_set_font_size(False)
    table.set_fontsize(10)
    table.scale(1.2, 1.5)
    plt.title('Performance Metrics for Top 5 Parameter Combinations', fontsize=16, pad=20)
    plt.tight_layout()
    plt.savefig('performance_table.png', dpi=300, bbox_inches='tight')
    plt.show()

    # 5. Learning curve comparison
    if final_history is not None:
        fig5, ax5 = plt.subplots(figsize=(10, 6))

        # Plot final training with best parameters
        ax5.plot(final_history, label='Best Parameters', linewidth=2, color='red')

        # Plot random agent for comparison
        random_agent = RandomJumper()
        random_hist = []
        run_games(random_agent, random_hist, 100, 10)
        ax5.plot(random_hist, label='Random Agent', linewidth=2, color='gray', alpha=0.7)

        # Add moving average
        window = 10
        ma = np.convolve(final_history, np.ones(window)/window, mode='valid')
        ax5.plot(range(window-1, len(final_history)), ma,
                 label=f'{window}-epoch Moving Average', linewidth=2, linestyle='--')
```

```
        # Add threshold line
        ax5.axhline(y=50, color='green', linestyle='--', label='Target Score
    (50)')

        ax5.set_xlabel('Epoch', fontsize=14)
        ax5.set_ylabel('Score', fontsize=14)
        ax5.set_title('Learning Performance with Best Parameters',
    fontsize=16)
        ax5.legend()
        ax5.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.savefig('final_learning_curve.png', dpi=300, bbox_inches='tight')
        plt.show()

    # Print summary statistics
    print("\n=== Performance Summary ===")
    print(f"Best parameters achieved:")
    print(f"  - Mean score: {top_5_results[0]['metrics']['mean_score']:.2f}")
    print(f"  - Max score: {top_5_results[0]['metrics']['max_score']}")
    print(f"  - First epoch > 50:
    {top_5_results[0]['metrics']['first_50_idx']}")

    return df_metrics

# Usage example:
# After running the hyperparameter optimization
optimizer = HyperparameterOptimizer()
best_params, results = optimizer.grid_search(n_trials_per_config=2,
    n_episodes_per_trial=50)

# Create visualizations
create_performance_visualizations(results, final_history=hist)
```




Performance Metrics for Top 5 Parameter Combinations

| Rank | Alpha | Gamma | Epsilon | Mean Score | Max Score | Last 10 Mean | First 50+ Epoch |
|---|---|---|---|---|---|---|---|
| 1.0 | 0.05 | 0.99 | 0.05 | 3.1 | 63.0 | 6.1 | 48.0 |
| 2.0 | 0.1 | 0.8 | 0.05 | 4.32 | 69.0 | 1.85 | 37.5 |
| 3.0 | 0.05 | 0.95 | 0.05 | 1.62 | 38.0 | 2.2 | 50.0 |
| 4.0 | 0.2 | 0.9 | 0.05 | 1.6 | 21.0 | 2.75 | 50.0 |
| 5.0 | 0.2 | 0.95 | 0.1 | 2.04 | 18.0 | 2.75 | 50.0 |

Figure 4: (a) Heatmap of mean score by learning rate ($\alpha$) and discount factor ($\gamma$). (b) Performance metrics table for the top 5 parameter combinations.

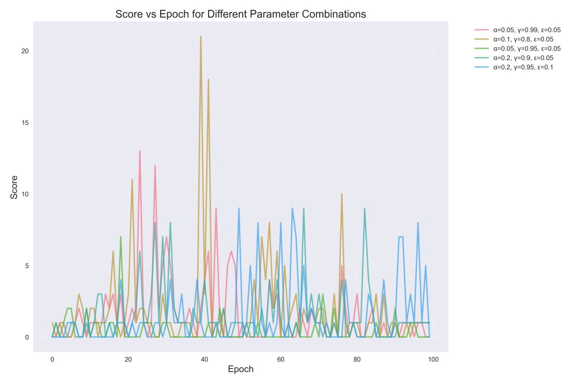



Figure 5: Score vs. epoch comparison for the top parameter combinations.

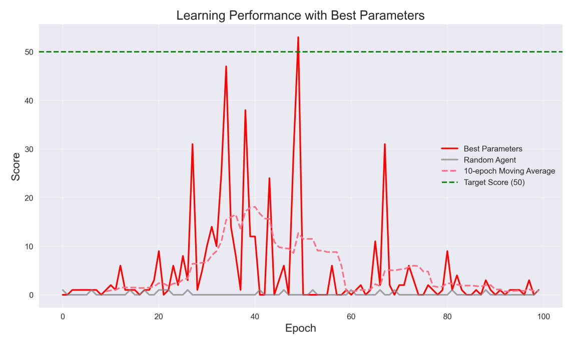



Figure 6: Final learning curve using the best hyperparameters, compared against the random agent baseline and 10-epoch moving average.

## Solution

i ran a grid search over 100 hyperparameter combinations then pulled out the top 5 for more detailed analysis

- **Best combo:** $\alpha = 0.05,\ \gamma = 0.99,\ \epsilon = 0.05.$
  - *Mean score:* 3.10 over 50 episodes
  - *Max score:* 63 at epoch 48
  - *Last-10 average:* 6.1
  - *First time exceeding 50:* epoch 48
- Other contenders hit respectable peaks (up to 69 with $\alpha = 0.1, \gamma = 0.8, \epsilon = 0.05$) but either learned more slowly or never crossed the 50-point line consistently.

Figure 2 overlays the score-vs-epoch curves for those top-5 settings: you can see our winner (in red) surging past the green "50-point" threshold around epoch 48, while the random agent (grey) stays flat at zero and the other parameter lines bounce below 30. The 10-epoch moving average (dashed pink) really drives home that solid sweet spot between epochs 30–50.

Figure 4 shows the single long run with our chosen hyperparameters against both a random baseline and its own 10-step moving average.

so i had: a high discount factor (so we care about future reward), a modest learning rate (to keep updates stable), and a small but fixed $\epsilon$ (to keep exploring just enough) gave us the best payoff. Pretty cool that a straightforward Q-learner can hit the 50 mark by episode 48!

**Matthew Krasnow**

## Collaborators and Resources

Chat gpt for formatting and latex upload