NAWIGACJA ROBOTOW HUMANOIDALNYCH

# WYKŁAD 8

## Uczenie przez wzmacnianie

w nawigacji i lokomocji humanoidow

dr inż. Mateusz Pomianek



Podstawy RL | PPO / SAC / TD3 | Sim-to-real | Lokomocja + Nav

# Plan wykladu

## Sekcja 1

Podstawy RL: MDP, polityka, nagroda, algorytmy model-free (PPO, SAC, TD3) - od intuicji do wzoru

## Sekcja 2

RL w lokomocji humanoidow: trenowanie chodu, biegania, wchodzenia na schody w MuJoCo / Isaac Sim

## Sekcja 3

RL w nawigacji spolecznej i integracja z ROS 2: reward shaping, curriculum, sim-to-real, wdrożenie

## Kody Python

Petla RL (Gym), PPO sketch, reward humanoid, curriculum, node ROS 2 z polityka RL

01

SEKCJA

# Podstawy uczenia przez wzmacnianie



*MDP, polityka, nagroda, Value function, algorytmy model-free*

# **Intuicja:** uczenie przez doswiadczenie

- **Dziecko uczy się chodzić przez tysiące prób i upadków:** nikt nie pisze mu równania kinematyki
- **Każda próba dostarcza sygnału**: upadłeś (kara) lub zrobiłeś krok (nagroda) - mozg aktualizuje strategie
- **Po wystarczająco wielu próbach** dziecko chodzi płynnie, bez świadomej analizy każdego mięśnia
- **Reinforcement Learning (RL) replikuje ten mechanizm**: agent (robot) + środowisko + nagroda r_t
- **Klucz**: robot NIE potrzebuje modelu dynamiki ani ręcznie zakodowanych reguł - uczy sie sam z interakcji
- **Przewaga nad klasycznym sterowaniem**: generalizacja do nieprzewidzianych warunków (nierówny teren, pchnięcia)
- **Wyzwanie**: potrzeba milionów kroków interakcji; w robotyce fizycznej = dni; stąd symulacja jako pośrednik

# Markowski Proces Decyzyjny (MDP): formalizacja RL

- **MDP definiują**: S - przestrzeń stanów, A - przestrzeń akcji, $P(s'|s,a)$ - model przejścia, $r(s,a)$ - nagroda, gamma - współczynnik dyskontowania
- **Stan s_t**: dla humanoida = $[q, dq, orientacja_IMU, kontakt_stop, cel_xyz]$ - typowo 40-80 wymiarów
- **Akcja a_t**: komendy momentów stawowych [Nm] lub docelowe katy stawów [rad] - 12-30 wymiarów
- **Nagroda r_t**: funkcja zaprojektowana przez inżyniera; klucz do sukcesu całego treningu
- **Cel agenta**: maksymalizacja sumy zdyskontowanych nagród $G_t = sum_\{k=0\}\^\{inf\} gamma\^k * r_\{t+k\}$
- **Polityka pi(a|s):** funkcja mapująca stan na rozkład akcji; parametryzowana siecią neuronowa theta
- **Value function** $V(s) = E[G_t \mid s_t = s]$: ocena jak dobry jest dany stan przy optymalnej polityce

# Porównanie algorytmow RL model-free

## PPO (Proximal Policy Opt.)

- **On-policy**: uczy sie z biezacej polityki
- **Clip loss**: kontroluje wielkosc aktualizacji
- **Stabilny**, latwy w implementacji
- **Popularny w lokomocji** (IsaacGym)
- **Wymaga duzo danych** (on-policy)
- **Ref**: Schulman et al., 2017

## SAC (Soft Actor-Critic)

- **Off-policy**: replay buffer, probkowanie
- **Maksymalizuje nagrode** + entropia
- **Probkowanie efektywniejsze niz PPO**
- **Nadaje sie do zadan ciaglych**
- **Dwa krytycy Q1, Q2** - stabilnosc
- **Ref**: Haarnoja et al., 2018

## TD3 (Twin Delayed DDPG)

- **Off-policy**, deterministyczna polityka
- **Dwa Q-networks:** redukuje overestimation
- **Opozniona aktualizacja aktora**
- **Szybki w zadaniach manipulacji**
- **Dobry punkt startowy** dla robotykow
- **Ref**: Fujimoto et al., 2018

# Architektura aktor-krytyk (Actor-Critic)

| | |
|---|---|
| **Stan s_t** | **Propriocepcja + cel**: [q, dq, IMU, contact, goal_dir] - wejście do obu sieci |
| **Aktor pi(a\|s)** | **MLP 3×[512 ELU]:** wyjście mu i log_sigma dla rozkladu Gaussa nad akcja a_t |
| **Próbkowanie akcji** | **a_t ~ N(mu, sigma^2); reparametrization trick**: a = mu + sigma*eps, eps~N(0,1) |
| **Środowisko / robot** | **Wykonaj a_t**; odbierz r_t, s_{t+1}; dane do rollout buffer (PPO) lub replay (SAC) |
| **Krytyk V(s) / Q(s,a)** | **MLP 3×[512 ELU]:** szacuje wartosc stanu lub pary (stan, akcja); blad TD -> gradient |

## Kluczowe cechy

- **Aktor i krytyk** współdzielą warstwy w PPO
- **Typowy MLP**: 3 warstwy po 256-512 neuronow
- **Aktywacja**: ELU lub Tanh (ostatnia warstwa)
- **Batch norm** lub **layer norm**: stabilność
- lr aktor 3e-4; lr krytyk 1e-3
- **Gradient clipping**: max_norm = 1.0
- **Horyzont rollout PPO**: T = 24 krokow

KOD PYTHON

# Petla RL: MuJoCo Humanoid z Gymnasium

```python
import gymnasium as gym
import numpy as np

env     = gym.make("Humanoid-v5", render_mode=None)
obs, _  = env.reset(seed=42)  # wektor stanu (376,)
ep_rewards = []

for episode in range(5):
    obs, _ = env.reset()
    ep_ret = 0.0
    done   = False
    while not done:
        # Polityka - tu: losowa (zastap siecia PPO/SAC)
        action = env.action_space.sample()  # (17,) momenty
        obs, reward, terminated, truncated, info = env.step(action)
        ep_ret += reward
        done = terminated or truncated
    ep_rewards.append(ep_ret)
    print(f"Epizod {episode}: return = {ep_ret:.1f}")
env.close()
print(f"Średni zwrot z 5 epizodów: {np.mean(ep_rewards):.1f}")
```

**Opis:** Gymnasium Humanoid-v5: 376-wymiarowy stan, 17 momentow stawowych; pelna dynamika MuJoCo 3.x.

02

SEKCJA

# RL w lokomocji humanoidów



*Trenowanie chodu, biegania, schodow; funkcja nagrody; curriculum learning*

# Srodowiska symulacyjne dla lokomocji RL

## MuJoCo / Gymnasium

Klasyczny benchmark; Humanoid-v5, HumanoidStandup; elastyczny; Python-native; open-source od 2022

## IsaacGym / IsaacLab

GPU-parallel: 4096+ robotów jednocześnie na jednym GPU; PPO w 10 min; NVIDIA; open-source

## IsaacSim (Omniverse)

Fotorealistyczna symulacja; ROS 2 native; RTX rendering; domyślne środowisko dla Franka, H1

## PyBullet

Lekki, open-source; wolniejszy od Isaac; dobry do prototypowania;
`pip install pybullet`

## Genesis (2024)

Python-native; 10-100× szybszy od MuJoCo na CPU; diff physics; nowy standard akademicki

## Webots

Open-source; wbudowany ROS 2 driver; dobry dla studentów; słabsza dokładność kontaktu

# Projektowanie funkcji nagrody dla chodu humanoida

- **r_velocity**: $+v_x * \cos(yaw_err)$ - nagroda za ruch do przodu we właściwym kierunku; skaluje prędkość liniowo
- **r_alive**: $+0.5$ w każdym kroku w którym robot nie upadł; krytyczne dla uczenia wstawania
- **r_upright**: $+1.0 * (1 - |pitch| / pi)$ - nagroda za pionowa postawę tułowia (brak pochylenia)
- **r_feet_contact**: $+0.1$ gdy przynajmniej jedna stopa dotyka podłoża; kara za brak kontaktu obu stop
- **r_energy**: $-0.001 * sum(|tau_i * dq_i|)$ - kara za zbyt duże zużycie mocy aktuatorów
- **r_action_smooth**: $-0.01 * ||a_t - a_\{t - 1\}||\text{^}2$ - kara za gwałtowne zmiany komend (drgania)
- $r_total = r_velocity + r_alive + r_upright + r_feet + r_energy + r_smooth$; suma aż do upadku (done=True)

# Curriculum learning - stopniowe trudnosci treningu

1

## Faza 1: Wstawanie

**Cel**: robot uczy się stać. Nagroda tylko za r_upright + r_alive. Brak zadania prędkości. ~10M kroków.

2

## Faza 2: Chód wolny

Wprowadź r_velocity z docelowym v_ref = 0.5 m/s. Plaska, gładka nawierzchnia. ~50M kroków.

3

## Faza 3: Prędkość

Stopniowo zwiększ v_ref do 1.5 m/s. **Domain rand**: masa +/-10%, tarcie zmienne. ~100M kroków.

4

## Faza 4: Teren

Nierówności (heightfield), schody, rampy. Korekta funkcji nagrody **o margines ZMP**. ~200M kroków.

KOD PYTHON

# Funkcja nagrody dla chodu humanoida (PyTorch / IsaacLab)

```python
import torch

def compute_reward(obs: torch.Tensor,        # (N, obs_dim)
                   action: torch.Tensor,     # (N, act_dim)
                   prev_action: torch.Tensor,
                   alive: torch.Tensor,      # (N,) bool
                   cfg) -> torch.Tensor:
    vx       = obs[:, 0]                                  # prędkość do przodu [m/s]
    pitch    = obs[:, 6]                                  # pochylenie tułowia [rad]
    tau      = obs[:, 20:37]                              # momenty stawów [Nm]
    dq       = obs[:, 37:54]                              # prędkości stawów [rad/s]
    r_vel    = vx * cfg.w_vel                              # nagroda prędkości
    r_alive  = alive.float() * cfg.w_alive                 # przeżycie
    r_uprt   = (1.0 - pitch.abs() / 3.14) * cfg.w_upr      # pionowość
    r_enrg   = -(tau * dq).abs().sum(-1) * cfg.w_ener      # energia
    r_smth   = -(action - prev_action).pow(2).sum(-1) * cfg.w_sm
    return r_vel + r_alive + r_uprt + r_enrg + r_smth
```

**Opis:** cfg.w_vel=1.0, w_alive=0.5, w_upr=0.3, w_ener=0.001, w_sm=0.005 - startowe wagi do strojenia

# Domain randomization i transfer sim-to-real

## Domain Randomization (DR)

- **Losowanie parametrów modelu podczas treningu**: masa ciała +/-15%, momenty bezwładności +/-10%
- **Tarcie stopy**: u w [0.4, 1.2]; kluczowe dla różnych nawierzchni
- **Opóźnienie aktuatora**: 0-40 ms; symuluje opóźnienie sieci i kontrolera
- **Szum obserwacji**: N(0, sigma) do q, dq, IMU - odporność na błędy enkodera
- **Zewnętrzne zakłócenia**: losowe impulsy siły na tułów
- **Cel**: pokryć różnice miedzy symulacja a fizycznym robotem

## Transfer sim-to-real (best practices)

- **Fine-tuning na robocie**: zamroź aktora, douczaj tylko ostatnia warstwę (1h danych = 30 min treningu)
- **History buffer**: podaj ostatnie k=3 obserwacje i akcje; kompensuje opóźnienia
- **Asymetryczny AC**: krytyk ma dostęp do pełnego stanu w sim; aktor uczy się tylko z obserwacji
- **System identification**: dopasuj tarcie i masę przed treningu przez krótkie sesje na robocie
- **Metryki:** v_forward, margines ZMP, zużycie energii porównywalne sim vs. real

# Wyniki RL w lokomocji humanoidow (2022-2025)

## DeepMind - Harness

**PPO + MuJoCo**; humanoid uczy się wstawać, chodzić, biegać z jednej politykami. Nature 2019.

## ETH - ANYmal RL

**PPO + IsaacGym**; quadruped RL do 3 m/s; transfer sim2real w 15 min kalibracji. RSJ 2022.

## UC Berkeley - Cassie

**PPO + MuJoCo**; bieganie 3 m/s, kucanie, skakanie; odporność na pchnięcia. RSS 2022.

## Agility Digit - IROS24

**SAC + curriculum**; chód po terenie, schody, ramp; transfer na fizycznego Digit ~4h danych.

## Unitree H1 - 2024

**PPO + IsaacLab**; bieg 3,3 m/s z backflip; 2048 środowisk równoległych na jednym GPU.

## Figure 01 - 2024

**RL + VLA (GPT-4o);** humanoid może nauczyć się pieczenia jajek, sprzątania; end to end z LLM

03

SEKCJA

# RL w nawigacji i integracja z ROS 2



*Nawigacja spoleczna z RL, reward shaping, wdrozenie na robocie*

# RL dla nawigacji społecznej: reward shaping

- **Cel**: robot uczy się omijać pieszych bez ręcznie zaprojektowanego planisty lokalnego
- **r_goal**: -d(pos_t, goal) / d_initial - nagroda za zbliżanie sie do celu nawigacyjnego
- **r_social**: -sum_j [ max(0, d_safe - ||pos - ped_j||) ] - kara proporcjonalna do naruszenia strefy osobistej
- **r_collision**: -10 gdy ||pos - ped|| < 0.3 m lub kolizja ze ścianą; epizod kończy się
- **r_comfort**: -||a_t - a_{t-1}||^2 * 0.01 - kara za gwałtowne manewry (komfort pieszego)
- **r_progress**: +0.2 gdy robot przeszedł do nowego punktu kontrolnego; hamuje tendencje do okrążania celu
- **Środowisko**: CrowdNav (Chen et al.) lub SEAN-EP; 5-15 pieszych z Social Force Model lub nagraniami

# Hierarchiczne RL dla nawigacji humanoidow

## Wysoki poziom - nawigacja

- **Policy nawigacyjna**:
  $obs = [cel_{dir}, dist_{goal}, pedestrians_{relative}]$
  $-> a = [v, omega]$
- **Trening**: CrowdNav / IsaacSim; 8 pieszych, randomizowane cele
- **Częstotliwość**: 10 Hz; wyjście: wektor (v, omega) do planisty kroków
- **Transfer**: fine-tuning na danych z korytarza (30 min sesji)
- **Algorytm**: SAC; replay buffer 1M; batch 256; lr 3e-4

## Niski poziom - lokomocja

- **Policy lokomocyjna**:
  $obs = [v_{cmd}, omega_{cmd}, q, dq, IMU, contact]$
  $-> a = tau$
- **Trening: IsaacGym**; 2048 środowisk; 200M kroków; PPO
- **Częstotliwość**: 200 Hz (WBC loop integration)
- **Odbiera komendy od planisty** nawigacyjnego co 50 ms
- **Algorytm**: PPO; clip = 0.2; T = 24; mini-batches = 4

KOD PYTHON

# Uproszczona polityka nawigacyjna RL (SAC)

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

class NavPolicy(nn.Module):
    def __init__(self, obs_dim=18, act_dim=2, hidden=256):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.ELU(),
            nn.Linear(hidden, hidden),  nn.ELU(),
            nn.Linear(hidden, act_dim)  # mu [v, omega]
        )
    def forward(self, obs: torch.Tensor) -> torch.Tensor:
        mu = self.net(obs)
        # Ogranicz v in [-0.5, 1.0] m/s; omega in [-1.0, 1.0] rad/s
        v     = torch.tanh(mu[:, 0]) * 0.75 + 0.25  # preferencja do przodu
        omega = torch.tanh(mu[:, 1]) * 1.0
        return torch.stack([v, omega], dim=-1)
```

**Opis:** obs_dim=18: [goal_dx, goal_dy, dist_goal, 5x{ped_dx,dy,vx,vy,dist}]; output: (v, omega) do lokomocji.

# Integracja polityki RL z ROS 2

- **Node /rl_nav_policy**: subskrybuje **/odom**, **/goal_pose**, **/pedestrians**; publikuje **/rl_cmd_vel** (10 Hz)
- **Node /rl_loco_policy**: subskrybuje **/joint_states**, **/imu**, **/rl_cmd_vel**; publikuje **/joint_torques** (200 Hz)
- **Wczytanie wag**: `torch.load("nav_policy.pt").eval(); torch.jit.trace()` do 2× szybszej inferencji
- **Latencja**: `NavPolicy ~0.8 ms`; `LocoPolicy ~1.2 ms` na Jetson AGX Orin - miesci sie w cyklu sterowania
- **Safety wrapper**: przed kazdą akcja sprawdź clip prędkości i ZMP; watchdog zatrzymuje przy przekroczeniu
- **Fallback**: gdy RL policy nie odpowie przez >30 ms, aktywuj klasyczny DWA kontroler z Nav2
- **Logowanie**: zapisuj `(obs, action, reward)` do rosbag2 - dane do dalszego fine-tuningu online

KOD PYTHON

# ROS 2 node - polityka nawigacyjna RL (Python)

```python
import rclpy, torch
from rclpy.node import Node
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry

class RLNavNode(Node):
    def __init__(self):
        super().__init__("rl_nav")
        self.policy = torch.jit.load("nav_policy_traced.pt").eval()
        self.goal   = None          # PoseStamped goal
        self.peds   = []            # lista pieszych z detektora
        self.cmd_pub = self.create_publisher(Twist, "/rl_cmd_vel", 1)
        self.create_subscription(Odometry, "/odom", self.odom_cb, 1)
        self.create_timer(0.1, self.timer_cb)   # 10 Hz
    def timer_cb(self):
        if self.goal is None: return
        obs = self._build_obs()     # (1, 18) tensor
        with torch.no_grad():
            action = self.policy(obs).squeeze().numpy()
        cmd = Twist()
        cmd.linear.x  = float(action[0].clip(-0.5, 1.0))
        cmd.angular.z = float(action[1].clip(-1.0, 1.0))
        self.cmd_pub.publish(cmd)
```

**Opis:** torch.jit.load() laduje skompilowany TorchScript - 2x szybszy i niezalezny od kodu treningu.

# Ograniczenia i otwarte pytania RL w humanoidach

## Problemy techniczne i badawcze

- **Kosztowny trening**: 100M kroków PPO to ~2 h na 4×A100; realny robot = tygodnie bez DR
- **Reward hacking**: agent nagminnie oszukuje nagrodę nieoczekiwanymi ruchami (np.toczy się zamiast chodzić)
- **Długookresowe planowanie**: RL słabo radzi sobie z horyzontami >50 kroków bez hierarchii lub modelu świata
- **Generalizacja topologii**: polityka nauczona w jednym budynku może nie działać w nowym
- **Interpretowalność**: siec neuronowa nie podaje powodu decyzji; trudność certyfikacji bezpieczeństwa

## Kierunki (2024-2026)

- **RL + WMA**: DreamerV3 jako efektywny solver MDP z wyobraznia (patrz Wyklad 6)
- **RLHF (od człowieka):** ludzkie pary preferencji jako sygnał nagrody zamiast ręcznie projektowanej r
- **Foundation policies**: jeden model pre-trained na 100 robotach (Open-X-Embodiment + PPO)
- **Safe RL**: constrained MDP z gwarancja P(kolizja) < delta; certifiable przez conformal prediction
- **Multi-agent RL**: kilka humanoidów uczących się współpracować lub omijać się nawzajem

# RL vs klasyczne planowanie vs WMA

## Klasyczny planista (A*/DWA)

- **Deterministyczny**, formalnie weryfikowalny
- **Wymaga gotowej mapy** i modelu dynamiki
- **Szybki** w czasie wykonania (ms)
- **Nie uczy się** z doświadczenia
- **Słabe przy dużej nieliniowości** i zakłóceniach
- **Najlepszy**: znane środowisko, cert. bezpieczeństwo

## Model-based RL/WMA

- **Planowanie w wyobraźni**: 10-100× mniej danych
- **Generalizacja** z modelu przejścia
- **Może być interpretowany** przez model świata
- **Wymaga treningu** RSSM / TDMPC2
- **Średnio skuteczny** w dl. horyzoncie
- **Najlepszy**: ograniczone dane, bezpieczne uczenie

## Model-free RL (PPO/SAC)

- **Nie wymaga modelu dynamiki**
- **Skaluje się**: 2048 środowisk w Isaac
- **Najlepsza wydajność** w dl. treningu
- **Wymaga dużo danych**; może hakować nagrodę
- **Trudny do certyfikacji**
- **Najlepszy**: duża zdolność obliczeniowa, sim2real

# PODSUMOWANIE WYKLADU 8

01 **MDP i algorytmy PPO/SAC/TD3 to fundamenty RL**; siec aktor-krytyk uczy się polityki przez interakcje z symulacja.

02 **Funkcja nagrody i curriculum są kluczowe**: źle zaprojektowana nagroda prowadzi do reward hackingu zamiast chodzenia.

03 **Domain randomization + asymetryczny AC** to sprawdzone recepty na udany transfer sim-to-real w mniej niż 2h kalibracji.

04 **Hierarchiczne RL rozdziela nawigacje** (10 Hz) **od lokomocji** (200 Hz); oba poziomy integrują się z ROS 2 przez torchscript.

05 **RL, WMA i klasyczny planista są komplementarne**; optymalna architektura łączy je w hierarchii zależnie od horyzontu i ryzyka.

# Zadanie projektowe i pytania problemowe

1. Dlaczego polityka RL musi mieć dostęp do historii obserwacji (history buffer) przy transferze sim-to-real? Jaką długość bufora jest wystarczająca i dlaczego?
2. Zaprojektuj funkcje nagrody dla humanoida uczącego się wchodzić po schodach o nieznanej wysokości stopni (10-20 cm). Jakie składniki nagrody są niezbędne, a jakie mogą prowadzić do reward hackingu?
3. Wyjaśnij różnice miedzy on-policy (PPO) a off-policy (SAC) w kontekście ograniczonego czasu treningu na GPU. Kiedy wybrać jeden, a kiedy drugi?

**Zadanie**: zaimplementuj pętle RL z Gymnasium Humanoid-v5 (Kod 1) i zastąp losowa politykę najprostszym możliwym aktorem (MLP 2 warstwy, tanh, opt Adam). Porównaj średni return po 10k i 100k krokach.

**Bonus**: zintegruj TorchScript policy (Kod 4) z Nav2 w Gazebo i porównaj czas dotarcia do celu z 5 pieszymi względem klasycznego DWB kontrolera.