

ДОДАТОК А

Код Double DQN агент для динамічного масштабування складності

Лістинг А-1 – Код Double DQN агент для динамічного масштабування складності

```

import random
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque

DEVICE = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
STATE_DIM = 6 # skill, stress, flow, health, kd, session_len
ACTION_DIM = 8 # кількість можливих дій агента

# ваги функції винагороди
W1, W2, W3, W4 = 0.5, 0.3, 0.1, 0.1
SIGMA = 0.2

def compute_reward(
    skill: float,
    challenge: float,
    stress: float,
    session_delta: float,
    prev_challenge: float,
) -> float:
    # наскільки гравець близько до зони потоку
    r_flow = float(np.exp(-((skill - challenge) ** 2) / (2 *
SIGMA ** 2)))

    # нагороджуємо за те, що гравець не кинув сесію

```

```

r_ret = min(session_delta / 300.0, 1.0)

# бонус коли гравець добре грає і не в стресі
r_perf = 1.0 if (skill > 0.6 and stress < 0.5) else 0.0

# штраф за різкі стрибки складності
r_pen = min(abs(challenge - prev_challenge) / 0.05, 1.0)

return W1 * r_flow + W2 * r_ret + W3 * r_perf - W4 * r_pen

```

```

class QNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        # три шари цілком достатньо - простір стану невеликий
        self.net = nn.Sequential(
            nn.Linear(STATE_DIM, 128), nn.ReLU(),
            nn.Linear(128, 128), nn.ReLU(),
            nn.Linear(128, ACTION_DIM),
        )

    def forward(self, x):
        return self.net(x)

```

```

class ReplayBuffer:
    def __init__(self, capacity: int = 50_000):
        self._buf = deque(maxlen=capacity)

    def push(self, s, a, r, s_next, done):
        self._buf.append((s, a, r, s_next, done))

    def sample(self, batch_size: int):
        batch = random.sample(self._buf, batch_size)
        s, a, r, sn, d = zip(*batch)

```

```

        to_t      =      lambda      x:      torch.tensor(np.array(x),
dtype=torch.float32).to(DEVICE)

        return (
            to_t(s),
            torch.tensor(a, dtype=torch.long).to(DEVICE),
            to_t(r),
            to_t(sn),
            torch.tensor(d, dtype=torch.float32).to(DEVICE),
        )

def __len__(self):
    return len(self._buf)

class DDAAgent:
    GAMMA          = 0.97
    LR              = 1e-3
    BATCH          = 256
    EPS_START      = 1.0
    EPS_END        = 0.05
    EPS_DECAY      = 0.9995
    TARGET_SYNC    = 500    # раз на скільки кроків копіюємо ваги у
target-мережу

    def __init__(self):
        self.q      = QNetwork().to(DEVICE)
        self.q_tgt  = QNetwork().to(DEVICE)
        self.q_tgt.load_state_dict(self.q.state_dict())
        self.q_tgt.eval()

        self.opt          = optim.Adam(self.q.parameters()),
lr=self.LR)

        self.buf      = ReplayBuffer()
        self.eps      = self.EPS_START
        self._steps = 0

```

```

def act(self, state: np.ndarray) -> int:
    if random.random() < self.eps:
        return random.randrange(ACTION_DIM)
    s = torch.tensor(state,
dtype=torch.float32).unsqueeze(0).to(DEVICE)
    with torch.no_grad():
        return int(self.q(s).argmax().item())

def step(self, s, a, r, s_next, done):
    self.buf.push(s, a, r, s_next, done)

    # не починаємо вчитись поки не накопичиться хоч трохи
досвіду
    if len(self.buf) < self.BATCH * 4:
        return

    self._train_once()
    self.eps = max(self.EPS_END, self.eps *
self.EPS_DECAY)
    self._steps += 1

    if self._steps % self.TARGET_SYNC == 0:
        self.q_tgt.load_state_dict(self.q.state_dict())

def _train_once(self):
    s, a, r, sn, done = self.buf.sample(self.BATCH)

    # Double DQN: online-мережа вибирає найкращу дію, target
її оцінює
    # це прибирає систематичне переоцінювання Q-значень
    with torch.no_grad():
        best_a = self.q(sn).argmax(1, keepdim=True)
        q_tgt_v = self.q_tgt(sn).gather(1,
best_a).squeeze(1)
        y = r + self.GAMMA * q_tgt_v * (1 - done)

```

```

q_pred = self.q(s).gather(1, a.unsqueeze(1)).squeeze(1)
loss    = nn.functional.huber_loss(q_pred, y)

self.opt.zero_grad()
loss.backward()

# clip щоб градієнти не вибухали при нестационарному
середовищі
nn.utils.clip_grad_norm_(self.q.parameters(), 1.0)
self.opt.step()

def export_onnx(self, path: str):
    """Зберігає модель у ONNX для завантаження через NNE
    Plugin у UE5."""
    dummy = torch.zeros(1, STATE_DIM).to(DEVICE)
    torch.onnx.export(
        self.q, dummy, path,
        input_names=['state'],
        output_names=['q_values'],
        opset_version=13,
    )
    print(f'збережено: {path}')

```

ДОДАТОК Б

Код модулю профілювання поведінкових показників гравця

Лістинг А-2 – Код модулю профілювання поведінкових показників гравця

```
import numpy as np
from collections import deque
from dataclasses import dataclass
from enum import IntEnum

class PlayerState(IntEnum):
    FLOW = 0
    FRUSTRATION = 1
    BOREDOM = 2

@dataclass
class RawEvent:
    timestamp: float
    hit: bool # влучив чи промахнувся
    evaded: bool # чи встиг ухилитись
    item_used: bool # використав хіл/предмет
    dmg_received: float # скільки шкоди отримав
    died: bool
    aim_angle: float # кут атаки в радіанах

class PlayerProfiler:
    # менше 60 сек - реакція нервова, більше - занадто інертна
    WINDOW_SEC = 60.0

    # більше 3 смертей за хвилину - гравець явно в паніці
    DR_MAX = 3.0

    # емпірично: при std(кути) > 1.8 рад гравець б'є куди попало
```

```

CF_MAX = 1.8
# ширина каналу потоку - підбирали на тестових сесіях
FLOW_EPS = 0.15

def __init__(self):
    self._events: deque[RawEvent] = deque()

    self.skill_score    = 0.5
    self.stress_index   = 0.0
    self.flow_zone      = False
    self.challenge_lvl  = 0.5    # оновлює агент, не ми

    # ці три поля приходять із рушія окремим каналом
    self._health        = 1.0
    self._kd_ratio      = 1.0
    self._session_len   = 0.0

def push(self, ev: RawEvent) -> None:
    self._events.append(ev)

    # прибираємо події за межами вікна
    cutoff = ev.timestamp - self.WINDOW_SEC
    while self._events and self._events[0].timestamp <
cutoff:
        self._events.popleft()

    self._recalc()

def _recalc(self) -> None:
    if not self._events:
        return

    evs = list(self._events)
    n    = len(evs)

    # --- skill_score ---

```

```

        attacks      = [e for e in evs if e.hit or not e.evaded]
        accuracy      = sum(e.hit for e in attacks) /
max(len(attacks), 1)
        evasion       = sum(e.evaded for e in evs) / n
        items_used    = sum(e.item_used for e in evs)
        total_dmg     = sum(e.dmg_received for e in evs)
        resource_eff   = min(items_used / max(total_dmg, 1e-6),
1.0)

        self.skill_score = (
            0.40 * accuracy +
            0.35 * evasion +
            0.25 * resource_eff
        )

        # --- stress_index ---
        span          = evs[-1].timestamp - evs[0].timestamp
        death_rate    = sum(e.died for e in evs) / max(span / 60.0,
1e-6)

        angles        = np.array([e.aim_angle for e in evs])
        chaos          = float(np.std(angles)) if len(angles) > 1
else 0.0

        self.stress_index = min(
            1.0,
            death_rate / self.DR_MAX + chaos / self.CF_MAX
        )

        # --- flow_zone ---
        delta          = abs(self.skill_score -
self.challenge_lvl)
        self.flow_zone = (delta <= self.FLOW_EPS and
self.stress_index <= 0.8)

    def state_vector(self) -> np.ndarray:
        return np.array([

```

```

        self.skill_score,
        self.stress_index,
        float(self.flow_zone),
        self._health,
        self._kd_ratio,
        self._session_len,
    ], dtype=np.float32)

def set_health(self, h: float):    self._health      = h
def set_kd(self, kd: float):      self._kd_ratio    = kd
def set_session(self, t: float):  self._session_len = t

class StateClassifier:
    """
    k-NN класифікатор (k=3, евклідова метрика).
    Три класи: Flow, Frustration, Boredom.
    Вхід: (skill_score, stress_index, flow_zone).
    """
    K = 3

    # невеликий навчальний набір - типові поведінкові профілі
    # кожен рядок: [skill, stress, flow_zone, label]
    _TRAINING_DATA = np.array([
        # Flow - гравець у балансі, виклик відповідає навичкам
        [0.65, 0.30, 1.0, PlayerState.FLOW],
        [0.70, 0.35, 1.0, PlayerState.FLOW],
        [0.55, 0.40, 1.0, PlayerState.FLOW],
        [0.60, 0.25, 1.0, PlayerState.FLOW],
        [0.75, 0.45, 1.0, PlayerState.FLOW],

        # Frustration - забагато помирає, хаотична поведінка
        [0.25, 0.90, 0.0, PlayerState.FRUSTRATION],
        [0.20, 0.85, 0.0, PlayerState.FRUSTRATION],
        [0.30, 0.95, 0.0, PlayerState.FRUSTRATION],
        [0.15, 0.80, 0.0, PlayerState.FRUSTRATION],
    ])

```

```

[0.35, 0.88, 0.0, PlayerState.FRUSTRATION],

# Boredom - вміє більше, ніж вимагає гра, стресу нуль
[0.85, 0.10, 0.0, PlayerState.BOREDOME],
[0.90, 0.08, 0.0, PlayerState.BOREDOME],
[0.80, 0.12, 0.0, PlayerState.BOREDOME],
[0.88, 0.15, 0.0, PlayerState.BOREDOME],
[0.78, 0.05, 0.0, PlayerState.BOREDOME],
])

def classify(self, profiler: PlayerProfiler) -> PlayerState:
    pt = np.array([
        profiler.skill_score,
        profiler.stress_index,
        float(profiler.flow_zone),
    ])

    X = self._TRAINING_DATA[:, :3]
    labels = self._TRAINING_DATA[:, 3].astype(int)

    dists = np.linalg.norm(X - pt, axis=1)

    # беремо k найближчих і голосуємо більшістю
    k_idx = np.argsort(dists)[:self.K]
    k_labels = labels[k_idx]
    counts = np.bincount(k_labels, minlength=3)
    return PlayerState(int(np.argmax(counts)))

```

ДОДАТОК В

Код ігрової підсистеми інференсу та оркестрації DDA

Лістинг А-3 – Код ігрової підсистеми інференсу та оркестрації DDA

```
#include "DDAGameSubsystem.h"
#include "NNE.h"
#include "NNERuntimeCPU.h"
#include "PlayerProfilerComponent.h"
#include "LoDAIController.h"
#include "EngineUtils.h"

DEFINE_LOG_CATEGORY(LogDDA);

void UDDAGameSubsystem::Initialize(FSubsystemCollectionBase&
Collection)
{
    Super::Initialize(Collection);

    // завантажуюємо ONNX модель з Content/AI/
    UNNEModelData* ModelData = LoadObject<UNNEModelData>(
        nullptr, TEXT("/Game/AI/dda_agent.dda_agent")
    );
    if (!ModelData)
    {
        UE_LOG(LogDDA, Warning, TEXT("ONNX не знайдено, DDA
вимкнено"));
        return;
    }

    // отримуємо CPU runtime для інференсу - перевіряємо що він
доступний
    TWeakInterfacePtr<INNERuntimeCPU> Runtime =
UE::NNE::GetRuntime<INNERuntimeCPU>(TEXT("NNERuntimeBasicCpu")
);
```

```

    if (!Runtime.IsValid())
    {
        UE_LOG(LogDDA, Warning, TEXT("NNE Runtime не знайдено,
DDA вимкнено"));
        return;
    }

    // створюємо модель та її екземпляр
    Model = Runtime->CreateModelCPU(ModelData);
    if (!Model.IsValid())
    {
        UE_LOG(LogDDA, Warning, TEXT("Не вдалось створити
модель, DDA вимкнено"));
        return;
    }

    ModelInstance = Model->CreateModelInstanceCPU();
    if (!ModelInstance.IsValid())
    {
        UE_LOG(LogDDA, Warning, TEXT("Не вдалось створити
екземпляр моделі, DDA вимкнено"));
        return;
    }

    // запускаємо інференс раз на секунду - частіше не потрібно
    GetWorld()->GetTimerManager().SetTimer(
        InferenceTimer,                                     this,
        &UDDAGameSubsystem::RunInferenceStep, 1.0f, true
    );

    UE_LOG(LogDDA, Log, TEXT("DDA підсистема ініціалізована
успішно"));
}

void UDDAGameSubsystem::RunInferenceStep()
{

```

```

// якщо модель або профайлер недоступні - пропускаємо крок
if (!ModelInstance.IsValid() || !ProfilerComp) return;

// отримуємо вектор стану гравця [skill, stress, flow,
health, kd, session]
TArray<float> State = ProfilerComp->GetStateVector();

TArray<UE::NNE::FTensorBindingCPU> Inputs, Outputs;
TArray<float> QValues;
QValues.SetNum(8);
Inputs.Add ({ State.GetData(), (uint64)State.Num() *
sizeof(float) });
Outputs.Add({ QValues.GetData(), (uint64)QValues.Num() *
sizeof(float) });

// запускаємо інференс - отримуємо Q-значення для кожної дії
ModelInstance->RunSync(Inputs, Outputs);

// вибираємо дію з максимальним Q-значенням (greedy policy)
int32 Best = 0;
for (int32 i = 1; i < 8; ++i)
    if (QValues[i] > QValues[Best]) Best = i;

// оновлюємо рівень складності - крок не більше 0.05 за раз
const float Delta =
FMath::Clamp(ActionToChallengeDelta(Best), -0.05f, 0.05f);
CurrentChallenge = FMath::Clamp(CurrentChallenge + Delta,
MinDifficulty, MaxDifficulty);

// передаємо новий рівень складності всім AI контролерам на
рівні
for (TActorIterator<AController> It(GetWorld()); It; ++It)
{
    auto* C = Cast<ALoDAIController>(*It);
    if (C) C->SetChallengeLevel(CurrentChallenge);
}

```

```

        // сповіщаємо дашборд про зміну складності
        OnChallengeUpdated.Broadcast(CurrentChallenge, Best,
QValues);
    }

```

```

float UDDAGameSubsystem::ActionToChallengeDelta(int32 Action)
{
    // відображення індексу дії на дельту складності
    switch (Action)
    {
        case 0: return +0.05f; // сильно підвищити
        case 1: return -0.05f; // сильно знизити
        case 2: return -0.02f; // трохи знизити
        case 3: return +0.02f; // трохи підвищити
        case 4: return +0.03f; // помірно підвищити
        case 5: return -0.03f; // помірно знизити
        case 6: return +0.01f; // мікрокорекція вгору
        default: return 0.0f; // нейтраль
    }
}

```

```

void UDDAGameSubsystem::Deinitialize()
{
    // очищаємо таймер перед знищенням підсистеми
    if (GetWorld())
        GetWorld()-
>GetTimerManager().ClearTimer(InferenceTimer);
    ModelInstance.Reset();
    Model.Reset();
    Super::Deinitialize();
}

```

```

void      UDDAGameSubsystem::ReloadConfig(const      FDDAConfig&
NewConfig)
{
    MinDifficulty = NewConfig.DifficultyMin;
    MaxDifficulty = NewConfig.DifficultyMax;
    // затискаємо поточне значення без стрибка
    CurrentChallenge      =      FMath::Clamp(CurrentChallenge,
MinDifficulty, MaxDifficulty);

    UE_LOG(LogDDA, Log, TEXT("ReloadConfig: min=%.2f max=%.2f
speed=%.2f strategy=%d"),
        MinDifficulty,                        MaxDifficulty,
NewConfig.AdaptationSpeed, NewConfig.NPCStrategyIndex);
}

```

ДОДАТОК Г

Відомість кваліфікаційної роботи

Позначення					Найменування		Дод. відомості	
					Текстові документи			
1.					Пояснювальна записка		63 с.	
					</			