Wids-2025 End-Term Report

FlappyBird: where Flappy learns to fly

Prashant Meena (24B0949)

31/01/2026

Chapter 1: Week 1&2&3&4

In week-1, I practiced Python and its libraries, including NumPy, Pygame, and Matplotlib. In week-2, I studied Markov chains and Markov Decision Processes (MDP), focusing on state design, reward design, discretization, and environment testing.

The assignment solutions were uploaded to the GitHub repository: `https://github.com/prashantdlp/FlappyBird-AI`

Markov Chains and Markov Decision Processes

Many sequential decision-making problems can be modeled as a Markov Decision Process (MDP) under the assumption of full state observability. An MDP is defined as a 5-tuple (S, A, P, R, γ), where S is the set of states, A is the set of actions, P denotes the state transition probabilities (i.e., the consequences of actions), R is the reward function providing feedback to the agent, and $\gamma \in [0, 1]$ is the discount factor.

The discount factor γ determines the importance of future rewards. When $\gamma = 0$, the agent considers only immediate rewards, leading to greedy behavior. As γ approaches 1, future rewards are weighted more heavily, encouraging long-term planning. In episodic tasks, $\gamma = 1$ implies that future and immediate rewards are valued equally.

A bit Formal :)

At each time step t, the agent observes the current state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}$ according to a policy $\pi(a \mid s) = P(a_t = a \mid s_t = s)$. The environment then transitions to a new state s_{t+1} sampled from the transition distribution $P(\cdot \mid s_t, a_t)$ and emits a scalar reward $r_t = R(s_t, a_t)$.

The return is defined as the discounted cumulative reward:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

The value function of a policy π is given by:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right]$$

and the corresponding action-value function is:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s,\ a_0 = a\right]$$

The value function satisfies the Bellman expectation equation:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\left[R(s, a) + \gamma V^{\pi}(s')\right].$$

The optimal value function is defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s),$$

and satisfies the Bellman optimality equation:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\left[R(s, a) + \gamma V^*(s')\right].$$

The objective of reinforcement learning is to find an optimal policy π^* that maximizes the expected discounted return:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right].$$

Markov Property

A state signal that succeeds in retaining all relevant information is said to be Markov, or to have the Markov property .

Discretization

Tabular Q-learning requires a finite set of states, whereas most real-world environments are continuous. Discretization addresses this mismatch by mapping continuous variables into a finite number of bins. This introduces an inherent trade-off. Very fine discretization leads to a large Q-table, slow convergence, and sparse updates, while overly coarse discretization results in faster learning at the cost of precision and suboptimal decisions. The objective is to choose a resolution that is sufficient for effective decision-making without unnecessary complexity.

Flappy Bird Insight. In Flappy Bird, the agent does not require exact pixel-level information. Instead, decision-relevant abstractions are sufficient:

- Whether the bird is above or below the pipe gap
- The direction and magnitude of its vertical velocity
- Whether the next pipe is near or far

Thus, discretization should capture task-relevant structure rather than perfect environmental accuracy.

Reward Design

Reward design plays a crucial role in shaping agent behavior. The reward function should align closely with the true objective of the task while remaining simple and stable. Poorly designed rewards can lead to unintended behaviors, such as reward hacking or learning shortcuts that do not solve the intended problem. Sparse rewards make learning difficult due to limited feedback, whereas overly dense or large rewards can dominate learning and cause instability. Effective reward design balances guidance and flexibility, encouraging progress toward the goal without over-constraining the policy.

Environment Validation and Stress Testing

Before training an agent, it is essential to verify that the environment is well-defined and learnable. Random policies provide a simple yet effective sanity check, helping identify implementation bugs, impossible objectives, or poorly scaled rewards. If a random agent fails immediately, the task may be overly difficult; if it survives indefinitely, the task may be trivial. Unbounded or exploding rewards often indicate reward design errors.

Environment Stress Testing. Reinforcement learning systems can be highly sensitive to small changes in environment parameters. Stress testing evaluates robustness by varying key factors such as gravity, reward scaling, and observation noise. Agents that perform well only under a narrowly tuned configuration are brittle and lack generalization. Analyzing sensitivity to such perturbations provides insight into which environmental variables dominate the learned behavior.

Value functions

In reinforcement learning, the agent doesn't learn rules or strategies directly. Instead, it learns VALUE FUNCTIONS.

There are two important value functions :

1. State Value Function :- expected future reward when starting from state s

2. Action-value function :- expected future reward when starting from state s and taking action a

Temporal Difference (TD) Learning

Temporal Difference (TD) learning is a class of reinforcement learning methods that update value estimates incrementally, without waiting for an episode to terminate. The central idea is to learn from the discrepancy between the predicted value of a state–action pair and a better estimate obtained after observing the next transition. This discrepancy is known as the *TD error*.

In Q-learning, the TD target is defined as

$$\text{target} = r + \gamma \max_{a'} Q(s', a'),$$

where r is the immediate reward obtained after taking action a in state s, s' is the resulting next state, and $\gamma \in [0, 1]$ is the discount factor that determines the importance of future rewards. The term $\max_{a'} Q(s', a')$ represents the estimated value of acting optimally from the next state onward.

Rather than directly replacing the current estimate, the Q-value is updated gradually using a learning rate $\alpha \in (0, 1]$:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big(\text{target} - Q(s, a)\big).$$

This update moves the current estimate toward the TD target, allowing stable learning from partial experience and enabling online, step-by-step improvement.

Q-learning

Q-learning is one of the most fundamental and widely used reinforcement learning algorithms. Its update rule is given by

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\Big),$$

where α is the learning rate controlling how strongly new information overrides old estimates, and γ is the discount factor that determines the importance of future rewards.

At first glance, this update rule looks familiar because it is directly derived from Temporal Difference (TD) learning. The term

$$r + \gamma \max_{a'} Q(s', a')$$

serves as the TD target, while the difference between this target and the current estimate $Q(s, a)$ represents the TD error.

Q-learning is particularly powerful because it is *model-free*, *off-policy*, and *tabular*. It does not require any knowledge of the environment's transition dynamics or reward probabilities. Instead, it learns purely from experience in the form of observed tuples (s, a, r, s'). As long as an environment can provide states, actions, rewards, and next states, the same Q-learning algorithm can be applied without modification.

As a result, Q-learning can be used across a wide variety of environments such as

multi-armed bandits, GridWorld, FrozenLake, Taxi-v3, and even games like Flappy Bird. The environment changes, but the learning algorithm remains exactly the same—highlighting the generality and elegance of Q-learning.

Exploration vs. Exploitation

A reinforcement learning agent often faces a fundamental dilemma: should it choose the best-known action based on current knowledge, or should it try a new action to discover potentially better outcomes? This trade-off is known as the *exploration vs. exploitation* problem. Pure exploitation can cause the agent to converge to a sub-optimal policy if better actions are never tried, while pure exploration prevents convergence altogether, as the agent never commits to the best-performing actions.

ϵ-greedy Strategy

A simple and effective way to balance exploration and exploitation is the ϵ-greedy strategy. With probability ϵ, the agent selects an action uniformly at random (exploration), and with probability $1 - \epsilon$, it selects the action with the highest estimated Q-value (exploitation). To encourage efficient learning, ϵ is typically decayed over time, allowing the agent to explore more during early training and gradually shift toward exploitation as its value estimates become more accurate.

Pseudo Code Example

Algorithm 1 Q-learning for Flappy Bird

Initialize Q-table $Q(s, a)$ arbitrarily
Set learning rate α, discount factor γ
Set exploration rate ϵ and decay factor d
for each episode **do**
    Reset environment and observe initial state s
    while episode not terminated **do**
        With probability ϵ, select a random action a
        Otherwise, select $a = \arg\max_{a'} Q(s, a')$
        Execute action a
        Observe reward r and next state s'
        Update Q-value:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big)$$

        Set $s \leftarrow s'$
    end while
    Decay exploration rate: $\epsilon \leftarrow d \cdot \epsilon$
end for