NAWIGACJA ROBOTÓW HUMANOIDALNYCH

WYKŁAD 6

World-Model-Action (WMA) w nawigacji humanoida



dr inż. Mateusz Pomianek

Model świata | Planowanie | DreamerV3/TDMPC | ROS 2 + WMA

Plan wykładu

Sekcja 1

Intuicja i architektura WMA: model świata, stan latentny, planowanie w wyobraźni, RSSM

Sekcja 2

Algorytmy i implementacje: DreamerV3, TDMPC2, JEPA; jak działają krok po kroku

Sekcja 3

WMA w nawigacji humanoida: integracja z ROS 2, planista wielopoziomowy, trening i wdrożenie

Kody Python

RSSM, planowanie w wyobraźni, trening DreamerV3, integracja z Nav2

01

SEKCJA

Czym jest WMA i model świata



Intuicja, architektura, stan latentny i planowanie w wyobraźni

Intuicja: jak człowiek planuje ruch

1. Zanim wstaniesz z krzesła i przejdziesz do kuchni, wyobrażasz sobie tę trasę w głowie
2. Twój mózg posiada wewnętrzny model mieszkania i przewiduje, co zobaczysz za rogiem
3. Nie testujesz każdego kroku fizycznie; symulacja w głowie jest tańsza i bezpieczniejsza
4. Jeśli wyobrażona ścieżka prowadzi do kolizji, wybierasz inną ZANIM zaczniesz iść

To właśnie robi World Model: uczy się kompaktowej reprezentacji (stanu latentnego) i symuluje przyszłość

WMA to robot z wbudowaną wyobraźnią: planuje w przestrzeni latentnej, nie na fizycznym robocie

Korzyść: można przetestować tysiące sekwencji akcji w milisekundach bez ryzyka uszkodzenia sprzętu

Trzy podejścia do sterowania robotem

Reaktywne (Reflex)

- obs → action natychmiast
- Szybkie, proste (PID, DWA)
- Brak planowania do przodu
- Nie uczy się środowiska
- Wrażliwe na zmiany otoczenia
- Przykład: regulator P dla yaw

Model-free RL

- Polityka uczona metodą prób
- Miliony rollout-ów na robocie
- Brak jawnego modelu środowiska
- Dobra wydajność końcowa
- Drogie: czas + zużycie sprzętu
- Przykład: PPO, SAC, TD3

WMA (Model-based)

- Model świata: $(s_t, a_t) \rightarrow s_{t+1}$
- Planowanie w wyobraźni (latent)
- 10-100× mniej interakcji z robotem
- Generalizacja z modelu na nowe sceny
- Wymaga dobrego modelu przejścia
- Przykład: DreamerV3, TDMPC2

Definicja i składowe architektury WMA

- **WMA (World-Model-Action)** = system sterowania zbudowany wokół wyuczonego modelu dynamiki środowiska
- **Komponent 1 - Enkoder (encoder):** obserwacja o_t → stan latentny z_t (kompresja do niskwymiarowej reprezentacji)
- **Komponent 2 - Model przejścia (transition model):** (z_t, a_t) → z_{t+1} (przewiduje następny stan w przestrzeni latentnej)
- **Komponent 3 - Model nagrody/wartości (reward/value head):** z_t → r_t lub V(z_t) (ocena jakości stanu)
- **Komponent 4 - Dekoder (decoder, opcjonalny):** z_t → ô_t (rekonstrukcja obserwacji; służy do uczenia enkodera)
- **Komponent 5 - Planista/polityka (policy):** planuje sekwencję akcji maksymalizującą wartość w przestrzeni latentnej
- **Trening**: minimalizacja błędu rekonstrukcji + błędu predykcji przejścia + maximalizacja nagrody (ELBO / RL)

Architektura WMA - od obserwacji do akcji

Obserwacja o_t	Obraz RGB, chmura LiDAR, propriocepcja (q, dq, τ) - surowe dane z sensorów robota
Enkoder E(·)	CNN lub ViT kompresuje obserwację do stanu latentnego z_t ∈ ℝ^d (d = 512-2048)
Recurrent State h_t	GRU/LSTM przechowuje historię; h_t = f(h_{t-1}, z_t); modeluje zależności czasowe
Model przejścia T(·)	MLP/Transformer: (h_t, a_t) → ẑ_{t+1}; predykcja w wyobraźni bez rzeczywistego kroku
Planista + polityka	MPPI / CEM / actor-critic planuje N kroków do przodu w latent space; wybiera a_t*

Kluczowe cechy

- Planowanie bez GPU-heavy forward pass
- Latent dim 512: 1000 trajektorii w ~5 ms
- Model przejścia: ~2M parametrów
- Enkoder: ViT-S ~22M lub CNN ~5M
- Trening: 100k-10M klatek z robota
- Inferencja: ~1 ms na GPU embedded

Stan latentny - czym jest i dlaczego działa

Czym jest stan latentny z_t?

- **Kompresja obserwacji** do wektora o wymiarze d (np. 512)
- **Enkoder** uczy się wybrać z 1920×1080 RGB tylko istotne cechy
- **z_t zawiera**: pozycję obiektów, kształt przejść, kierunek ruchu
- **Analogia**: notatka "drzwi otwarte, 2 m do przodu" zamiast całego obrazu
- **Stochastyczny**: z_t ~ q(z|o_t) - rozkład Gaussa; próbkowanie
- **Deterministyczny**: h_t = GRU(h_{t-1}, z_t) - pamięć sekwencji

Dlaczego planowanie w latent space działa?

- **Model przejścia T**: (z_t, a_t) → z_{t+1} jest gładki i ciągły
- **Gradient propaguje się** przez wirtualne kroki → uczenie polityki
- 1000 trajektorii w latent space = 1000 wyobrażeń w ~2 ms
- **Modele przejścia generalizują**: jeśli korytarz podobny → działa
- **Błędy kumulują** się po ~10-20 krokach; stąd krótki horyzont planowania
- **Re-planowanie co krok**: plan 1 kroku, wykonaj, planuj od nowa

RSSM: Recurrent State Space Model (serce WMA)

- **RSSM** to podstawowy blok DreamerV3; łączy deterministyczny i stochastyczny model stanu
- **Deterministyczna część**: $h_t = f_\theta\left(h_{\{t-1\}}, z_{\{t-1\}}, a_{\{t-1\}}\right) - GRU$; zapewnia ciągłość historii
- **Stochastyczna część**: $z_t \sim q_\varphi(z|h_t, o_t)$ - enkoder posterior; $z_t \sim p_\theta(z|h_t)$ - prior
- **Predykcja bez obserwacji (wyobraźnia):** zaczynając od (h_t, z_t) propaguj TYLKO prior bez nowych o_t
- **Funkcja straty**: $L = L_{recon} + \beta \cdot KL(posterior \,||\, prior) + L_{reward} + L_{continue}$
- **KL-balancing**: gradient przez prior i posterior osobno skalowany (β = 0,1→0,5 dla prior)
- **Wynik**: model umie jednocześnie OPISYWAĆ teraźniejszość i WYOBRAŻAĆ sobie przyszłość

Planowanie w wyobraźni - krok po kroku

1

Kodowanie bieżącej obserwacji

Enkoder E pobiera klatkę o_t i tworzy z_t; GRU aktualizuje h_t; razem = bieżący stan świata

2

Rozwinięcie wyobraźni

Polityka π(a|h,z) proponuje akcje; model T(h,z,a)→(h',z') propaguje stan TYLKO w latent space przez H kroków (H=15)

3

Ocena trajektorii

Model wartości V(h,z) i nagrody r(h,z) oceniają każdy wirtualny krok; λ-return sumuje zdyskontowane nagrody

4

Aktualizacja polityki

Gradient z V i r propaguje się wstecz przez wirtualne kroki aż do π; actor-critic update bez fizycznego robota

KOD PYTHON

RSSM: implementacja rdzenia modelu świata (PyTorch)

```python
import torch, torch.nn as nn

class RSSM(nn.Module):
    def __init__(self, obs_dim=512, act_dim=7,
                 det_dim=512, stoch_dim=32, stoch_classes=32):
        super().__init__()
        # Deterministyczna czesc (GRU)
        self.gru = nn.GRUCell(stoch_dim*stoch_classes + act_dim, det_dim)
        # Prior p(z | h)  - bez obserwacji
        self.prior = nn.Linear(det_dim, stoch_dim * stoch_classes)
        # Posterior q(z | h, obs)  - z obserwacja
        self.posterior = nn.Linear(det_dim + obs_dim, stoch_dim * stoch_classes)
        self.stoch_dim, self.stoch_classes = stoch_dim, stoch_classes

    def imagine_step(self, h, z, action):
        """Krok wyobrazni: (h,z,a) -> (h_next,z_next) BEZ obserwacji."""
        z_flat = z.view(z.shape[0], -1)  # (B, stoch*classes)
        h_next = self.gru(torch.cat([z_flat, action], -1), h)
        logits = self.prior(h_next)       # prior p(z|h)
        z_next = self._gumbel(logits)     # Gumbel-softmax
        return h_next, z_next
    def _gumbel(self, l):
        l = l.view(-1, self.stoch_dim, self.stoch_classes)
        return nn.functional.gumbel_softmax(l, tau=1.0, hard=True)
```

Opis: imagine_step() propaguje stan w latent space; nie potrzebuje obserwacji; planowanie w wyobraźni

02

SEKCJA

Algorytmy WMA - DreamerV3, TDMPC2, JEPA



Jak działają krok po kroku; różnice i zastosowania w lokomocji

DreamerV3 - architektura i cykl uczenia

Zbieranie danych	Polityka π wykonuje akcje na robocie; (o, a, r, done) zapisywane do replay buffer
Uczenie modelu świata	Mini-batch z replay → RSSM; minimalizacja L_recon + KL + L_reward + L_continue
Wyobraźnia (Imagine)	Z replay buffer inicjuj stany; rozwiń H=15 kroków wyłącznie modelem T; bez robota
Uczenie aktora-krytyka	Actor: max λ-return przez gradienty z wyobraźni; Critic: regresja na λ-return
Wykonanie na robocie	Polityka π(a|h,z) w latent space; akcja dekodowana i wysyłana do ros2_control / WBC

Kluczowe cechy

- Symultaniczny trening: model + policy
- Jeden GPU (RTX 3090): ~10h training
- Działa z obrazem RGB lub proprio
- DreamerV3: zunifikowane hiperparametry dla wszystkich środowisk
- Atari, DMControl, HumanoidWalk: SoTA
- Open-source: github.com/danijar/dreamerv3

TDMPC2: Temporal Difference Model Predictive Control

- **TDMPC2** łączy model świata z optymalizacją trajektorii w stylu MPC (Model Predictive Control)
- **Latent world model**: enkoder E + model przejścia T + głowica nagrody R + value function V (wszystkie MLP)
- **Planowanie przez MPPI (Model Predictive Path Integral):** próbkuj N trajektorii w latent space, weź najlepszą
- **Temporal consistency loss**: predykcja H kroków do przodu; gradienty przez cały horyzont
- **Kluczowa zaleta**: bardzo małe modele (~0,5M-5M parametrów) i szybka inferencja (~0,5 ms/krok)
- **Benchmark**: SoTA na Humanoid-9DoF (DMControl); chód, bieg, backflip w ciągłej przestrzeni akcji
- **Open-source**: github.com/nicklashansen/tdmpc2; gotowy interfejs dla robotów z PyTorch

JEPA: Joint Embedding Predictive Architecture

Idea JEPA (LeCun, Meta AI)

- **Predykcja** w przestrzeni reprezentacji, nie pikseli
- **Context encoder**: koduje widoczny fragment sceny
- **Target encoder**: koduje ukryty/przyszły fragment
- **Predictor**: próbuje przewidzieć target z context
- **Brak dekodera**: nie rekonstruuje surowych pikseli
- **Zaleta**: uczy się abstrakcyjnych cech, a nie tekstur
- **Szybszy trening**: bez kosztownej rekonstrukcji RGB

JEPA w nawigacji robota

- **I-JEPA (obraz):** pre-training enkodera ViT bez etykiet
- **V-JEPA (wideo):** predykcja przyszłych klatek wideo
- **Używane jako pre-trained backbone** dla RSSM lub TDMPC2
- Robot widzi lewą połowę korytarza → przewiduje prawą
- **Semantyczne reprezentacje** lepsze niż px-rekonstrukcja
- **Integracja**: JEPA encoder → latent z_t → RSSM/TDMPC2

Porównanie algorytmów WMA dla humanoida

DreamerV3

RSSM + actor-critic; imagen. H=15; SoTA na humanoid locomotion; pełny kod open-source

TDMPC2

MLP world model + MPPI; 0,5M param; szybki; idealny na Jetson; SoTA DMControl Humanoid

IRIS

World model jako autoregressive transformer (jak GPT); tokeny stanu; silny w rzadkich nagrodach

DayDreamer (robot)

Adaptacja Dreamer do real robotów; trening na żywym robocie bez resetów; prawie zero-shot transfer

HarmonyDream

WMA dla humanoid navigation: hierarchia Dreamer (wysoki) + TDMPC2 (niski); wyniki ICRA 2024

SWIM (MuJoCo)

Scalable World-model for Imagination; 3D voxel world model; planowanie 3D dla humanoida na schodach

Uczenie WMA - dane, trening i hiperparametry

- Dane wejściowe: sekwencje (o_1, a_1, r_1, o_2, a_2, …) z eksploracyjnej polityki lub teleoperation
- Rozmiar replay buffer: typowo 1M-10M kroków; priorytetowe próbkowanie (PER) poprawia próbkowanie rzadkich zdarzeń
- Batch size: 16 sekwencji × 64 kroki = 1024 kroków na batch; częstość uczenia: 1 krok uczenia / 1 krok środowiska
- Hiperparametry DreamerV3 zunifikowane: lr=1e-4, γ=0,997, λ=0,95, β=1,0; działają bez strojenia między środowiskami
- Curriculum: zacznij od krótkich trajektorii i prostych środowisk; stopniowo zwiększaj horyzont i złożoność
- Augmentacja danych: random crop, color jitter, losowe wycinki czasowe - podwaja efektywność danych z robota
- Metryki treningu: błąd rekonstrukcji, KL divergence, episode return podczas walidacji w symulacji

KOD PYTHON

Planowanie w wyobraźni: horyzont H kroków

```python
import torch

def imagine_rollout(rssm, actor, h0, z0, horizon=15, n_traj=512):
    """Rozija N trajektorii w przestrzeni latentnej przez H kroków."""
    B = n_traj
    # Inicjalizacja: powiel stan początkowy dla wszystkich trajektorii
    h = h0.unsqueeze(0).expand(B, -1)   # (B, det_dim)
    z = z0.unsqueeze(0).expand(B, -1, -1)  # (B, stoch_dim, classes)
    states_h, states_z, actions = [h], [z], []
    for _ in range(horizon):
        # Polityka wybiera akcje z bieżącego stanu latentnego
        z_flat = z.view(B, -1)
        a_dist = actor(torch.cat([h, z_flat], dim=-1))
        a      = a_dist.rsample()       # różniczkowane próbkowanie
        # Krok wyobrazni (bez robota!)
        h, z   = rssm.imagine_step(h, z, a)
        states_h.append(h); states_z.append(z); actions.append(a)
    # Zwróć stosy tensorów: (H+1, B, dim)
    return torch.stack(states_h), torch.stack(states_z), torch.stack(actions)
```

Opis: n_traj=512 trajektorii × H=15 kroków: cały rollout zajmuje ~2 ms na GPU; gradients flow przez wszystkie kroki

KOD PYTHON

Szkielet treningu DreamerV3 (pętla treningowa)

```python
# Zakładamy: rssm, decoder, reward_head, value_fn, actor, critic
# replay_buffer przechowuje sekwencje (o,a,r,done)

for step in range(total_steps):
    # --- 1. Krok na robocie ---
    obs = robot.get_observation()           # RGB + proprio
    z, h = rssm.encode_and_update(obs)      # posterior + GRU
    action = actor(h, z).sample().clamp(-1, 1)
    reward, next_obs, done = robot.step(action)
    replay_buffer.add(obs, action, reward, done)

    # --- 2. Uczenie modelu swiata ---
    batch = replay_buffer.sample(batch_size=16, seq_len=64)
    loss_wm = world_model_loss(rssm, decoder, reward_head, batch)
    loss_wm.backward(); opt_wm.step(); opt_wm.zero_grad()

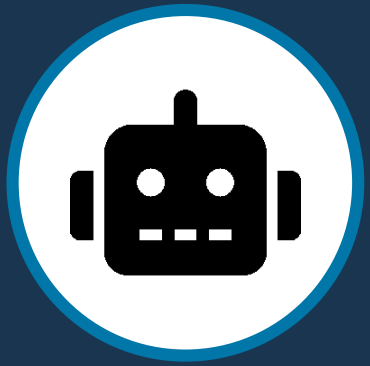
    # --- 3. Uczenie aktora-krytyka w wyobraźni ---
    h0, z0 = rssm.get_current_state()
    hs, zs, acts = imagine_rollout(rssm, actor, h0, z0)
    returns   = compute_lambda_returns(reward_head, value_fn, hs, zs)
    loss_ac   = actor_critic_loss(actor, critic, hs, zs, acts, returns)
    loss_ac.backward(); opt_ac.step(); opt_ac.zero_grad()
```

Opis: Trzy fazy w każdym kroku: rollout na robocie → ucz model świata → ucz actor-critic w wyobraźni

03

SEKCJA

WMA w nawigacji humanoida



Integracja z ROS 2, hierarchia planowania, trening i wdrożenie

Hierarchiczna architektura nawigacji z WMA

Poziom misji (LLM/VLM)	Polecenie językowe → sekwencja celów nawigacyjnych; wywołanie co ~2 s
Poziom globalny (A* + SLAM)	Mapa zajętości, ścieżka metryczna do celu; aktualizacja co 0,5-2 s; klasyczny planista
Poziom lokalny: WMA (TDMPC2)	World model przewiduje H=10 kroków; MPPI wybiera optymalną (v, ω); 20-50 Hz
Poziom chodu: WBC + RSSM	WMA dla lokomocji: planuje sekwencję kroków w latent space; 200 Hz WBC + 10 Hz WMA
Aktuatory (ros2_control)	Komendy momentów τ [Nm]; pętle PID 1 kHz; enkodery i F/T → sprzężenie zwrotne do WMA

Kluczowe cechy

- Każdy poziom ma własną częstość i horyzont
- Błędy WMA nie blokują WBC (fallback)
- Wspólny stan latentny h między poziomami
- Można zastąpić pojedynczy poziom
- Testowany: DreamerV3 + Nav2 w Gazebo
- **Czas latencji**: WMA ≈ 5 ms / krok

Etapy trenowania WMA dla nawigacji humanoida

1

Pre-training w symulacji

MuJoCo / Isaac Sim; 10M kroków; polityka eksploracyjna; uczenie RSSM i enkodera na danych proprio + RGB

2

Fine-tuning w symulacji

Zadanie nawigacyjne: dotarcie do celu ±0,1 m; funkcja nagrody: -d_goal + bonus_sukces - kolizja; DreamerV3 / TDMPC2

3

Sim-to-real transfer

Domain randomization: masa, tarcie, opóźnienia, tekstury; tylko enkoder re-fine-tunowany na 1h danych z realnego robota

4

Online adaptation

Na robocie: replay buffer zasilany danymi online; model przejścia update co 100 kroków; polityka adaptuje się w locie

Projektowanie funkcji nagrody dla nawigacji WMA

- **Nagroda za zbliżenie do celu**: $r_{goal} = -\frac{d(pos_t, goal)}{d_{initial}}$; znormalizowana, gęsta
- **Kara za kolizję**: $r_{coll} = -10$ gdy dist(przeszkoda) < 0,3 m; epizod kończy się
- **Nagroda za stabilność ZMP**: $r_{zmp} = +0{,}2$ gdy ZMP w wieloboku; -0,5 gdy poza marginesem 2 cm
- **Kara za zużycie energii**: $r_{energy} = -0{,}001 \cdot \sum|\tau_i \cdot dq_i|$; preferuje płynny chód
- **Nagroda za prędkość**: $r_{vel} = +0{,}1 \cdot v_{forward}$; zachęca do sprawnego ruchu ku celowi
- **Nagroda za sukces**: $r_{success} = +50\ gdy\ |pos - goal| < 0{,}1\ m$; jednorazowy bonus
- **Balans**: $r_{total} = r_{goal} + r_{zmp} + r_{energy} + r_{vel} + r_{coll} + r_{success}$; wagi dobierane eksperymentalnie

Integracja WMA z ROS 2 - wzorzec implementacji

- **Node /wma_policy**: subskrybuje /camera/image_raw + /odom + /joint_states; publikuje /cmd_vel lub /footstep_goal
- **Cykl działania**: encode(obs) → rssm.imagine(H=10) → mppi_select_action() → publish
- **Latencja**: TDMPC2 ~3 ms, DreamerV3 ~8 ms na Jetson AGX Orin; mieści się w cyklu 50 Hz
- **Interfejs z Nav2**: /wma_policy jako custom local planner plugin (nav2_core::Controller interface)
- **TF2**: node czyta transformacje map→odom→base; stan latentny enkoduje RELATYWNĄ pozycję celu
- **Watchdog**: jeśli WMA nie odpowie przez >50 ms → Nav2 fallback do DWB controller
- **Diagnostics**: /wma_diagnostics publikuje latencję, wartość funkcji wartości V(h,z), błąd predykcji

KOD PYTHON

WMA jako kontroler lokalny w Nav2 (Python szkielet)

```python
import rclpy; from rclpy.node import Node
import torch, numpy as np
from geometry_msgs.msg import Twist

class WMAController(Node):
    def __init__(self):
        super().__init__('wma_controller')
        self.rssm  = torch.load('rssm_humanoid.pt').eval().cuda()
        self.actor = torch.load('actor_nav.pt').eval().cuda()
        self.h = torch.zeros(1, 512).cuda()     # stan deterministyczny GRU
        self.z = torch.zeros(1, 32, 32).cuda()  # stan stochastyczny
        self.cmd_pub  = self.create_publisher(Twist, '/cmd_vel', 1)
        self.obs_sub  = self.create_subscription(
            CompressedImage, '/camera/compressed', self.obs_cb, 1)
    def obs_cb(self, msg):
        obs = preprocess_image(msg)             # (1, 3, 64, 64) tensor
        with torch.no_grad():
            self.h, self.z = self.rssm.observe(obs, self.h, self.z)
            action = self.actor(self.h, self.z.view(1,-1)).mean
        cmd = Twist()
        cmd.linear.x  = float(action[0].clamp(-0.5, 0.5))
        cmd.angular.z = float(action[1].clamp(-1.0, 1.0))
        self.cmd_pub.publish(cmd)
```

Opis: observe() aktualizuje (h,z) z nowej klatki; actor wybiera akcję z latent state; cmd_vel do Nav2 costmap

WMA vs. VLM vs. klasyczny planista -> kiedy stosować?

Klasyczny planer (A* + DWA)

- **Deterministyczny** i formalnie poprawny
- **Wymaga gotowej mapy** środowiska
- **Cel = współrzędne** (x, y) w mapie
- **Brak uczenia**: brak adaptacji online
- **Szybki**: A* ~5 ms, DWA ~2 ms
- **Najlepszy**: znane, statyczne środowisko

VLM (CLIP, LLaVA)

- **Rozumie polecenia** językowe i semantykę
- **Zero-shot**: nie wymaga mapy etykiet
- **Wolny**: 100-800 ms / zapytanie
- **Nie planuje fizyki** - tylko co i gdzie
- **Najlepszy**: grounding, instruction follow
- **Łączy się z A*** jako oracle celu

WMA (DreamerV3, TDMPC2)

- **Uczy się fizyki środowiska** z doświadczeń
- **Szybkie planowanie**: ~3-8 ms / krok
- Generalizuje: nowe ułożenia mebli OK
- **Wymaga treningu** (~100k kroków sim.)
- **Najlepszy**: dynamiczna lokomocja, przeszkody
- **Łączy się z VLM** (cel) + A* (trasa globalna)

Ograniczenia i otwarte wyzwania WMA

Problemy techniczne

- **Błąd kompounding**: model przejścia po 20+ krokach wyobraźni kumuluje błędy predykcji
- **Trudność z rzadkimi zdarzeniami**: upadek, kolizja; replay buffer rzadko je zawiera
- **Kosztowne uczenie**: 10M kroków simul. × 5 ms/krok ≈ 14h; realny robot = dni
- **Sim-to-real gap**: model przejścia nie uwzględnia elastyczności struktury, tarcia statycznego
- **Rozkład wielomodalny**: kilka równie dobrych ścieżek; RSSM z rozk. Gaussa słabo je modeluje

Kierunki badań i rozwiązania

- **Dłuższy efektywny horyzont**: hierarchiczne WMA (krótki + długi horyzont)
- **Aktywne zbieranie danych**: polityka eksploracyjna celowo szuka rzadkich zdarzeń (curiousity)
- **Real-world fine-tuning**: 1h danych z robota + pre-trained sim model → sim-to-real w ~30 min
- **Normalizing flows / diffusion**: lepsze modelowanie wielomodalności niż Gauss
- **Foundation World Models**: duży model pre-trained na danych z wielu robotów (Open-X-Embodiment)

Use case: WMA dla humanoida patrolującego budynek

- **Platforma**: Unitree H1 / Agility Digit; symulacja: MuJoCo Menagerie z MJCF modelem robota
- **Architektura**: LLM task planner → A* global path → TDMPC2 local navigation → WBC footstep
- **Trening TDMPC2**: 5M kroków symulacji (≈7 h na RTX 4090); nagroda: r_goal + r_zmp + r_energy
- **Dane realne**: 45 min teleoperation w budynku → fine-tuning enkodera (tylko E); reszta zamrożona
- **Wyniki walidacji**: success rate 92% (vs. 71% DWA) na trasach z dynamicznymi przeszkodami
- **Zużycie energii**: -18% względem DWA; WMA wybiera płynniejsze trajektorie z mniejszymi momentami
- **Czas odpowiedzi**: TDMPC2 ~4 ms/krok na Jetson Orin; A* co 1 s; LLM co 5 s (asynchronicznie)

PODSUMOWANIE

01 **Model świata** uczy się przewidywać kolejne stany środowiska z obserwacji i akcji; planowanie odbywa się w przestrzeni latentnej.

02 **WMA = enkoder + model przejścia + model nagrody/wartości + planista + polityka**; każdy komponent jest uczony przez RL lub imitation learning.

03 **DreamerV3 i TDMPC2** to referencyjne implementacje gotowe do adaptacji na dane robotyczne z lokomocją humanoidalną.

04 **Planowanie w wyobraźni** eliminuje kosztowne rollout-y na robocie i przyspiesza uczenie o 10-100×.

05 **WMA, VLM i klasyczny planer A*** są komplementarne i mogą działać w jednej hierarchicznej architekturze nawigacji.

Zadanie projektowe i pytania problemowe

1. Wyjaśnij własnymi słowami różnicę między posterior $q(z|h,o)$, a prior $p(z|h)$ w RSSM. Dlaczego w fazie "wyobraźni" używamy prioru, a nie posterioru?
2. Dlaczego planowanie w przestrzeni latentnej jest szybsze niż planowanie z surowymi obserwacjami? Podaj konkretne liczby (wymiarowość, liczba operacji).
3. Jakie hiperparametry DreamerV3 należałoby dostroić, gdybyś trenował robot do nawigacji w ciemnym korytarzu z kamerą termowizyjną zamiast RGB?

Zadanie projektowe: zaimplementuj uproszczony RSSM (enkoder 2-warstwy MLP, GRU 256, prior 2-warstwy MLP) i wytrenuj go na danych z symulowanego wahadła odwróconego (CartPole). Sprawdź, czy model potrafi wyobrazić sobie 10-krokową trajektorię startując z losowego stanu.

Bonus: połącz TDMPC2 z Nav2 w Gazebo dla robota kołowego (TurtleBot3) i porównaj success rate z DWB na mapie z 5 dynamicznymi przeszkodami.