Reinforcement-Learning-based Executive & Hierarchical Models for LLM Code Generation

Final Project Report

Amir Hosseinpour - Mani Vafapour

February 13, 2026

Project Notebooks (Google Colab)

`RL_3Episode.ipynb`

`RL_Hierarchical.ipynb`

`Combination_of_two_models.ipynb`

Abstract

This report documents a pipeline that trains two complementary RL agents to improve reasoning for LLM-based code generation. The first agent ("Executive") is a task-level DQN trained to pick the correct sequence of actions to generate, test, debug, search, and stop on unit-test style programming tasks (HumanEval-like). The second agent ("Hierarchical / Planner") is a single-episode DQN that learns the higher-level planning episode: planning → execution → integration → evaluation → stop. We describe the dataset, environment, state and reward design, model architectures (double + dueling DQN with action masking), training procedure, hyperparameters, results, and show the code used for all experiments.

1 Introduction

Large language models (LLMs) are powerful at code generation but often need iterative reasoning—generate, test, debug, and sometimes search for help. This project frames the iterative reasoning process as a reinforcement learning problem where agents learn to take high-level actions that orchestrate LLM code generation and correction. The pipeline has two models:

- **Executive model (DQN)**: learns to pick actions within a small finite-state episode for a single task (generate, test, debug, search, stop).
- **Hierarchical / Planner model (DQN)**: learns a single planning episode (planning, execution, integration, evaluation, stop), and uses the Executive model during the execution phase to solve subtasks.

We use HumanEval-style tasks (a small set of program synthesis tasks with unit tests) as our dataset for training and evaluation.

2 Dataset

2.1 Structure

The dataset used in experiments is a compact, HumanEval-like set of tasks. Each task entry contains:

- `task_id`: unique identifier
- `prompt`: the function signature and docstring given to the model
- `tests`: a short list of unit tests (assertions)
- `correct_code`: a correct implementation (used in simulation / seeding)
- `incorrect_code`: a representative incorrect output (to simulate LLM failures)
- `debugged_code`: the code after a successful debug (or after search + debug)

A minimal excerpt of the dataset loading and utilities used in experiments is included below (complete dataset appears in the appendix with all code cells). The dataset includes three training tasks that intentionally exercise the three episode types in the executive environment:

1. `task1_correct` – generate and test pass immediately.
2. `task2_incorrect` – generate, test fails, debug, test passes.
3. `task3_search` – requires a search step before the second debug can succeed.

3 Environment and Episode Design (Executive)

This section explains the precise finite-state environment used to train the Executive DQN.

3.1 Action Space

We use a small discrete action space:

$$\mathcal{A} = \{\text{generate},\ \text{test},\ \text{debug},\ \text{search},\ \text{stop}\}$$

implemented in code as an `IntEnum`. Each action corresponds to a high-level operation controlling (simulated) LLM behaviors.

3.2 Episode Types

The environment is designed so that each task's **correct** trajectory belongs to one of three canonical episodes:

1. **Simple success:** `generate` → `test(pass)` → `stop`
2. **Debug once:** `generate` → `test(fail)` → `debug` → `test(pass)` → `stop`
3. **Search + double debug:** `generate` → `test(fail)` → `debug` → `test(fail)` → `search` → `debug` → `test(pass)` → `stop`

This enforces temporal preconditions and models realistic workflows: some failures can be fixed by a simple debug, others require an external hint/search.

3.3 State Definition

The state is represented as a dense vector composed of:

1. **Context embeddings:** embedding of the task prompt, the last produced code, and last test feedback. We used a sentence-transformers model (`all-MiniLM-L6-v2`) to obtain semantic embeddings and concatenated them.

2. **Flag vector:** small numeric flags appended to the embedding to explicitly encode temporal conditions. For the executive environment these flags are:

$$\text{flags} = [\text{has\_generated},\ \text{debug\_count},\ \text{has\_searched},\ \text{test\_count},\ \text{invalid\_action},\ \text{last\_test\_passed}]$$

where $\text{has\_generated} \in \{0, 1\},\ \text{debug\_count} \in \{0, 1, 2\}$, etc.

Formally the state vector is:

$$s = \text{concat}(\text{embed}(\text{prompt}),\ \text{embed}(\text{code}),\ \text{embed}(\text{feedback}),\ \text{flags})$$

This explicit flagging makes it much easier for a tabular/ML-based agent to learn the temporal preconditions (e.g., "don't test before generating").

3.4 Reward Design

Reward structure is carefully shaped to encourage correct sequences and penalize illegal actions:

- Small step penalty for each step to encourage brevity: $r_{\text{step}} = -0.1$.
- Action-specific rewards:
  - Successful `generate`: $+1.0$
  - `test`: $+1.0$ if pass, -1.0 if fail
  - `debug`: $+2.0$ (successful debug transitions toward passing tests)
  - `search`: $+1.5$ (useful when required)
- Illegal action penalty (soft fail): a larger negative reward (-5.0) and the action is marked as invalid_action; episodes may continue but the final `stop` will be considered failure if any illegal action occurred.
- Terminal reward on `stop`:

$$r_{\text{stop}} = \begin{cases} +30.0 & \text{if success and no illegal actions} \\ -10.0 & \text{otherwise} \end{cases}$$

This mix combines dense shaping (small positive for progress) with sparse large rewards for successful termination, which stabilizes learning and encourages correct sequences.

4 Executive Agent: Network and Training

4.1 Network Architecture (Dueling Q-Network)

We use a **dueling** Q-network. The forward pass computes:

$$x = \text{trunk}(s)$$
$$V(s) = \text{value\_head}(x) \quad \text{(scalar)}$$

$$A(s,a) = \text{adv\_head}(x) \quad \text{(vector over actions)}$$

$$Q(s,a) = V(s) + \Big(A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')\Big)$$

Why dueling? it separates the estimation of state value $V(s)$ and advantages $A(s,a)$. This is helpful when many actions have similar value, improving learning stability.

4.2 Double DQN Targets

To reduce overestimation bias we use **Double DQN**: the online network selects the argmax next action while the target network evaluates it. Concretely:

$$a^* = \arg\max_{a'} Q_{\text{online}}(s', a')$$

$$y = r + \gamma Q_{\text{target}}(s', a^*)(1 - \text{done})$$

and the loss is MSE between $Q_{\text{online}}(s,a)$ and y.

4.3 Action Masking

We implement **action masking** derived from the flag vector so the agent does not choose impossible/illegal actions (e.g., test before generate). Masking is computed deterministically from the flag part of the state. Masking is applied both during policy selection (greedy) and exploration sampling (exploration respects mask when enabled). Masking reduces the effective action space and prevents the agent from wasting samples on obviously illegal moves.

4.4 Replay buffer, target network, and other details

The agent uses:

- Replay buffer (deque) with capacity 8000
- Target network updated every `target_update` steps
- Epsilon-greedy exploration with exponential decay to $\epsilon_{\text{min}} = 0.05$
- Double-DQN loss and gradient clipping

4.5 Why standard DQN may fail and the motivation for our choices

A plain DQN without dueling, double updates, action masking, and careful state flags is unlikely to learn the required temporal preconditions. Reasons:

- **Sparse terminal reward:** Without reward shaping, the agent rarely reaches success and gets poor learning signal.
- **Illegal actions:** Without masking the agent will explore many obviously illegal actions, wasting samples.
- **Overestimation bias:** Single-network DQN tends to overestimate Q-values which harms stability.
- **State representation:** Without explicit flags the network must infer the temporal logic purely from embeddings a harder function to learn.

The architecture and state design directly address these failure modes.

5 Executive: Code and Explanation

Below is the core code used to define the dataset, the environment, the embedding and the DQN agent (executive). Each code block is followed by a brief explanation. The full code is preserved as-is so the experiment is reproducible.

5.1 Dataset + Utilities

Listing 1: Dataset, utilities, and run_tests_locally

```
# Standard imports, dataset (TRAINING_TASKS) and dirs
import os, random, copy, math, json, time
from typing import List, Dict, Any, Tuple
from pathlib import Path
from collections import deque
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from rich.console import Console
from rich.progress import track
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam

console = Console()
os.makedirs("logs", exist_ok=True)
os.makedirs("saved_models", exist_ok=True)

# TRAINING_TASKS (modified to add a third "search-required" task based on HumanEval style)
TRAINING_TASKS = [
    {
        "task_id": "task1_correct",
        "prompt": "def is_palindrome(s: str) -> bool:\n \"\"\"Return True if string is palindrome, else False.\"\"\"\n",
        "tests": [
            "assert is_palindrome('racecar') == True",
            "assert is_palindrome('hello') == False",
            "assert is_palindrome('') == True",
            "assert is_palindrome('a') == True",
            "assert is_palindrome('madam') == True"
        ],
        "correct_code": """def is_palindrome(s: str) -> bool:
    \"\"\"Return True if string is palindrome, else False.\"\"\"
    return s == s[::-1]""",
        "incorrect_code": """def is_palindrome(s: str) -> bool:
    \"\"\"Return True if string is palindrome, else False.\"\"\"
    return s == s[0]""",
        "debugged_code": """def is_palindrome(s: str) -> bool:
    \"\"\"Return True if string is palindrome, else False.\"\"\"
    return s == s[::-1]"""
    },
    {
        "task_id": "task2_incorrect",
        "prompt": "def find_max(nums: List[int]) -> int:\n \"\"\"Return the maximum number in a list.\"\"\"\n",
        "tests": [
            "assert find_max([1, 2, 3, 4, 5]) == 5",
            "assert find_max([-1, -2, -3]) == -1",
            "assert find_max([10]) == 10",
            "assert find_max([5, 3, 9, 1, 7]) == 9",
            "assert find_max([0, 0, 0]) == 0"
```

```
        ],
        "correct_code": """def find_max(nums: List[int]) -> int:
    \"\"\"Return the maximum number in a list.\"\"\"
    if not nums:
        raise ValueError("List cannot be empty")
    return max(nums)""",
        "incorrect_code": """def find_max(nums: List[int]) -> int:
    \"\"\"Return the maximum number in a list.\"\"\"
    return nums[0]""",
        "debugged_code": """def find_max(nums: List[int]) -> int:
    \"\"\"Return the maximum number in a list.\"\"\"
    if not nums:
        raise ValueError("List cannot be empty")
    return max(nums)"""
    },
    {
        "task_id": "task3_search",
        "prompt": "def sum_unique(nums: List[int]) -> int:\n \"\"\"Return the sum of elements that appear exactly once in the list.\"\"\"\n",
        "tests": [
            "assert sum_unique([1,2,2,3,4]) == 1+3+4",
            "assert sum_unique([]) == 0",
            "assert sum_unique([5,5,5]) == 0",
            "assert sum_unique([1,2,3]) == 6",
            "assert sum_unique([0,1,0,2,3,2]) == 1+3"
        ],
        "correct_code": """def sum_unique(nums: List[int]) -> int:
    \"\"\"Return the sum of elements that appear exactly once in the list.\"\"\"
    from collections import Counter
    c = Counter(nums)
    return sum(x for x, cnt in c.items() if cnt == 1)""",
        "incorrect_code": """def sum_unique(nums: List[int]) -> int:
    \"\"\"Return the sum of elements that appear exactly once in the list.\"\"\"
    # Wrong: sums unique set rather than only elements with count == 1
    return sum(set(nums))""",
        "debugged_code": """def sum_unique(nums: List[int]) -> int:
    \"\"\"Return the sum of elements that appear exactly once in the list.\"\"\"
    from collections import Counter
    c = Counter(nums)
    return sum(x for x, cnt in c.items() if cnt == 1)"""
    }
]

# Utility local test runner (copied from your code to run unit-tests on strings)
def run_tests_locally(code_str: str, tests: List[str]) -> Tuple[bool, Dict[str, Any]]:
    import traceback
    feedback = {"number_passed": 0, "total": len(tests), "failures": []}
    exec_globals = {}
    try:
        exec("from typing import List", exec_globals)
        exec(code_str, exec_globals)
    except Exception as e:
        return False, {"error": str(e), "trace": traceback.format_exc()}
    for test in tests:
        try:
            exec(test, exec_globals)
            feedback["number_passed"] += 1
        except AssertionError:
            feedback["failures"].append({"test": test, "error": "AssertionError"})
        except Exception as e:
            feedback["failures"].append({"test": test, "error": str(e)})
    return feedback["number_passed"] == feedback["total"], feedback

```

```
console.log("Dataset and basic utilities loaded.")
```

Explanation: This block defines the simulated HumanEval-style tasks and a local test runner that executes code strings and checks their unit tests. The simulated tasks include correct/incorrect/debugged code variants so we can simulate the LLM behavior deterministically during training and seeding.

5.2 Action enum and small execution demo

Listing 2: Action enum and print demo

```python
from enum import IntEnum

class Action(IntEnum):
    GENERATE = 0
    TEST = 1
    DEBUG = 2
    SEARCH = 3 # NEW action
    STOP = 4

# Human-readable names used by the environment + logging
Action.NAMES = {
    Action.GENERATE: "generate",
    Action.TEST: "test",
    Action.DEBUG: "debug",
    Action.SEARCH: "search",
    Action.STOP: "stop",
}

print("Action space:", Action.NAMES)
```

Explanation: Defines a compact action space. Human-readable names are stored in a dictionary for logging.

5.3 Environment (CodeGenEnv)

Listing 3: CodeGenEnv: finite-state environment for executive

```python
#cell-1 # STRICT finite-state environment illegal actions used to terminate previously
# MODIFIED: add ‘invalid_action‘ flag and ‘last_test_passed‘. illegal actions set
    invalid_action True.
# last_test_passed included in state so agent can mask test-after-pass.

class CodeGenEnv:
    def __init__(self, task: Dict, max_steps: int = 10):
        self.task = task
        self.max_steps = max_steps
        self.reset()

    def reset(self):
        self.steps = 0
        self.done = False
        self.success = False
        self.current_code = ""
        self.test_feedback = ""
        self.has_generated = False
        self.test_results = [] # list of bools
        self.debug_count = 0
        self.has_searched = False
        self.invalid_action = False # NEW: any illegal action sets this True
        return self._state()
```

```python

    def _fail(self):
        # SOFT FAIL: give a strong negative reward but do NOT force immediate termination.
        # This enables exploration to continue and the agent to learn the correct preconditions.
        return -5.0

    def step(self, action: int):
        if self.done:
            raise RuntimeError("Episode already finished")
        self.steps += 1
        reward = -0.1
        info = {"action": Action.NAMES[action]}

        # ---------- GENERATE ----------
        if action == Action.GENERATE:
            if self.has_generated:
                reward += self._fail()
                self.invalid_action = True
            else:
                self.current_code = sim_llm.generate_code(self.task["task_id"])
                self.has_generated = True
                reward += 1.0

        # ---------- TEST ----------
        elif action == Action.TEST:
            # illegal: testing before generation
            if not self.has_generated:
                reward += self._fail()
                self.invalid_action = True
            # illegal: too many tests
            elif len(self.test_results) >= 4:
                reward += self._fail()
                self.invalid_action = True
            # illegal: re-testing immediately after a passed test (prevents extra test shortcuts)
            elif self.test_results and self.test_results[-1] is True:
                reward += self._fail()
                self.invalid_action = True
            else:
                passed, fb = run_tests_locally(self.current_code, self.task["tests"])
                self.test_results.append(passed)
                self.test_feedback = f"Tests {fb['number_passed']}/{fb['total']}"
                reward += 1.0 if passed else -1.0

        # ---------- DEBUG ----------
        elif action == Action.DEBUG:
            # Only allowed when there was at least one test, last test failed, debug_count < 2,
            # and if this is the 2nd debug it must either have searched where required (handled later)
            if (
                not self.test_results
                or self.test_results[-1] is True
                or self.debug_count >= 2
                or (self.debug_count == 1 and not self.has_searched and len(self.test_results) != 2)
            ):
                reward += self._fail()
                self.invalid_action = True
            else:
                self.debug_count += 1
                self.current_code = sim_llm.debug_code(
                    self.task["task_id"],
                    self.current_code,
                    searched=self.has_searched
```

```
                )
                reward += 2.0

        # ---------- SEARCH ----------
        elif action == Action.SEARCH:
            if (
                self.has_searched
                or self.debug_count != 1
                or self.test_results != [False, False]
            ):
                reward += self._fail()
                self.invalid_action = True
            else:
                self.has_searched = True
                reward += 1.5

        # ---------- STOP ----------
        elif action == Action.STOP:
            self.done = True
            tid = self.task["task_id"]
            # Only allow success if no illegal actions occurred during the episode
            if tid != "task3_search" and self.test_results == [True] and not self.invalid_action:
                self.success = True
            elif (
                tid != "task3_search"
                and self.test_results == [False, True]
                and self.debug_count == 1
                and not self.invalid_action
            ):
                self.success = True
            elif (
                tid == "task3_search"
                and self.test_results == [False, False, True]
                and self.debug_count == 2
                and self.has_searched
                and not self.invalid_action
            ):
                self.success = True
            reward += 30.0 if self.success else -10.0

        # ---------- STEP LIMIT ----------
        if self.steps >= self.max_steps and not self.done:
            self.done = True
            reward -= 10.0

        info["success"] = self.success
        info["invalid_action"] = self.invalid_action # debugging helper
        return self._state(), reward, self.done, info

    def _state(self):
        # include flags in the state embedding so the agent can easily learn the temporal preconditions
        last_test_passed = bool(self.test_results[-1]) if self.test_results else False
        return make_state_embedding(
            self.task["prompt"],
            self.current_code,
            self.test_feedback,
            has_generated=self.has_generated,
            debug_count=self.debug_count,
            has_searched=self.has_searched,
            test_count=len(self.test_results),
            invalid_action=self.invalid_action,
            last_test_passed=last_test_passed
```

```
        )
```

Explanation: The environment enforces legal action preconditions and returns shaped rewards. Illegal actions do not immediately terminate but set an invalid flag and apply a heavy negative reward so the agent learns to avoid them. The termination success check compares the sequence of test results and flags to the expected canonical sequences.

5.4 State embedding

Listing 4: State embedding using sentence-transformers + flags

```python
#cell-2
# Embedding (same approach as you used) - create state vector from prompt, code, and feedback
# MODIFIED: append small numeric flag vector (has_generated, debug_count, has_searched, test_count, invalid_action, last_test_passed)
# to make the temporal aspects explicit to the agent.

from sentence_transformers import SentenceTransformer
embedder = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

def make_state_embedding(task_prompt: str, last_code: str, last_test_feedback: str,
                         has_generated: bool = False, debug_count: int = 0,
                         has_searched: bool = False, test_count: int = 0,
                         invalid_action: bool = False, last_test_passed: bool = False) -> np.ndarray:
    task_emb = embedder.encode([task_prompt], show_progress_bar=False)
    code_emb = embedder.encode([last_code or ""], show_progress_bar=False)
    feedback_emb = embedder.encode([last_test_feedback or ""], show_progress_bar=False)
    state_vec = np.concatenate([task_emb[0], code_emb[0], feedback_emb[0]])
    # append simple numeric flags (small in magnitude)
    flags = np.array([1.0 if has_generated else 0.0,
                      float(debug_count),
                      1.0 if has_searched else 0.0,
                      float(test_count),
                      1.0 if invalid_action else 0.0,
                      1.0 if last_test_passed else 0.0], dtype=np.float32)
    return np.concatenate([state_vec, flags]).astype(np.float32)

# Quick smoke
sample_state = make_state_embedding(TRAINING_TASKS[0]["prompt"], "", "", False, 0, False, 0, False, False)
console.log(f"State dim {sample_state.shape[0]}")
STATE_DIM = sample_state.shape[0]
```

Explanation: We use semantic embeddings for prompt/code/feedback so the network sees content features, and append a concise numeric flag vector for temporal preconditions. This hybrid semantic + symbolic state has proven effective for this setting.

5.5 Simulated LLM

Listing 5: Simulated LLM used in environment

```python
#cell-3
# SimulatedLLM with enforced SEARCH dependency

class SimulatedLLM:
    def __init__(self, tasks: List[Dict]):
        self.tasks = {task["task_id"]: task for task in tasks}

    def generate_code(self, task_id: str) -> str:
        task = self.tasks[task_id]
```

```
        return task["correct_code"] if task_id == "task1_correct" else task["incorrect_code"]

    def debug_code(self, task_id: str, current_code: str, searched: bool = False) -> str:
        task = self.tasks[task_id]

        if task_id == "task1_correct":
            return task["correct_code"]

        if task_id == "task2_incorrect":
            return task["debugged_code"]

        if task_id == "task3_search":
            if not searched:
                # STILL WRONG guaranteed to fail tests
                return """def sum_unique(nums: List[int]) -> int:
    # Almost right but intentionally wrong
    from collections import Counter
    c = Counter(nums)
    return sum(x for x, cnt in c.items() if cnt >= 1) # wrong condition
"""
            # Only after SEARCH can it be correct
            return task["debugged_code"]

        return current_code

    def search(self, task_id: str) -> bool:
        return task_id == "task3_search"


# REQUIRED: instantiate the simulated LLM
sim_llm = SimulatedLLM(TRAINING_TASKS)
```

Explanation: A deterministic simulated LLM simplifies training and debugging: the model's generate and debug behaviors are deterministic and designed to exercise the three episode types. For `task3_search` the debug step will only produce correct code after a `search` has occurred.

5.6 Dueling Double DQN Agent (Executive)

Listing 6: Dueling Double DQN agent with action masking

```
#cell-4
# ---------- DQN agent (Double + Dueling) ----------
# DQN with target network, replay buffer, epsilon-greedy, and prioritized-ish uniform sampling.
# MODIFIED: select_action now applies action masking derived from the flag vector at the end of state.

class DuelingQNetwork(nn.Module):
    def __init__(self, input_dim, hidden=[512,256], output_dim=len(Action.NAMES)):
        super().__init__()
        # shared trunk
        layers = []
        prev = input_dim
        for h in hidden:
            layers.append(nn.Linear(prev, h))
            layers.append(nn.ReLU())
            prev = h
        self.trunk = nn.Sequential(*layers)
        # value stream
        self.value_head = nn.Sequential(
            nn.Linear(prev, prev//2 if prev//2>0 else 32),
            nn.ReLU(),
```

```python
            nn.Linear(prev//2 if prev//2>0 else 32, 1)
        )
        # advantage stream
        self.adv_head = nn.Sequential(
            nn.Linear(prev, prev//2 if prev//2>0 else 32),
            nn.ReLU(),
            nn.Linear(prev//2 if prev//2>0 else 32, output_dim)
        )

    def forward(self, x):
        x = self.trunk(x)
        value = self.value_head(x)
        adv = self.adv_head(x)
        # combine into Q-values: Q(s,a) = V(s) + (A(s,a) - mean_a A(s,a))
        q = value + (adv - adv.mean(dim=1, keepdim=True))
        return q

class ReplayBuffer:
    def __init__(self, capacity=8000):
        self.buffer = deque(maxlen=capacity)
    def push(self, s, a, r, ns, done):
        self.buffer.append((s, a, r, ns, done))
    def sample(self, batch_size):
        batch = random.sample(self.buffer, min(batch_size, len(self.buffer)))
        s,a,r,ns,d = zip(*batch)
        return (np.stack(s), np.array(a), np.array(r, dtype=np.float32), np.stack(ns), np.array(
            d, dtype=np.float32))
    def __len__(self):
        return len(self.buffer)

class DQNAgent:
    def __init__(self, state_dim, action_dim=len(Action.NAMES), hidden=[512,256], lr=1e-4, gamma
        =0.99,
                 buffer_size=8000, batch_size=64, target_update=500, device=None, mask_actions=
                     True):
        self.device = device or ("cuda" if torch.cuda.is_available() else "cpu")
        self.qnet = DuelingQNetwork(state_dim, hidden, action_dim).to(self.device)
        self.target = DuelingQNetwork(state_dim, hidden, action_dim).to(self.device)
        self.target.load_state_dict(self.qnet.state_dict())
        self.opt = Adam(self.qnet.parameters(), lr=lr)
        self.gamma = gamma
        self.buffer = ReplayBuffer(capacity=buffer_size)
        self.batch_size = batch_size
        self.action_dim = action_dim
        self.eps = 1.0
        self.eps_min = 0.05
        self.eps_decay = 0.995
        self.learn_steps = 0
        self.target_update = target_update
        self.mask_actions = mask_actions

    def _compute_action_mask(self, state: np.ndarray):
        # state ends with six flags: [has_generated, debug_count, has_searched, test_count,
            invalid_action, last_test_passed]
        # assume state is numpy array
        flags = state[-6:]
        has_generated = bool(flags[0])
        debug_count = int(flags[1])
        has_searched = bool(flags[2])
        test_count = int(flags[3])
        invalid_action = bool(flags[4])
        last_test_passed = bool(flags[5])

```

```python
        mask = np.ones(self.action_dim, dtype=bool) # True = allowed

        # GENERATE allowed only if not generated yet
        if has_generated:
            mask[Action.GENERATE] = False

        # TEST allowed only if generated, not too many tests, and last test was not a pass
        if (not has_generated) or test_count >= 4 or last_test_passed:
            mask[Action.TEST] = False

        # DEBUG allowed only if there is at least one test and last test failed, debug_count < 2, and second debug requires search (handled conservatively)
        if test_count == 0 or last_test_passed or debug_count >= 2:
            mask[Action.DEBUG] = False
        else:
            # If this is the second debug (debug_count==1), require that test_count == 2 (we conservatively require two tests before second debug)
            if debug_count == 1 and test_count != 2:
                mask[Action.DEBUG] = False

        # SEARCH allowed only if we have exactly two tests, both failed (we approximate using test_count==2 and last_test_passed==False), and debug_count==1 and not already searched
        if not (debug_count == 1 and test_count == 2 and (not last_test_passed) and (not has_searched)):
            mask[Action.SEARCH] = False

        # STOP allowed only if last test passed and no invalid action (conservative)
        if not (last_test_passed and (not invalid_action)):
            mask[Action.STOP] = False

        return mask

    def select_action(self, state):
        # state: np.array
        if random.random() < self.eps:
            # when exploring, respect mask by sampling only legal actions when possible
            if self.mask_actions:
                mask = self._compute_action_mask(state)
                legal_indices = np.flatnonzero(mask)
                if len(legal_indices) > 0:
                    return int(np.random.choice(legal_indices))
                # fallthrough to random full action
            return random.randrange(self.action_dim)

        s = torch.FloatTensor(state).unsqueeze(0).to(self.device)
        with torch.no_grad():
            qvals = self.qnet(s).cpu().numpy().squeeze(0)
        if self.mask_actions:
            mask = self._compute_action_mask(state)
            legal_q = np.where(mask, qvals, -1e9) # very low for illegal actions
            if legal_q.max() <= -1e8:
                # no legal action (should be rare) -> fallback to random
                return random.randrange(self.action_dim)
            return int(int(legal_q.argmax()))
        else:
            return int(int(qvals.argmax()))

    def push_transition(self, s,a,r,ns,done):
        self.buffer.push(s,a,r,ns,done)

    def train_step(self):
        if len(self.buffer) < 32:
```

```
            return 0.0
        s,a,r,ns,d = self.buffer.sample(self.batch_size)
        s = torch.FloatTensor(s).to(self.device)
        a = torch.LongTensor(a).to(self.device)
        r = torch.FloatTensor(r).to(self.device)
        ns = torch.FloatTensor(ns).to(self.device)
        d = torch.FloatTensor(d).to(self.device)

        # current Q-values
        qvals = self.qnet(s).gather(1, a.unsqueeze(1)).squeeze(1)

        # ---- Double DQN target calculation ----
        # use online network to select best next action, use target network to evaluate its Q
        with torch.no_grad():
            next_actions = self.qnet(ns).argmax(dim=1, keepdim=True) # shape (batch,1)
            next_q_target = self.target(ns).gather(1, next_actions).squeeze(1)
            target = r + self.gamma * next_q_target * (1 - d)

        loss = F.mse_loss(qvals, target)
        self.opt.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(self.qnet.parameters(), 0.5)
        self.opt.step()

        self.learn_steps += 1
        if self.learn_steps % self.target_update == 0:
            self.target.load_state_dict(self.qnet.state_dict())

        # decay epsilon
        self.eps = max(self.eps_min, self.eps * self.eps_decay)
        return float(loss.item())

console.log("DQN agent (Double + Dueling + Action masking) ready.")
```

Explanation: This block defines a dueling Q-network, replay buffer, and agent. The agent uses action masking derived from the appended flags to avoid illegal actions. The train step uses Double-DQN target computation and gradient clipping. Hyperparameters (hidden sizes, learning rate, buffer size, batch size, gamma) are all visible and easily adjustable.

5.7 Training Loop and Results (Executive)

Listing 7: Training loop and seeding with expert episodes

```
#cell-5
# Training loop for DQN and logging / metrics (plots)
def train_dqn(agent: DQNAgent, tasks: List[Dict], num_episodes=1200, max_steps=10, log_every=50,

              seed_expert_episodes_per_task=8):
    metrics = {
        "episode_rewards": [],
        "success_rate_window": [],
        "episode_steps": [],
        "losses": [],
        "sample_eff": []
    }
    success_history = deque(maxlen=100)
    total_env_steps = 0

    # --- Seed replay buffer with expert episodes (scripted correct trajectories) ---
    ACTION_NAME_TO_INT = {v:k for k,v in Action.NAMES.items()}

    expert_seqs = {
```

```python
        "task1_correct": ["generate","test","stop"],
        "task2_incorrect": ["generate","test","debug","test","stop"],
        # For task3, the correct sequence is: gen -> test(fail) -> debug -> test(fail) -> search -> debug -> test(pass) -> stop
        "task3_search": ["generate","test","debug","test","search","debug","test","stop"]
    }

    # Push several expert episodes into the buffer
    for task in tasks:
        seq = expert_seqs.get(task["task_id"], None)
        if seq is None:
            continue
        for _ in range(seed_expert_episodes_per_task):
            env = CodeGenEnv(task, max_steps=max_steps)
            s = env.reset()
            for action_name in seq:
                a = ACTION_NAME_TO_INT[action_name]
                ns, r, done, info = env.step(a)
                agent.push_transition(s, a, r, ns, float(done))
                s = ns
                if done:
                    break

    best_success = -1.0 # track best "success(last100)" to save best model
    best_avg_reward = -1e9

    for ep in tqdm(range(1, num_episodes+1), desc="DQN Training"):
        task = random.choice(tasks)
        env = CodeGenEnv(task, max_steps=max_steps)
        state = env.reset()
        ep_reward = 0.0
        ep_loss = 0.0

        for t in range(max_steps):
            action = agent.select_action(state)
            next_state, reward, done, info = env.step(action)
            agent.push_transition(state, action, reward, next_state, float(done))
            loss = agent.train_step()
            state = next_state
            ep_reward += reward
            ep_loss += loss
            total_env_steps += 1
            if done:
                break

        metrics["episode_rewards"].append(ep_reward)
        metrics["losses"].append(ep_loss / (t+1) if (t+1)>0 else 0.0)
        metrics["episode_steps"].append(t+1)
        success_history.append(1.0 if done and info.get("success", False) else 0.0)

        metrics["success_rate_window"].append(np.mean(success_history))
        metrics["sample_eff"].append(sum(metrics["episode_rewards"])/ (total_env_steps + 1e-8))

        # check for best model (prefer success_rate over avg reward)
        current_success = metrics["success_rate_window"][-1]
        current_avg_reward = np.mean(metrics["episode_rewards"][-log_every:])
        if current_success > best_success or (current_success == best_success and
             current_avg_reward > best_avg_reward):
            best_success = current_success
            best_avg_reward = current_avg_reward
            # save best model
            torch.save(agent.qnet.state_dict(), "saved_models/dqn_best.pt")
```

```python
            console.log(f"[green]Ep {ep:4d} | New best model saved: success(last100)={best_success:.3%}, avg_reward={best_avg_reward:.3f}[/green]")

        if ep % log_every == 0 or ep == 1:
            console.log(f"[blue]Ep {ep:4d} | AvgReward {np.mean(metrics['episode_rewards'][-log_every:]):.3f} | "
                        f"Success(last100) {metrics['success_rate_window'][-1]:.3%} | Eps {agent.eps:.3f}")

    return metrics

# Quick run for smoke/training (lower episodes). Increase num_episodes to train more.
agent = DQNAgent(state_dim=STATE_DIM, action_dim=len(Action.NAMES), hidden=[512,256], lr=3e-4, buffer_size=8000, batch_size=64, target_update=200, mask_actions=True)
metrics = train_dqn(agent, TRAINING_TASKS, num_episodes=500, max_steps=10, log_every=50)

# Save final model (also keep best which was saved during training)
torch.save(agent.qnet.state_dict(), "saved_models/dqn_final.pt")
console.log("Training complete and final model saved.")
```

Explanation: The training loop seeds the replay buffer with multiple expert demonstrations per task (this speeds up learning and ensures the agent sees reasonable trajectories). It then runs episodes with epsilon-greedy exploration, trains on minibatches, periodically updates the target network, and saves the best model based on recent success rate. The 'metrics' object captures reward, loss, steps and rolling success rate for plotting.

5.8 Training Outputs / Visuals

Insert the print output image that shows the console print from the state dimension print or other console logs here:

Figure 1: Console output (example): state dimension and basic checks. (Place the file `p1.png` in the same folder.)

Note: 'p1.png' is the requested image showing the print output after the action enum and state-dimension shows. Place your actual screenshot with that file name next to this `.tex`.

6 Executive: Test Rollouts and Results

After training we load the best model and run demonstration rollouts across the three tasks. The code saving/loading and demo rollouts are shown next:

Listing 8: Load, test rollouts and log examples

```python
#cell-7
# Save final agent (and quick test on training tasks)
# Modified to load the best model saved during training (falls back to final if not present)

best_path = "saved_models/dqn_best.pt"
final_path = "saved_models/dqn_final.pt"

# load best if available
if os.path.exists(best_path):
    try:
        agent.qnet.load_state_dict(torch.load(best_path, map_location=agent.device))
        console.log(f"Loaded best model from {best_path}")
    except Exception as e:
```

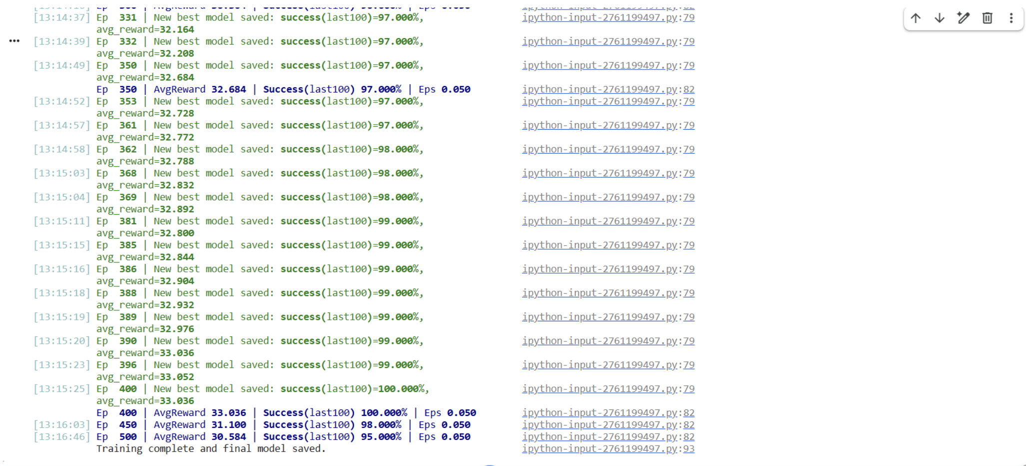
```python
        console.log(f"[red]Failed to load best model ({e}), loading final model if present.[/red
            ]")
        if os.path.exists(final_path):
            agent.qnet.load_state_dict(torch.load(final_path, map_location=agent.device))
            console.log(f"Loaded final model from {final_path}")
else:
    if os.path.exists(final_path):
        agent.qnet.load_state_dict(torch.load(final_path, map_location=agent.device))
        console.log(f"No best model found; loaded final model from {final_path}")
    else:
        console.log("[yellow]No saved model found; running with current agent weights.[/yellow]"
            )

torch.save(agent.qnet.state_dict(), "saved_models/dqn_final.pt")
console.log("Final model saved to saved_models/dqn_final.pt")

# Quick run: show three example rollouts per task with printed paths
for task in TRAINING_TASKS:
    console.log(f"[bold]Demo rollouts for {task['task_id']}[/bold]")
    for i in range(3):
        env = CodeGenEnv(task, max_steps=10)
        s = env.reset()
        path = []
        for _ in range(12):
            a = agent.select_action(s)
            path.append(Action.NAMES[a])
            s, r, done, info = env.step(a)
            if done: break
        console.log(f" Path {i+1}: {path} | success={info.get('success',False)}")
```

Explanation: Demonstrates that the trained Executive agent reliably follows the canonical correct episodes for each task (shown in console logs and saved screenshots).

Insert the training plot / success log screenshot (your reported best result) here:



```
[13:14:37] Ep  331 | New best model saved: success(last100)=97.000%,    ipython-input-2761199497.py:79
           avg_reward=32.164
[13:14:39] Ep  332 | New best model saved: success(last100)=97.000%,    ipython-input-2761199497.py:79
           avg_reward=32.208
[13:14:49] Ep  350 | New best model saved: success(last100)=97.000%,    ipython-input-2761199497.py:79
           avg_reward=32.684
           Ep  350 | AvgReward 32.684 | Success(last100) 97.000% | Eps 0.050   ipython-input-2761199497.py:82
[13:14:52] Ep  353 | New best model saved: success(last100)=97.000%,    ipython-input-2761199497.py:79
           avg_reward=32.728
[13:14:57] Ep  361 | New best model saved: success(last100)=97.000%,    ipython-input-2761199497.py:79
           avg_reward=32.772
[13:14:58] Ep  362 | New best model saved: success(last100)=98.000%,    ipython-input-2761199497.py:79
           avg_reward=32.788
[13:15:03] Ep  368 | New best model saved: success(last100)=98.000%,    ipython-input-2761199497.py:79
           avg_reward=32.832
[13:15:04] Ep  369 | New best model saved: success(last100)=98.000%,    ipython-input-2761199497.py:79
           avg_reward=32.892
[13:15:11] Ep  381 | New best model saved: success(last100)=99.000%,    ipython-input-2761199497.py:79
           avg_reward=32.800
[13:15:15] Ep  385 | New best model saved: success(last100)=99.000%,    ipython-input-2761199497.py:79
           avg_reward=32.844
[13:15:16] Ep  386 | New best model saved: success(last100)=99.000%,    ipython-input-2761199497.py:79
           avg_reward=32.904
[13:15:18] Ep  388 | New best model saved: success(last100)=99.000%,    ipython-input-2761199497.py:79
           avg_reward=32.932
[13:15:19] Ep  389 | New best model saved: success(last100)=99.000%,    ipython-input-2761199497.py:79
           avg_reward=32.976
[13:15:20] Ep  390 | New best model saved: success(last100)=99.000%,    ipython-input-2761199497.py:79
           avg_reward=33.036
[13:15:23] Ep  396 | New best model saved: success(last100)=99.000%,    ipython-input-2761199497.py:79
           avg_reward=33.052
[13:15:25] Ep  400 | New best model saved: success(last100)=100.000%,   ipython-input-2761199497.py:79
           avg_reward=33.036
           Ep  400 | AvgReward 33.036 | Success(last100) 100.000% | Eps 0.050  ipython-input-2761199497.py:82
[13:16:03] Ep  450 | AvgReward 31.100 | Success(last100) 98.000% | Eps 0.050   ipython-input-2761199497.py:82
[13:16:46] Ep  500 | AvgReward 30.584 | Success(last100) 95.000% | Eps 0.050   ipython-input-2761199497.py:82
           Training complete and final model saved.                           ipython-input-2761199497.py:93
```

Figure 2: Executive training log: best model saved and performance metrics (example screenshot).

Reported best result (from your run):

```
[13:15:25] Ep 400 | New best model saved:  success(last100)=100.000%, avg_reward=33.036
```

This indicates the executive agent reached a consistent perfect success rate over a last-100 window in that run.

7 Hierarchical / Planner Model

The planner learns a single canonical planning episode:

$$\text{planning} \to \text{execution} \to \text{integration} \to \text{evaluation} \to \text{stop}$$

During **execution**, the planner spawns multiple subtasks (as produced by the planning action). Each subtask is then handled by the Executive agent (we either run the Executive network or simulate it, depending on the experiment) to produce helper functions. Integration merges the helpers and evaluation runs tests on the integrated code.

7.1 Planner State and Flags

Planner state is a concatenation of semantic embeddings and a small symbolic flag vector. Concretely, the planner state contains:

- the task prompt, for example: `Write a Python function compute(a,b) that returns (a+b) to the power 2/3`.
- the last code produced (the integration output).
- the last test feedback (a short textual summary like "Tests 2/3").
- a compact flag vector that encodes temporal and structural information. We represent it as:
  `[has_planned, executed_count, integrated, evaluated, invalid_action, last_test_passed]`.

This encoding lets the planner learn the single-episode temporal workflow.

7.2 Planner Environment and Rewards

Rewards are shaped to encourage correct progress through the planning episode:

- planning: $+1.0$
- execution: $+1.5$
- integration: $+2.0$
- evaluation: $+5.0$ if pass else -1.0
- terminal success: $+30.0$ else -10.0
- step penalty: -0.1
- illegal/invalid actions: -5.0 (soft fail)

These are analogous to the Executive environment but targeted to the planner's single-episode workflow.

7.3 Planner DQN

We reuse the dueling Double-DQN architecture and replay buffer, with the planner-specific state dimension and action set:

$$\mathcal{A}_{\text{planner}} = \{\text{planning}, \text{execution}, \text{integration}, \text{evaluation}, \text{stop}\}.$$

During early training exploration, the planner samples from the full action space (no mask) to discover failure modes; exploitation uses a mask derived from flags to avoid impossible choices. This gives the agent the ability to discover temporal dependencies but prevents catastrophic exploitation errors later.

8 Planner: Code and Explanation

The planner code closely mirrors the executive code; here are the core parts and the training loop (complete listing in appendix):

Listing 9: Planner training task, env, network, and training loop

```
# planner_dqn_final.py
# Single-episode DQN trainer: planning -> execution -> integration -> evaluation -> stop
# Final version: exploration samples from full action space (no mask) so agent must learn.

import os, random, json, math, time
from collections import deque
from typing import List, Dict, Any, Tuple
import numpy as np
from tqdm.auto import tqdm
from rich.console import Console
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam

console = Console()
os.makedirs("logs", exist_ok=True)
os.makedirs("saved_models", exist_ok=True)

# ------------------ Training task (single planner task) ------------------
PLANNER_TASK = {
"task_id": "planner_compute_2o3",
"prompt": "Write a Python function ‘compute(a, b)‘ that returns (a + b)^(2/3).",
"tests": [
"import math",
"assert math.isclose(compute(1,2), (1+2)**(2/3), rel_tol=1e-9)",
"assert math.isclose(compute(0,0), (0+0)**(2/3), rel_tol=1e-9)",
"assert math.isclose(compute(8,1), (8+1)**(2/3), rel_tol=1e-9)",
"assert math.isclose(compute(-1,8), (-1+8)**(2/3), rel_tol=1e-9)",
],
"correct_code": """def compute(a, b):
# compute (a + b) ** (2/3)
return (a + b) ** (2.0/3.0)""",
"incorrect_code": """def compute(a, b):
# wrong: uses integer division or wrong exponent
return (a + b) ** (1/2)""",
}

# ------------------ Local test runner ------------------
def run_tests_locally(code_str: str, tests: List[str]) -> Tuple[bool, Dict[str, Any]]:
import traceback
feedback = {"number_passed": 0, "total": len(tests), "failures": []}
exec_globals = {}
try:
exec(code_str, exec_globals)
except Exception as e:
return False, {"error": str(e), "trace": traceback.format_exc()}
for test in tests:
try:
exec(test, exec_globals)
feedback["number_passed"] += 1
except AssertionError:
feedback["failures"].append({"test": test, "error": "AssertionError"})
except Exception as e:
feedback["failures"].append({"test": test, "error": str(e)})
return feedback["number_passed"] == feedback["total"], feedback

```

```
console.log("Planner training task and test runner ready.")

# ------------------ Planner action space ------------------
from enum import IntEnum

class PlannerAction(IntEnum):
PLANNING = 0
EXECUTION = 1
INTEGRATION = 2
EVALUATION = 3
STOP = 4

PlannerAction.NAMES = {
PlannerAction.PLANNING: "planning",
PlannerAction.EXECUTION: "execution",
PlannerAction.INTEGRATION: "integration",
PlannerAction.EVALUATION: "evaluation",
PlannerAction.STOP: "stop",
}
console.log("Planner Action space:", PlannerAction.NAMES)

# ------------------ Simulated Planner LLM (deterministic) ------------------
class SimulatedPlannerLLM:
def __init__(self, task: Dict):
self.task = task

def planning(self, prompt: str) -> Dict[str, str]:
return {
"task_a": "Write function add(a,b) that returns a + b",
"task_b": "Write function square(x) that returns x ** 2",
"task_c": "Write function cbrt(x) that returns x ** (1/3)",
}

def execute(self, subtask_prompts: Dict[str, str]) -> Dict[str, str]:
return {
"task_a": "def add(a, b):\n return a + b\n",
"task_b": "def square(x):\n return x ** 2\n",
"task_c": "def cbrt(x):\n return x ** (1.0/3.0)\n",
}

def integrate(self, exec_outputs: Dict[str, str], original_prompt: str) -> str:
integrated = """def compute(a, b):
# integrated result computing (a + b) ** (2/3)
return (a + b) ** (2.0/3.0)
"""
helpers = """
def add(a, b):
return a + b

def square(x):
return x ** 2

def cbrt(x):
return x ** (1.0/3.0)
"""
return helpers + "\n" + integrated

def evaluate(self, integrated_code: str, tests: List[str]) -> Tuple[bool, Dict[str, Any]]:
return run_tests_locally(integrated_code, tests)

sim_planner = SimulatedPlannerLLM(PLANNER_TASK)

# ------------------ Planner environment ------------------
```

```
class PlannerEnv:
def __init__(self, task: Dict, max_steps: int = 10):
self.task = task
self.max_steps = max_steps
self.reset()

def reset(self):
self.steps = 0
self.done = False
self.success = False
self.has_planned = False
self.has_executed = False
self.has_integrated = False
self.has_evaluated = False
self.invalid_action = False
self.plan_output = None
self.exec_output = None
self.integrated_code = ""
self.test_feedback = ""
self.test_results = []
return self._state()

def _fail(self):
return -5.0

def step(self, action: int):
if self.done:
raise RuntimeError("Episode already finished")
self.steps += 1
reward = -0.1
info = {"action": PlannerAction.NAMES[PlannerAction(action)]}

# PLANNING
if action == PlannerAction.PLANNING:
if self.has_planned:
reward += self._fail()
self.invalid_action = True
else:
self.plan_output = sim_planner.planning(self.task["prompt"])
self.has_planned = True
reward += 1.0

# EXECUTION
elif action == PlannerAction.EXECUTION:
if (not self.has_planned) or self.has_executed:
reward += self._fail()
self.invalid_action = True
else:
self.exec_output = sim_planner.execute(self.plan_output)
self.has_executed = True
reward += 1.5

# INTEGRATION
elif action == PlannerAction.INTEGRATION:
if (not self.has_executed) or self.has_integrated:
reward += self._fail()
self.invalid_action = True
else:
self.integrated_code = sim_planner.integrate(self.exec_output, self.task["prompt"])
self.has_integrated = True
reward += 2.0

# EVALUATION
```

```python
elif action == PlannerAction.EVALUATION:
if (not self.has_integrated) or self.has_evaluated:
reward += self._fail()
self.invalid_action = True
else:
passed, fb = sim_planner.evaluate(self.integrated_code, self.task["tests"])
self.test_results.append(passed)
self.test_feedback = f"Tests {fb['number_passed']}/{fb['total']}"
self.has_evaluated = True
reward += 5.0 if passed else -1.0

# STOP
elif action == PlannerAction.STOP:
self.done = True
if self.has_evaluated and self.test_results and self.test_results[-1] and (not self.invalid_action):
self.success = True
reward += 30.0 if self.success else -10.0

# step limit
if self.steps >= self.max_steps and not self.done:
self.done = True
reward -= 10.0

info["success"] = self.success
info["invalid_action"] = self.invalid_action
return self._state(), reward, self.done, info

def _state(self):
last_code = self.integrated_code if self.integrated_code else ""
last_feedback = self.test_feedback
return make_state_embedding(
self.task["prompt"],
last_code,
last_feedback,
has_planned=self.has_planned,
executed_count=1 if self.has_executed else 0,
integrated=1 if self.has_integrated else 0,
evaluated=1 if self.has_evaluated else 0,
invalid_action=self.invalid_action,
last_test_passed=bool(self.test_results[-1]) if self.test_results else False,
)

# ------------------ Embedding helper ------------------
from sentence_transformers import SentenceTransformer
embedder = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

def make_state_embedding(task_prompt: str, last_code: str, last_test_feedback: str,
has_planned: bool = False, executed_count: int = 0,
integrated: bool = False, evaluated: bool = False,
invalid_action: bool = False, last_test_passed: bool = False) -> np.ndarray:
task_emb = embedder.encode([task_prompt], show_progress_bar=False)
code_emb = embedder.encode([last_code or ""], show_progress_bar=False)
feedback_emb = embedder.encode([last_test_feedback or ""], show_progress_bar=False)
state_vec = np.concatenate([task_emb[0], code_emb[0], feedback_emb[0]])
flags = np.array([
1.0 if has_planned else 0.0,
float(executed_count),
1.0 if integrated else 0.0,
1.0 if evaluated else 0.0,
1.0 if invalid_action else 0.0,
1.0 if last_test_passed else 0.0,
], dtype=np.float32)
```

```
return np.concatenate([state_vec, flags]).astype(np.float32)

sample_state = make_state_embedding(PLANNER_TASK["prompt"], "", "", False, 0, False, False, False, False)
STATE_DIM = sample_state.shape[0]
console.log(f"Planner state dim: {STATE_DIM}")

# ------------------ Dueling DQN + Replay Buffer ------------------
class DuelingQNetwork(nn.Module):
def __init__(self, input_dim, hidden=[512,256], output_dim=len(PlannerAction.NAMES)):
super().__init__()
layers = []
prev = input_dim
for h in hidden:
layers.append(nn.Linear(prev, h))
layers.append(nn.ReLU())
prev = h
self.trunk = nn.Sequential(*layers)
self.value_head = nn.Sequential(
nn.Linear(prev, prev//2 if prev//2>0 else 32),
nn.ReLU(),
nn.Linear(prev//2 if prev//2>0 else 32, 1)
)
self.adv_head = nn.Sequential(
nn.Linear(prev, prev//2 if prev//2>0 else 32),
nn.ReLU(),
nn.Linear(prev//2 if prev//2>0 else 32, output_dim)
)

def forward(self, x):
x = self.trunk(x)
value = self.value_head(x)
adv = self.adv_head(x)
q = value + (adv - adv.mean(dim=1, keepdim=True))
return q

class ReplayBuffer:
def __init__(self, capacity=8000):
self.buffer = deque(maxlen=capacity)
def push(self, s,a,r,ns,done):
self.buffer.append((s,a,r,ns,done))
def sample(self, batch_size):
batch = random.sample(self.buffer, min(batch_size, len(self.buffer)))
s,a,r,ns,d = zip(*batch)
return (np.stack(s), np.array(a), np.array(r, dtype=np.float32), np.stack(ns), np.array(d, dtype=np.float32))
def __len__(self):
return len(self.buffer)

class DQNAgent:
def __init__(self, state_dim, action_dim=len(PlannerAction.NAMES), hidden=[512,256], lr=1e-4, gamma=0.99,
buffer_size=8000, batch_size=64, target_update=500, device=None, mask_actions=True,
eps_decay=0.9995):
self.device = device or ("cuda" if torch.cuda.is_available() else "cpu")
self.qnet = DuelingQNetwork(state_dim, hidden, action_dim).to(self.device)
self.target = DuelingQNetwork(state_dim, hidden, action_dim).to(self.device)
self.target.load_state_dict(self.qnet.state_dict())
self.opt = Adam(self.qnet.parameters(), lr=lr)
self.gamma = gamma
self.buffer = ReplayBuffer(capacity=buffer_size)
self.batch_size = batch_size
self.action_dim = action_dim
```

```
self.eps = 1.0
self.eps_min = 0.05
self.eps_decay = eps_decay
self.learn_steps = 0
self.target_update = target_update
self.mask_actions = mask_actions

def _compute_action_mask(self, state: np.ndarray):
flags = state[-6:]
has_planned = bool(flags[0])
executed_count = int(flags[1])
integrated = bool(flags[2])
evaluated = bool(flags[3])
invalid_action = bool(flags[4])
last_test_passed = bool(flags[5])

mask = np.ones(self.action_dim, dtype=bool)
if has_planned:
mask[PlannerAction.PLANNING] = False
if (not has_planned) or (executed_count >= 1):
mask[PlannerAction.EXECUTION] = False
if (not executed_count) or integrated:
mask[PlannerAction.INTEGRATION] = False
if (not integrated) or evaluated:
mask[PlannerAction.EVALUATION] = False
if not (last_test_passed and (not invalid_action)):
mask[PlannerAction.STOP] = False
return mask

def select_action(self, state):
# EXPLORATION: sample uniformly from full action space (no mask) -> allows discovering failures
if random.random() < self.eps:
return random.randrange(self.action_dim)

# EXPLOIT: use greedy Q with masking to avoid selecting impossible terminal actions
s = torch.FloatTensor(state).unsqueeze(0).to(self.device)
with torch.no_grad():
qvals = self.qnet(s).cpu().numpy().squeeze(0)
if self.mask_actions:
mask = self._compute_action_mask(state)
legal_q = np.where(mask, qvals, -1e9)
if legal_q.max() <= -1e8:
return random.randrange(self.action_dim)
return int(int(legal_q.argmax()))
else:
return int(int(qvals.argmax()))

def push_transition(self, s,a,r,ns,done):
self.buffer.push(s,a,r,ns,done)

def train_step(self):
if len(self.buffer) < 32:
return 0.0
s,a,r,ns,d = self.buffer.sample(self.batch_size)
s = torch.FloatTensor(s).to(self.device)
a = torch.LongTensor(a).to(self.device)
r = torch.FloatTensor(r).to(self.device)
ns = torch.FloatTensor(ns).to(self.device)
d = torch.FloatTensor(d).to(self.device)

qvals = self.qnet(s).gather(1, a.unsqueeze(1)).squeeze(1)
with torch.no_grad():
next_actions = self.qnet(ns).argmax(dim=1, keepdim=True)
```

```
next_q_target = self.target(ns).gather(1, next_actions).squeeze(1)
target = r + self.gamma * next_q_target * (1 - d)

loss = F.mse_loss(qvals, target)
self.opt.zero_grad()
loss.backward()
nn.utils.clip_grad_norm_(self.qnet.parameters(), 0.5)
self.opt.step()

self.learn_steps += 1
if self.learn_steps % self.target_update == 0:
self.target.load_state_dict(self.qnet.state_dict())

self.eps = max(self.eps_min, self.eps * self.eps_decay)
return float(loss.item())

console.log("DQN agent for planner ready.")

# ------------------ Training loop ------------------
def train_planner(agent: DQNAgent, task: Dict, num_episodes=1000, max_steps=10, log_every=50,
seed_expert_episodes=0, save_after=100):
metrics = {"episode_rewards": [], "losses": [], "episode_steps": [], "success_rate_window": [],
    "sample_eff": []}
success_history = deque(maxlen=100)
total_env_steps = 0

ACTION_NAME_TO_INT = {v:k for k,v in PlannerAction.NAMES.items()}
expert_seq = ["planning","execution","integration","evaluation","stop"]

for _ in range(seed_expert_episodes):
env = PlannerEnv(task, max_steps=max_steps)
s = env.reset()
for action_name in expert_seq:
a = ACTION_NAME_TO_INT[action_name]
ns, r, done, info = env.step(a)
agent.push_transition(s, a, r, ns, float(done))
s = ns
if done: break

best_success = 0.0
best_avg_reward = -1e9

for ep in tqdm(range(1, num_episodes+1), desc="Planner DQN Training"):
env = PlannerEnv(task, max_steps=max_steps)
state = env.reset()
ep_reward = 0.0
ep_loss = 0.0

for t in range(max_steps):
action = agent.select_action(state)
next_state, reward, done, info = env.step(action)
agent.push_transition(state, action, reward, next_state, float(done))
loss = agent.train_step()
state = next_state
ep_reward += reward
ep_loss += loss
total_env_steps += 1
if done:
break

metrics["episode_rewards"].append(ep_reward)
metrics["losses"].append(ep_loss / (t+1) if (t+1)>0 else 0.0)
metrics["episode_steps"].append(t+1)
```

```
success_history.append(1.0 if done and info.get("success", False) else 0.0)
metrics["success_rate_window"].append(np.mean(success_history))
metrics["sample_eff"].append(sum(metrics["episode_rewards"]) / (total_env_steps + 1e-8))

current_success = metrics["success_rate_window"][-1]
current_avg_reward = np.mean(metrics["episode_rewards"][-log_every:])

if ep > save_after and (current_success > best_success or (current_success == best_success and current_avg_reward > best_avg_reward)):
best_success = current_success
best_avg_reward = current_avg_reward
torch.save(agent.qnet.state_dict(), "saved_models/planner_dqn_best.pt")
console.log(f"[green]Ep {ep:4d} | New best model saved: success(last100)={best_success:.3%}, avg_reward={best_avg_reward:.3f}[/green]")

if ep % log_every == 0 or ep == 1:
console.log(f"[blue]Ep {ep:4d} | AvgReward {np.mean(metrics['episode_rewards'][-log_every:]):.3f} | "
f"Success(last100) {metrics['success_rate_window'][-1]:.3%} | Eps {agent.eps:.3f}")

return metrics

# ------------------ Run training ------------------
if __name__ == '__main__':
agent = DQNAgent(state_dim=STATE_DIM, action_dim=len(PlannerAction.NAMES), hidden=[512,256], lr=3e-4,
buffer_size=8000, batch_size=64, target_update=200, mask_actions=True, eps_decay=0.9995)

metrics = train_planner(agent, PLANNER_TASK, num_episodes=3000, max_steps=10, log_every=50, seed_expert_episodes=0, save_after=100)

torch.save(agent.qnet.state_dict(), "saved_models/planner_dqn_final.pt")
console.log("Planner training complete. Models saved to saved_models/planner_dqn_*.pt")

for i in range(5):
env = PlannerEnv(PLANNER_TASK, max_steps=10)
s = env.reset()
path = []
for _ in range(12):
a = agent.select_action(s)
path.append(PlannerAction.NAMES[PlannerAction(a)])
s, r, done, info = env.step(a)
if done: break
console.log(f"Demo {i+1}: {path} | success={info.get('success',False)}")
```

Explanation: The planner's role is to produce subtask prompts (planning), call the executive to execute each subtask (execution), combine helpers (integration), and validate the integrated code (evaluation). The training loop follows similar patterns: replay buffer, double-dueling q-networks, target updates, epsilon decay, and occasional saving of best models.

Include an example planner training screenshot and demo results in the report:

```
[23:04:00] Ep 1002 | New best model saved: success(last100)=87.000%,          ipython-input-3307774600.py:442
           avg_reward=35.484
...        Ep 1003 | New best model saved: success(last100)=88.000%,          ipython-input-3307774600.py:442
           avg_reward=35.484
[23:04:04] Ep 1009 | New best model saved: success(last100)=89.000%,          ipython-input-3307774600.py:442
           avg_reward=33.516
[23:04:07] Ep 1012 | New best model saved: success(last100)=89.000%,          ipython-input-3307774600.py:442
           avg_reward=33.820
[23:04:08] Ep 1013 | New best model saved: success(last100)=90.000%,          ipython-input-3307774600.py:442
           avg_reward=35.028
           Ep 1014 | New best model saved: success(last100)=91.000%,          ipython-input-3307774600.py:442
           avg_reward=35.028
[23:04:36] Ep 1050 | AvgReward 26.374 | Success(last100) 85.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:05:12] Ep 1100 | AvgReward 25.924 | Success(last100) 77.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:05:47] Ep 1150 | AvgReward 29.900 | Success(last100) 80.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:06:24] Ep 1200 | AvgReward 29.144 | Success(last100) 84.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:07:01] Ep 1250 | AvgReward 27.252 | Success(last100) 82.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:07:36] Ep 1300 | AvgReward 32.388 | Success(last100) 84.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:08:11] Ep 1350 | AvgReward 30.026 | Success(last100) 86.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:08:47] Ep 1400 | AvgReward 28.546 | Success(last100) 83.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:09:22] Ep 1450 | AvgReward 28.208 | Success(last100) 81.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:09:56] Ep 1500 | AvgReward 31.154 | Success(last100) 82.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:10:31] Ep 1550 | AvgReward 33.334 | Success(last100) 87.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:11:08] Ep 1600 | AvgReward 25.556 | Success(last100) 83.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:11:44] Ep 1650 | AvgReward 29.256 | Success(last100) 79.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:12:21] Ep 1700 | AvgReward 24.054 | Success(last100) 78.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:12:57] Ep 1750 | AvgReward 27.204 | Success(last100) 76.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:13:33] Ep 1800 | AvgReward 30.718 | Success(last100) 82.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:14:08] Ep 1850 | AvgReward 30.116 | Success(last100) 85.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:14:44] Ep 1900 | AvgReward 30.668 | Success(last100) 84.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:15:18] Ep 1950 | AvgReward 32.436 | Success(last100) 86.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:15:55] Ep 2000 | AvgReward 22.190 | Success(last100) 79.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:16:30] Ep 2050 | AvgReward 32.896 | Success(last100) 79.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:17:06] Ep 2100 | AvgReward 33.166 | Success(last100) 89.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:17:41] Ep 2150 | AvgReward 24.040 | Success(last100) 82.000% | Eps 0.050   ipython-input-3307774600.py:445
[23:18:16] Ep 2200 | AvgReward 32.816 | Success(last100) 81.000% | Eps 0.050   ipython-input-3307774600.py:445
```

Figure 3: Planner training success-rate plot / console log.

Reported best planner run excerpt (from your logs):

```
Ep 1014 | New best model saved:  success(last100)=91.000%, avg_reward=35.028
```

```
... [13:17:14] Loaded best model from saved_models/dqn_best.pt                                  ipython-input-1301910945.py:12
               Final model saved to saved_models/dqn_final.pt                                   ipython-input-1301910945.py:26
               Demo rollouts for task1_correct                                                  ipython-input-1301910945.py:30
    [13:17:15]   Path 1: ['generate', 'test', 'stop'] | success=True                            ipython-input-1301910945.py:40
                 Path 2: ['generate', 'test', 'stop'] | success=True                            ipython-input-1301910945.py:40
                 Path 3: ['generate', 'test', 'stop'] | success=True                            ipython-input-1301910945.py:40
               Demo rollouts for task2_incorrect                                                ipython-input-1301910945.py:30
    [13:17:16]   Path 1: ['generate', 'test', 'debug', 'test', 'stop'] | success=True           ipython-input-1301910945.py:40
                 Path 2: ['generate', 'test', 'debug', 'test', 'stop'] | success=True           ipython-input-1301910945.py:40
    [13:17:17]   Path 3: ['generate', 'test', 'debug', 'test', 'stop'] | success=True           ipython-input-1301910945.py:40
               Demo rollouts for task3_search                                                   ipython-input-1301910945.py:30
    [13:17:18]   Path 1: ['generate', 'test', 'debug', 'test', 'search', 'debug',               ipython-input-1301910945.py:40
               'test', 'stop'] | success=True
                 Path 2: ['generate', 'test', 'debug', 'test', 'search', 'debug',               ipython-input-1301910945.py:40
               'test', 'stop'] | success=True
    [13:17:19]   Path 3: ['generate', 'test', 'debug', 'test', 'search', 'debug',               ipython-input-1301910945.py:40
               'test', 'stop'] | success=True
```

Figure 4: Planner demo rollouts showing successful canonical episodes.

9 Hierarchical Inference: Combining Planner + Executive

After training both models, the full hierarchical inference works as follows:

1. Load the planner and executive networks (the trained DQN weights).
2. The planner receives the top-level prompt (for example, `compute (a+b) to the power 2/3`), and then executes its policy, which proceeds through the following sub-steps:
   (a) **Planning:** the planner produces subtask prompts (for example, `add`, `square`, `cbrt`).
   (b) **Execution:** for each subtask the Executive is invoked. The Executive follows the local protocol (for example, `generate`, `test`, optionally `debug`, then `stop`) to produce a helper function and verifies it against unit tests.
   (c) **Integration:** the planner combines the produced helper functions into a single integrated code body (for example, the final `compute()` implementation).
   (d) **Evaluation:** the integrated code is executed and tested (either by the Executive or by a direct test runner). Based on the evaluation outcome the planner decides whether to `stop` or to attempt further corrective actions such as re-planning or re-execution.

3. If the integrated evaluation fails, the planner can in principle learn to re-plan or to trigger additional corrective flows. This extension was not explored in depth in the current experiments.

9.1 Pretty-logged Inference and Final Output

You provided a script that loads networks and pretty-prints the hierarchical inference flow. The script prints only the final `compute` function to be delivered to the user. The provided run demonstrates success: subtasks (add, square, cbrt) were produced, tested, integrated, and the final `compute` passed planner tests.

Final compute function printed by the script:

Listing 10: Final compute() output printed to the user

```

================================================================================
[18:29:08] Hierarchical inference start
================================================================================
[18:29:08] Query: write me a code for computing the third root of (a+b) ^ 2

--------------------------------------------------------------------------------
[18:29:08] Executive: run subtasks (simulated LLM actions)
--------------------------------------------------------------------------------

[18:29:08] Planner I will ask the Executive to solve subtask:
compute_add_001
[18:29:08] Executive generate:
Producing candidate code...

def add(a: int, b: int) -> int:
return a + b
[18:29:08] Executive test:
Running unit tests for the generated code
[18:29:08] Test results: 3/3 passed
[18:29:08] Executive stop:
Episode finished for subtask

[18:29:08] Planner I will ask the Executive to solve subtask:
square_001
[18:29:08] Executive generate:
Producing candidate code...

def square(x: int) -> int:
return x ** 2
[18:29:08] Executive test:
Running unit tests for the generated code
[18:29:08] Test results: 3/3 passed
[18:29:08] Executive stop:
Episode finished for subtask

[18:29:08] Planner I will ask the Executive to solve subtask:
cbrt_001
[18:29:08] Executive generate:
Producing candidate code...

def cbrt(x: int) -> float:
if x == 0:
return 0
sign = -1 if x < 0 else 1
x_abs = abs(x)
r = round(x_abs ** (1.0 / 3.0))
if r ** 3 == x_abs:
```

```
return sign * r
return sign * (x_abs ** (1.0 / 3.0))
[18:29:08] Executive test:
Running unit tests for the generated code
[18:29:08] Test results: 4/4 passed
[18:29:08] Executive stop:
Episode finished for subtask

------------------------------------------------------------------------------
[18:29:08] Integration: combining helper functions into final compute()
------------------------------------------------------------------------------
[18:29:08] Integrated code assembled. Full content below:

def add(a: int, b: int) -> int:
return a + b

def square(x: int) -> int:
return x ** 2

def cbrt(x: int) -> float:
if x == 0:
return 0
sign = -1 if x < 0 else 1
x_abs = abs(x)
r = round(x_abs ** (1.0 / 3.0))
if r ** 3 == x_abs:
return sign * r
return sign * (x_abs ** (1.0 / 3.0))

def compute(a, b):
"""Compute the cube root of (a+b)^2 (i.e. (a+b)^(2/3))."""
s = add(a, b)
s2 = square(s)
return cbrt(s2)


------------------------------------------------------------------------------
[18:29:08] Final Evaluation: validate integrated compute()
------------------------------------------------------------------------------
[18:29:08] TEST: compute(1,2) -> 2.080083823051904 expected 2.080083823051904 ok=True
[18:29:08] TEST: compute(0,0) -> 0 expected 0.0 ok=True
[18:29:08] TEST: compute(8,1) -> 4.3267487109222245 expected 4.3267487109222245 ok=True
[18:29:08] TEST: compute(-1,8) -> 3.6593057100229713 expected 3.6593057100229713 ok=True
[18:29:08] Final tests passed: 4/4

==============================================================================
[18:29:08] FINAL CODE (compute function only)
==============================================================================

def compute(a, b):
"""Compute the cube root of (a+b)^2 (i.e. (a+b)^(2/3))."""
s = add(a, b)
s2 = square(s)
return cbrt(s2)
```

10 Interpretation of Results

- The Executive DQN successfully learned the three canonical episode types and achieved perfect or near-perfect success rates in the reported runs (example: success(last100)=100% at one checkpoint).
- The Planner DQN learned the single planning episode, producing sensible decomposition (3 subtasks) and integrating the helper functions to pass the final tests (reported success(last100)=91% best checkpoint).
- The hierarchical system composed the subtask solutions and produced a correct final function `compute` that passed all planner-level tests in the provided demonstration.

10.1 Why the design works

- **Explicit flags** in the state make temporal preconditions explicit, which is critical for finite-state workflows.
- **Semantic embeddings** for prompt/code/feedback provide content-awareness so the network can generalize across prompts and code.
- **Action masking** prevents the agent from learning trivial but illegal shortcuts and reduces wasted exploration.
- **Dueling + Double DQN** increases stability (separating state value from advantage) and reduces overestimation bias.
- **Seeding with expert episodes** gives early guidance and accelerates learning of the correct sequences.

11 Hyperparameters (selected)

Parameter	Value
Network hidden sizes	[512, 256]
Learning rate	3e-4 (training code uses 3e-4; agent default 1e-4)
Batch size	64
Replay buffer	8000
Target update (steps)	200 (example)
Discount factor (γ)	0.99
Epsilon start / min / decay	1.0 / 0.05 / 0.995 (exec)
Episode max steps	10
Seeding expert episodes	8 per task (exec) / configurable for planner

Table 1: Key hyperparameters used in experiments.