ChatGPT

Guia Completo para Treinar um Robô a Saltar via Aprendizado por Reforço (com Integração ROS2/ Gazebo)

Visão Geral e Objetivo do Treinamento

Este guia descreve um processo completo de treinamento por Aprendizado por Reforço (RL) para ensinar um robô legged (ex.: quadrúpede ou humanoide) a **realizar um salto sob comando**, acionado pela tecla **Espaço** do teclado. Além do pulo, o sistema será preparado para **outras habilidades** de locomoção, como andar para frente/trás, virar (esquerda/direita) – controlados pelas teclas **W/S/A/D** – e até agachar (útil como etapa do salto ou como ação independente). O objetivo é que, ao carregar o modelo treinado em um simulador físico (MuJoCo ou Gazebo), possamos controlar o robô **como em um jogo**, pressionando Espaço para pular e usando WSAD para deslocamento. Por fim, abordaremos a integração deste modelo em um ambiente **ROS2/Gazebo**, visando uso futuro no robô simulado (ou real) via ROS2.

Em resumo, teremos um *controlador aprendido* (policy de RL) capaz de mapear comandos simples do usuário (ex.: "pular agora", "andar para frente", "virar à esquerda") em ações coordenadas do robô. Essa abordagem substitui a necessidade de programar manualmente controladores de equilíbrio, locomoção ou sequências de salto – em vez disso, o robô aprende esses comportamentos através de tentativas no simulador e de uma função de recompensa bem definida. A seguir, detalhamos cada etapa: configuração do ambiente de treinamento, definição de observações/ações, design da função de recompensa para salto, execução do treinamento (incluindo possíveis *curriculums* ou estratégias multi-tarefa), e finalmente a implementação do controle por teclado no Gazebo com ROS2.

Configuração do Ambiente de Treino (Simulador e Modelo do Robô)

Antes de treinar, precisamos configurar um ambiente de simulação física realista para o robô. Duas opções populares citadas são **NVIDIA Isaac Gym** e **DeepMind MuJoCo**, frequentemente usados no framework *unitree_rl_gym*. O Isaac Gym permite simulações massivamente paralelas na GPU, acelerando o treinamento em 2–3 ordens de magnitude comparado à CPU, enquanto o MuJoCo oferece física de alta fidelidade, embora mais lenta [1] [2]. Uma estratégia comum é **treinar no Isaac Gym pela velocidade e depois testar no MuJoCo pela precisão** [2] (Sim2Sim), pois há diferenças entre simuladores que podem degradar um pouco o desempenho ao transferir a política [2]. Alternativamente, pode-se treinar diretamente no MuJoCo (simplicidade com fidelidade maior, porém tempo de treino maior) ou até utilizar o Gazebo acoplado a ROS2 para treinar (ex.: via *Gym-Gazebo2* [3]), embora esse último seja mais lento que simuladores especializados.

Modelo do Robô: Certifique-se de ter o modelo do robô no simulador escolhido. Por exemplo, no *unitree_rl_gym* existem configurações prontas para robôs da Unitree (p.ex. quadrúpede Go2, humanoide H1/ G1) incluindo dinâmicas e configurações de juntas [4]. Se você usar Gazebo, importe o URDF do robô e

configure plugins/controladores (ex.: usando `gazebo_ros2_control` para controlar as juntas). Em MuJoCo, carregue o modelo (XML MJCF) equivalente. Garanta que o modelo de treinamento e o do Gazebo tenham consistência em massa, centro de massa e limites das juntas.

Parâmetros Físicos e Escala de Tempo: Use um passo de simulação pequeno o suficiente para estabilidade (p.ex. 0.002s–0.005s). Para saltos dinâmicos, uma taxa alta de controle (por volta de 100 Hz) é recomendada para capturar rápidas mudanças de força e postura [5]. Evite gravidade exagerada ou damping excessivo – mantenha condições físicas realistas. Vale também habilitar aleatoriedade de domínio (*domain randomization*) moderada (fricção, massas, pequenos ruídos) durante o treino, para melhorar robustez e transferibilidade ao Gazebo/real.

Definição das Observações e Ações do Agente

Estado (Observações): O vetor de observação do agente deve incluir informações suficientes sobre o estado do robô e *comandos do usuário*. Tipicamente, em tarefas de locomoção, incluem-se atributos proprioceptivos como ângulos articulares, velocidades articulares, orientação do corpo (IMU), posição/velocidade do corpo, estados de contato dos pés, etc [6]. Além disso, é fundamental incluir no estado um **vetor de comando desejado** que representará a intenção do usuário: por exemplo, componentes de velocidade linear e angular (v_x, v_y, yaw) para indicar direção de marcha [7]. No nosso caso, podemos estender esse conceito para incluir um **sinal de pulo**. Algumas abordagens possíveis:

- Usar uma variável binária `jump_command` na observação, que será `1` quando o usuário apertar Espaço (solicitando um salto) e `0` caso contrário. A política deve então aprender a **mudar de comportamento** ao detectar esse comando.
- Alternativamente, definir um "vetor de comando" mais geral, onde diferentes dimensões codificam modos: por ex., `cmd = [v_x, yaw_rate, jump_flag]`. Assim, com `jump_flag=0`, os valores `v_x`/yaw definem marcha; com `jump_flag=1`, a política ignora `v_x` e foca em executar o salto.
- Caso opte por políticas separadas para cada habilidade (discutido adiante), então para a política de salto a observação pode ser composta apenas do estado do robô (ângulos, velocidades, orientação, etc.) sem comando de velocidade, mas possivelmente com um indicador se deve pular agora ou esperar. No treinamento do salto em si, podemos simplificar assumindo que o comando de pulo está ativo desde o início do episódio (ou em um momento aleatório pré-definido).

Ações: As ações geradas pela política de RL serão os comandos de controle para o robô. Há duas abordagens comuns: - **Comandos de Posição/Velocidade (alto nível):** Fazer a rede gerar **posições angulares desejadas** para cada articulação, que serão seguidas por controladores de baixo nível (PD) nos motores [7]. Essa é a abordagem usada no *unitree_rl_gym*: a política não gera torques brutos diretamente, mas sim ângulos alvo (*setpoints*) para cada junta, e um controlador de posição simulado converte isso em torque [8]. Isso tende a estabilizar o aprendizado e imitar os controladores internos do robô real [8]. Por exemplo, se o robô tem 12 DoFs nas pernas, a ação da rede pode ser um vetor de 12 ângulos alvo a cada passo de controle. - **Comandos de Torque (baixo nível):** Alternativamente, a política poderia emitir diretamente torques em cada articulação. Isso dá mais liberdade ao agente, porém é mais difícil de aprender e menos estável. No caso de saltos, torques diretos podem levar a comportamentos instáveis (o robô chutando o chão descontroladamente). Portanto, recomenda-se usar ações como posições/ângulos alvo com um controlador PD para aplicar torques – replicando a configuração de treino no Isaac Gym ou controlador do robô físico [8]. Se optar por torque direto, considere limitar amplitudes e derivar penalidades para ações bruscas.

Normalização e Escalas: Normalizar as observações (e eventualmente as ações) ajuda na aprendizagem. Ex.: escalar ângulos para [-1,1] dividindo pelo limite, velocidades divididas pela velocidade máxima esperada, etc. O comando de velocidade pode ser normalizado em relação a um máximo (ex.: v_x=1.0 representaria a velocidade máxima desejada para frente que se quer treinar). No caso do comando de pulo binário, pode simplesmente ser 0 ou 1 (ou 0/1 normalizado também). As ações de ângulo podem ser limitadas dentro dos limites físicos da junta.

Projeto da Política de RL e Algoritmo de Treinamento

Com ambiente e espaços definidos, escolha um algoritmo de RL apropriado. Uma escolha consagrada para robótica legged é **Proximal Policy Optimization (PPO)** devido à sua estabilidade e eficiência em ambientes contínuos. De fato, o framework Unitree utiliza a biblioteca **RSL-RL** (ETH Zurich) que implementa PPO totalmente na GPU [9] – ideal para treinar milhares de ambientes paralelos no Isaac Gym. PPO com múltiplos ambientes paralelos e observações em batelada é capaz de aprender locomoção complexa (até mesmo humanoides) em tempo viável, dada uma função de recompensa bem configurada.

Arquitetura da Rede: Use uma política estocástica gaussiana contínua para as ações (no caso de ações contínuas). A rede *actor* pode ser um MLP (perceptron multi-camadas) que recebe o vetor de observação e produz a média de uma distribuição Gaussiana para cada ação (e talvez um desvio padrão aprendível comum ou por ação). Uma arquitetura típica: 2 a 3 camadas densas de 128-256 neurônios com ativação ReLU/Tanh. A *critic* (rede de valor) pode compartilhar algumas camadas ou ser separada, mapeando o estado para um valor escalar V(s). Dado que queremos reagir rapidamente aos comandos, manter a arquitetura enxuta ajuda na inferência a 100Hz. (Inferências de uma MLP pequena levam <1ms [10] [11], então o gargalo será mais a comunicação e interface com simulador do que o cálculo em si.)

Configurações do Treino: Configure hiperparâmetros de PPO adequados: - *Horizonte de episódio:* defina quando um episódio termina. Para o salto, pode ser no momento do pouso + alguns steps de estabilização, ou um tempo fixo (por ex. 3 segundos) – o que ocorrer primeiro. Sempre termine o episódio se o robô **cair** (queda/colisão do tronco no chão), aplicando uma grande punição para desencorajar quedas. - *Número de ambientes paralelos:* quanto mais melhor (se usar Isaac Gym, ex. 1024 ou 4096 ambientes [12]). Em MuJoCo puro, paralelizar é mais limitado, talvez rodar ~16 instâncias em paralelo via threads/processos. - *Taxa de aprendizado, clipping, etc:* inicie com valores comuns (lr ~ 3e-4, clipping epsilon 0.2, batch_size = n_envs *n_steps per update, etc). RSL-RL/LeggedGym já fornece bons defaults para locomoção.* - Exploração e entropia: *Use um coeficiente de entropia pequeno mas não zero para incentivar explorar (p.ex. 0.01). Salto é um comportamento difícil, então pode ser necessário maior exploração inicialmente.* - Curriculum:* Considere estratégias de curriculum (ver detalhes abaixo) ajustando gradualmente dificuldade ou objetivo da recompensa conforme o agente melhora.

Execução do Treinamento: Inicie o treino com o script apropriado (por ex., `train.py` do legged_gym), especificando a tarefa e parâmetros. Exemplo indicado para o G1 humanoide: `--num_envs=4096 --max_iterations=10000 --task=g1 --headless True` [13] [12]. Isso rodaria 4096 ambientes paralelos sem renderização, por 10000 iterações (o que pode equivaler a dezenas de milhões de passos simulados). Logs de recompensa média e checkpoints da política (ex.: `model_10000.pt`) serão salvos periodicamente [14]. Não se assuste se no início o robô **cair frequentemente**; com o progresso, a recompensa média aumenta e os episódios duram mais sem quedas [15]. Treinar um humanoide para andar ou pular pode levar **várias horas ou dias** dependendo da GPU e complexidade [16]. Interrompa e retome com checkpoint se necessário (`--resume`).

Função de Recompensa para o Salto (Design e Shaping)

O design da **função de recompensa** é o coração do sucesso no treinamento por reforço. Para o comportamento de salto, a recompensa deve incentivar o robô a realizar um salto efetivo (suficientemente alto/longe) *sem cair* e com estabilidade. Diferente de andar, o pulo é uma das ações mais desafiadoras de aprender, pois envolve rápida mudança de postura, fase aérea e aterrissagem controlada [17]. Portanto, é essencial definir recompensas que guiem cada aspecto do salto [18]. Podemos decompor o salto em **fases** e atribuir termos de recompensa específicos a cada fase (*approach/preparation, takeoff, flight, landing*) [18]. Abaixo propomos componentes de recompensa:

- **Altura (ou Distância) do Salto:** Incentive o robô a alcançar uma altura desejada. Por exemplo, um termo de recompensa pode ser a altura máxima do centro de massa ou dos pés. Se o objetivo for pular para frente, pode-se recompensar a distância horizontal percorrida. Esta recompensa principal direciona o comportamento a *sair do chão*. Poderia ser algo como `r_altura = k1 * altura_com` durante a fase aérea (até um teto de altura alvo). Se há uma altura/destino específico, pode usar a diferença para a altura alvo [19] (ex.: recompensa decrescente com erro do alvo).
- **Sincronização/Agachamento (Fase de Preparação):** Para um salto eficiente, o robô deve primeiro **agachar** (flexionar as pernas) antes de estender. Podemos adicionar recompensa na fase de preparação se o robô atingir uma postura agachada adequada. Por exemplo, medir o ângulo do joelho ou altura do quadril: quanto mais baixo (até certo limite), maior a recompensa, incentivando a acumular energia elástica. Esse termo atua como *checkpoint* inicial do salto, garantindo cooperação entre membros anteriores e posteriores para um bom impulso [20]. No artigo de Zhou *et al.* (2024), foi projetada uma recompensa para **coordenar membros anteriores e posteriores** de modo a alinhar o vetor de impulso corretamente na decolagem [20] [21] – ou seja, assegurar que ao esticar as pernas, a força resulte em projeção para cima em vez de desequilíbrio.
- **Estabilidade da Postura (Durante Voo e Pouso):** Penalize rotações excessivas e instabilidade. Um salto bem-sucedido deve manter o robô sem tombar: pode-se incluir um termo que penaliza ângulos de rolamento/pitch grandes (|roll|, |pitch|) durante o voo e especialmente no momento do pouso. Também penalize se o robô tocar o chão com partes indevidas (ex.: joelhos, tronco). Zhou *et al.* implementaram um termo de recompensa para manter a estabilidade da postura e evitar giros no ar [22] – isso ajuda o robô a **não capotar** e aterrissar em pé. Poderíamos ter `r_estabilidade = k2 * exp(- |roll| - |pitch|)`, por exemplo, ou simplesmente deduzir uma punição se |roll/pitch| exceder certos graus.
- **Aterrissagem Suave:** Incentivar que, na aterrissagem, o impacto seja absorvido corretamente. Podemos punir choques muito bruscos (p. ex., picos de força de impacto ou mudanças rápidas de velocidade do corpo). Se os sensores simulados permitirem, pode monitorar força nos pés: penalize forças acima de um limiar (incentivando prepará-los). Uma aterrissagem bem sucedida poderia ser premiada com um bônus final.
- **Penalidade por Queda:** Se o robô cair (tronco atinge o chão ou perde equilíbrio irreversivelmente), termine o episódio e aplique uma penalidade significativa (ex.: -100) para deixar claro que cair é o pior resultado. Isso direciona a política a preferir **não saltar de qualquer jeito** se for levar a uma queda.
- **Custo Energético (Opcional):** Para evitar comportamentos explosivos desnecessários, inclua um leve custo por esforço (por exemplo, soma do quadrado dos torques ou potência instantânea) para encorajar saltar eficientemente. No entanto, cuidado para não punir tanto a energia a ponto de

desencorajar o salto alto – ajuste o peso desse termo de forma que o robô ainda precise saltar para ganhar a grande recompensa de altura.

- **Seguir Comando:** Se no nosso treinamento considerarmos cenários onde às vezes o comando de pulo não está ativo, e o robô deve *ficar parado* ou apenas caminhar, deve-se incluir recompensas de *seguimento de comando*. Por exemplo, recompensa de velocidade 0 quando não mandamos pular, etc. Contudo, se focarmos o treino só no salto (com comando sempre ativo), isso não se aplica. Para políticas unificadas multi-ação, veja a seção de múltiplas políticas.

Recompensa Total: Combine os termos acima de forma ponderada:

$$R_{\text{total}} = w_{\text{altura}} \cdot r_{\text{altura}} + w_{\text{prep}} \cdot r_{\text{agachamento}} + w_{\text{estab}} \cdot r_{\text{estabilidade}} + w_{\text{pouso}} \cdot r_{\text{pouso}} + w_{\text{energia}} \cdot r_{\text{energia}} + (\text{penalidades de q}$$

Ajuste os pesos w_i para balancear importâncias. Na prática, pode começar com pesos que garantam alta recompensa pelo objetivo principal (altura) comparado às penalidades, mas suficientes penalizações para impedir falhas graves. Considere também aplicar **shaping por estágios** dinamicamente: conforme o episódio avança, diferentes termos contam mais. Por exemplo, durante os primeiros instantes (preparação) enfatize r_{prep}, no meio do episódio (decolagem/vôo) dê mais peso a r_{altura} e r_{estab}, no final (pouso) aumente peso de r_{pouso}. Essa ideia de *stage-wise incentives* é comprovadamente eficaz para movimentos acrobáticos complexos [23] [24], pois simplifica o shaping tratando cada fase separadamente. Zhou *et al.* também usaram *incentivo de estágio* para ajustar a função de recompensa durante o salto, melhorando a estabilidade e precisão [18] [25] (sem tal mecanismo, eles notaram que o robô tendia a girar e falhar nos saltos).

Dica: mantenha a recompensa densa o suficiente – ou seja, forneça algum feedback em (quase) todos os passos, não apenas um sucesso/fracasso binário no fim, para facilitar a aprendizagem. Por exemplo, vá acumulando a recompensa de estabilidade e penalidade de esforço a cada passo, e use a altura instantânea como recompensa contínua enquanto estiver no ar. Pode-se dar um **bônus final** ao término do episódio se o robô conseguiu um salto "limpo" e pousou em pé (um impulso extra de motivação para concluir corretamente).

Estratégias de Treino: Curriculum e Agachamento como Checkpoint

Treinar um salto completo do zero pode ser difícil. Considere estratégias de **curriculum learning**: - **Incrementar Altura Gradualmente:** Comece treinando o robô para saltos pequenos. Uma forma simples: defina uma altura alvo baixa inicialmente (ex.: 10 cm). Depois de algumas centenas de iterações (quando a política começa a conseguir), aumente a altura alvo para 20 cm, e assim por diante, até a altura desejada. Isso evita que a política tente movimentos muito bruscos inicialmente que levam a quedas, permitindo aprendizado incremental. - **Treinar Agachamento Separadamente:** Você pode primeiro treinar uma política ou uma fase específica apenas para **agachar e manter equilíbrio**. Por exemplo, um pequeno treino supervisionado ou de RL onde a recompensa é apenas por abaixar o centro de massa sem cair. Embora não seja obrigatório (a política final pode aprender sozinha a agachar se recompensada por isso), essa fase isolada pode servir para inicializar a rede ou fornecer uma demonstração (no caso de usar *imitation learning* combinado). - **Dividir o Problema em Subtarefas:** Inspire-se em trabalhos de *aprendizado hierárquico*. Por exemplo, pesquisadores treinaram robôs para *backflips* segmentando o comportamento em fases e tratando-as com múltiplos objetivos [23]. No nosso caso, poderíamos conceber subpolíticas: uma para preparação (agachar), outra para extensão (salto) e pouso. No entanto, coordenar múltiplas subpolíticas em

sequência tem sua complexidade. Geralmente é mais simples moldar a recompensa para que uma única política realize toda a sequência de salto autonomamente (com o curriculum ou shaping adequado).

Ao aplicar curriculum, monitore se a política não "se acomoda" a saltos baixos e reluta em saltar mais alto. Se isso ocorrer, pode ser necessário reforçar a recompensa de altura nas fases posteriores do treino ou usar *reset* forçado do explorador de política.

Políticas Múltiplas vs. Política Unificada (Andar, Virar, Agachar e Saltar)

Como queremos também controlar o robô para **andar e virar com teclas WSAD**, surge a questão: devemos ter **políticas separadas para cada habilidade** ou **uma única política que faça tudo**? Há prós e contras em cada abordagem:

- **Políticas Separadas (Especialistas por Tarefa):** Mais simples de treinar cada comportamento isoladamente, com sua própria recompensa específica. Por exemplo, podemos ter:
- *Policy Walk*: treinada para seguir comandos de velocidade linear/angulação (W/S/A/D) e manter equilíbrio dinâmico.
- *Policy Jump*: treinada para executar um salto quando acionada (como detalhamos acima).
- *Policy Crouch*: (se desejado) treinada para abaixar o corpo (agachar) e talvez andar agachado ou permanecer estável em posição baixa.

Com isso, cada rede foca em um problema mais delimitado. Na hora da execução, um módulo supervisor (no caso, o próprio usuário/teclas) decide qual política está ativa: ex., se Espaço for pressionado, você **troca para a policy de salto**; quando solto, volta para a de caminhada. Essa arquitetura hierárquica é usada em pesquisas avançadas – por exemplo, o framework *ANYmal Parkour* treinou **cinco políticas especializadas** (andar, pular, subir obstáculos altos, descer obstáculos, agachar) e um nível superior de navegação decide qual acionar conforme o obstáculo [26]. No nosso caso, o *"nível superior"* é simplesmente o teleoperador humano com teclas, então é viável alternar manualmente. Uma vantagem dessa separação é que evita que um único modelo tenha que aprender dinâmicas muito diferentes (caminhar suave vs. saltar explosivo) simultaneamente.

- **Política Unificada (Multitarefa):** Aqui se treina uma única rede que, dada uma indicação de comando, realiza tanto andar quanto saltar. Isso significa incluir o comando de velocidade e também o comando de salto na entrada. Por exemplo, um vetor de comando [v_x, v_y, yaw, jump_flag] conforme mencionado. Durante o treinamento, teríamos que amostrar diferentes comandos: às vezes mandando velocidades (sem pulo) – recompensando andar; e ocasionalmente acionando `jump_flag` – recompensando salto. A recompensa teria que englobar ambos os contextos: seguir velocidades quando solicitado (como era feito na tarefa de locomoção base [27]), e pular quando solicitado (como na seção anterior). **Isso é mais complexo**, pois mistura dois objetivos distintos no mesmo processo de otimização. Pode demandar um algoritmo que suporte múltiplos objetivos ou pelo menos um shaping cuidadoso que não cause conflito (por ex., não querer que o robô tente pular quando o objetivo era apenas andar rápido). Pesquisas recentes abordam esse tipo de problema com RL multiobjetivo e composição de habilidades. Entretanto, para início, a solução de múltiplas políticas pode ser mais direta de implementar e depurar.

Recomendação: Treine **duas políticas separadas** inicialmente – uma de locomoção horizontal e outra de salto vertical. Isso permite desenvolver e testar cada uma isoladamente. A política de caminhada pode ser baseada na existente (ex.: usar a tarefa padrão do *unitree_rl_gym* para G1 ou Go1, que já considera comandos v_x, v_y, yaw e recompensas de equilíbrio/gait [27]). Nessa política de andar, as teclas W/S definiriam o v_x (frente/trás), A/D definem yaw (girar esquerda/direita), possivelmente Q/E poderiam definir v_y (lateral, se aplicável). O RL aprenderá a seguir esses comandos – de fato, no exemplo da Unitree, eles associaram as teclas W/S/A/D a comandos de velocidade linear/angulares e demonstraram que a política treinada controla a direção da marcha interativamente [28]. Essa política de andar deve incluir penalização de quedas e inclinação excessiva [27] para garantir estabilidade, além de recompensar velocidade na direção solicitada [27].

Em paralelo, a policy de salto foca apenas no ato de pular (provavelmente assumindo que o robô começa parado em posição neutra ou agachada). Poderíamos também integrar o **agachamento** como uma habilidade extra: por exemplo, treinar uma *policy crouch* para, ao pressionar uma tecla (digamos **C**), o robô baixar o corpo lentamente (útil para passar sob obstáculos). No entanto, note que um agachamento estático simples talvez nem precise de RL – poderia ser uma sequência predefinida de comandos nas juntas. Ainda assim, se quiser consistência via RL, pode treinar ou include a posição baixa como parte da política de andar (alguns trabalhos de parkour incluem agachar como um comando de velocidade vertical negativo, por ex.).

Transição entre Políticas: Caso use políticas separadas, é importante projetar como alternar sem problemas: - Quando o usuário apertar Espaço para pular, idealmente o robô deveria estar em uma postura estável para iniciar o salto. Se ele estiver andando rápido, talvez seja preciso brevemente **diminuir a velocidade** ou emitir um comando para pará-lo antes de trocar para a política de salto. Isso porque a policy de salto pode ter sido treinada assumindo partida da parada. Uma solução é: ao pressionar espaço, trave os comandos de movimento e sinalize para a policy de salto assumir controle; possivelmente faça o robô entrar em posição de agachamento (pode ser o próprio começo da policy de salto). - Durante o salto (desde o agachar, decolar, até aterrissar), mantenha a política de salto no comando. Após aterrissagem completa e estabilização, você pode voltar o controle para a política de andar. Poderá detectar a conclusão do salto talvez pelo tempo (ex.: 1-2 segundos após decolagem) ou quando os quatro pés estiverem de volta no chão e velocidade linear ~0. - Para virar (A/D) ou andar (W/S) durante locomoção normal, é simples: essas teclas apenas mudam o comando enviado para a policy de andar (ajustam a meta de velocidade). A policy de andar lida internamente em manter equilíbrio e caminhar naquela direção – como um controlador de alto nível de velocidade. De fato, a Unitree demonstrou que sua policy RL age como um **controlador de alto nível**, convertendo comandos simples ("vá para frente", "vire") em movimento coordenado das pernas [29] [30]. Sem essa policy, no Gazebo o robô não anda sozinho – mandar torques constantes faria ele cair; é a policy de RL que preenche essa lacuna [31].

Em suma, estruturar com múltiplas policies nos deixa com um *catálogo de habilidades*, parecido com o que pesquisadores implementam para robôs versáteis [32] [26]. No nosso caso, o "alto nível" é manual (teclado), mas no futuro poderia ser um planner automático escolhendo entre andar ou saltar conforme obstáculos (similar ao módulo de navegação no trabalho ANYmal Parkour [26]). Para começar, focaremos em fazer o *switch* manual suave e treinar cada habilidade isoladamente.

Treinamento da Política de Salto (Implementação Passo-a-Passo)

Vamos detalhar um **roteiro de etapas** para treinar a política de salto:

1. **Preparar o Ambiente de Simulação:** Certifique-se que o simulador (Isaac Gym ou MuJoCo) está instalado e configurado. Carregue o modelo do robô e verifique se ele fica de pé estavelmente no início. Desabilite qualquer controlador de alto nível existente – queremos que o RL controle as articulações. No caso do *unitree_rl_gym*, escolher a tarefa base apropriada (ex.: `g1` para humanoide G1) como ponto de partida é útil, pois já define observações e ações padrão [6] [7]. Você pode copiar essa configuração e editar especificamente a função de recompensa e talvez a terminação de episódio para adaptá-la ao salto.

2. **Definir Observação e Ação:** Inclua o comando de salto conforme discutido (pode ser um simples flag no estado). Se for treinar a policy de salto isoladamente, talvez fixe esse flag em 1 (ou nem inclua) – significando que aquela policy *sempre* tenta pular. As ações serão os ângulos das articulações das pernas (ou torques, se escolhido). Garanta que o agente tenha informação suficiente para aterrissar: por exemplo, incluir no estado a velocidade vertical do tronco e altura atual pode ajudar a julgar o momento do pouso. Estado resumido: [orientação (quaternion ou Euler), velocidades angulares, ângulos das juntas, velocidades das juntas, possivelmente contatos dos pés, etc., + *jump_flag*].

3. **Implementar a Função de Recompensa:** Codifique os termos descritos na seção anterior:

4. Cálculo da altura do COM ou dos pés.
5. Verificação de tombamento (por orientação ou contatos indevidos).
6. Cálculo de esforço (opcional).

7. Etc. Faça depuração imprimindo recompensas parciais para ver se fazem sentido. Ajuste pesos inicialmente de forma heurística. Exemplo inicial: altura_weight=5.0, prep_weight=1.0, stab_weight=1.0, landing_weight=2.0, energy_weight=0.001, fall_penalty=-5.0 (esses valores devem ser ajustados conforme unidade das métricas). Inclua lógica de término: `done = True` se tronco tocar o chão ou se tempo > T_max (ex.: T_max = 2 segundos se espera concluir salto nesse período).

8. **Inicialização do Treino:** Inicie o treinamento usando PPO. Monitore no console a **recompensa média** por episódio. No início, será muito baixa ou negativa (o robô provavelmente cai ou não sai do lugar). Verifique também eventos: ele está conseguindo agachar? Sair do chão? Muitas vezes no começo, a política explora movimentos aleatórios e cairá – isso é esperado [15].

9. **Acompanhamento e Ajustes:** À medida que o treino progride, você deve ver a recompensa média subir gradualmente. Se ficar zerada ou não evolui, possivelmente a tarefa está difícil demais – considere facilitar: por ex., aumente um termo de shaping (recompensar qualquer flexão de joelho para pelo menos guiar o comportamento inicialmente). Uma técnica: dar recompensa também por **pequenos saltos**, não só atingir altura alvo. Assim mesmo tentativas parciais acumulam alguma pontuação.

10. Se o robô fica se agachando mas não saltando, talvez aumente o incentivo de decolar (pode-se adicionar uma recompensa pequena pela velocidade vertical positiva do COM).

11. Se o robô salta mas cai de costas, aumente a penalização de rotação ou incentive pousar em pé (talvez recompensar se ângulo de perna no pouso está adequado).
12. Use ferramentas de visualização (se disponível) para assistir alguns episódios e entender o comportamento.
13. Ajuste hiperparâmetros de PPO se necessário: p.ex., diminuir o fator gamma (talvez 0.98) se recompensas importantes são no fim e você quer valorizar elas mais proximamente; aumentar o coeficiente de entropia se exploração estiver estagnando.
14. **Curriculum:** Quando a política começar a acertar saltos baixos consistentemente, você pode automaticamente elevar o objetivo de altura. Isso pode ser feito de forma incremental a cada N iterações (hard-coded) ou até dinamicamente: ex., se a taxa de sucesso > 80% em altura atual, aumente a meta. Esse processo pode repetir até a altura desejada final.

15. **Checkpointing:** Salve modelos periodicamente. Se um ajuste der errado (recompensa colapsar), você pode voltar a um checkpoint anterior.

16. **Convergência:** Saber quando parar. Critérios: a recompensa média se estabiliza perto de um valor máximo esperado (por ex., se a recompensa máxima teórica somando todos termos seria ~+10, e o agente está conseguindo ~8-9 consistentemente); o robô raramente cai nos episódios (digamos <5% das tentativas); visualmente, o comportamento está satisfatório (salta e aterrissa com estabilidade razoável). Pode ainda refinar por mais algumas milhares de iterações para ver se melhora marginais. Lembre-se de que adicionar mais iterações em tarefas difíceis como salto de humanoide pode ajudar, mas retornos decrescentes – evite overfitting também (monitorar se não começa a piorar).

17. **Teste Simples do Modelo:** Terminado o treinamento, pegue o checkpoint final (ou o melhor). Se treinou no Isaac Gym, uma boa prática do *unitree_rl_gym* é rodar um **teste em MuJoCo** com o modelo. Por exemplo, eles têm um script `deploy_mujoco.py` que carrega a política treinada no MuJoCo para ver como ela se sai [33]. Faça isso: coloque o robô no MuJoCo, inicialize a política e veja se ele realiza o salto (provavelmente com alguma interface de tecla ou simplesmente automaticamente). Isso valida a robustez em outro simulador. Espere alguma degradação de performance (talvez saltos um pouco menores ou instabilidade) devido a diferenças físicas [2] – se for grave, pode indicar necessidade de mais randomização de domínio no treino ou tunagem ao simulador destino.

Integração no Gazebo com ROS2 (Controle por Teclado)

Com o modelo treinado pronto, o passo final é integrá-lo na simulação **Gazebo** usando **ROS2**, permitindo o controle via teclado (WSAD + Espaço). Vamos estruturar essa integração em componentes:

1. Preparar Simulação no Gazebo

- **URDF e Controladores:** Insira o modelo URDF do robô no Gazebo. Configure plugins de controle nas juntas. Uma forma simples é usar o pacote `gazebo_ros2_control` com, por exemplo, controladores de posição para as articulações das pernas. Isso nos permite publicar ângulos desejados e o Gazebo aplicará torques para atingi-los. Alternativamente, use controladores de esforço (torque) se preferir controle mais direto – mas então talvez precise aplicar a lei de controle PD manualmente em cima (ver próximo item).

- **Spawning:** Crie um `launch` que lance o Gazebo com o robô. Certifique-se de que o robô inicia numa pose estável (ex.: em pé no chão). Desabilite gravidade ou fixar no ar *não* é desejado – queremos gravidade normal para ver o comportamento real.
- **ROS2 Interfaces:** Verifique que você consegue obter os estados do robô via ROS2 (p. ex., assinando em `/joint_states` para ângulos atuais, ou usando APIs do control interface para obter feedback). O Gazebo publicará estados se os controladores estiverem configurados para isso.

2. Nó de Controle com a Política RL

Crie um nó (pode ser em Python usando rclpy, ou em C++ usando rclcpp). Esse nó será o **"cérebro"** que roda a inferência da rede treinada: - **Carregar o Modelo RL:** Use a biblioteca de ML correspondente. Por exemplo, se a policy foi treinada em Python/PyTorch, você pode salvar o modelo como `.pt` (scripted or state_dict) e então carregar no ROS2 Python node usando `torch.load`. Alternativamente, para desempenho, você pode exportar para TorchScript or ONNX. Em C++, usar LibTorch é uma opção [34], conforme o exemplo do Unitree (eles fornecem um executável C++ com LibTorch para controle do G1 real) [34]. Em nosso caso, Python é mais simples inicialmente. - **Assinar Estado do Robô:** O nó deve subscrever tópicos do estado necessário para montar a observação. Você pode subscrever `/joint_states` (para posições e velocidades das articulações) e talvez um tópico de IMU para orientação (ou usar /odom se disponível para base pose/twist). Cada vez que receber novos estados, armazene-os (talvez em variáveis globais ou atributos do nó). - **Assinar Comandos do Usuário:** Precisamos receber os comandos de teclado (WSAD, Espaço). Podemos fazer isso de duas formas: - Usar um nó de teleop pronto, como `teleop_twist_keyboard` que publica mensagens de velocidade (geometry_msgs/Twist) em um tópico, digamos `/cmd_vel`. Nesse modo, você mapearia WSAD para linear.x e angular.z do Twist. Por padrão, teleop_twist_keyboard usa teclas do cursor, mas é configurável e WASD costuma controlar linear e angular velocities. Se usar esse, então no seu nó, subscreva `/cmd_vel` para obter comandos de velocidade *desejada*. - Lidar diretamente com eventos de teclado: há pacotes que capturam key strokes e publicam por exemplo uma mensagem custom (ex.: std_msgs/Char ou algo). O projeto *rl_sar* mencionado faz algo assim: lá, ao apertar certa tecla, eles alternam modos [35] e associaram W/S/A/D internamente no nó rl_sim para ajustar velocidades [28]. Você poderia incorporar lógica semelhante: por exemplo, usar a biblioteca curses ou similar para ler teclado (mas isso só funciona se rodar em um terminal dedicado). - Dado que Twist via teleop é simples, vamos imaginar que W-> +v_x, S-> -v_x, A-> +yaw, D-> -yaw. Espaço não tem representação em Twist, então podemos mapear **Espaço** para um evento separado – talvez pressionar espaço could set `linear.z` in the Twist or use another topic/flag. Simplesmente, podemos criar um tópico custom, e modificar o teleop node para publicar uma boolean em, digamos, `/jump_cmd` when space is pressed.

Para fins deste guia, suponha que temos um *flag de pulo* acessível no nó (pode ser através de um tópico latched que envia True on press, or the node itself captures it). - **Laço de Controle (Inference Loop):** No nó de controle, execute um loop a ~100 Hz (10ms interval). A cada iteração: - Leia a última observação do estado do robô (ângulos, vel, orientação, etc) e o último comando do usuário (v_x, yaw desejado, e flag de pulo). - Monte o tensor de observação exatamente como esperava durante treino (mesma ordem e normalização de inputs). Por ex., `[joint_angles, joint_vels, base_orientation, base_angvel, command(vx,vy,yaw), jump_flag]`. - Passe esse tensor na *policy neural network* para obter a ação. **Importante:** use o modelo *treinado* e fixado (no tempo de execução não exploramos, apenas usamos a política determinística ou com leve estocasticidade se desejado). Tipicamente, em RL deployment, usa-se a média da ação aprendida (sem ruído) para controle mais estável. Se seu modelo é estocástico, você pode muestrear ações, mas para

reprodução confiável, fixe uma semente ou use a média. - Obtenha as ações (ex.: vetor de 12 ângulos desejados). - **Publicar comandos nas juntas:** Se você tem controladores de posição no Gazebo, publique as posições alvo em seus tópicos. Por exemplo, muitos setups têm um tópico por junta ou um único tópico de tipo `sensor_msgs/JointState` ou `trajectory_msgs/JointTrajectory`. Simplesmente enviar um JointState com nome das juntas e posições pode funcionar se o controller as aceita como referência. No projeto rl_sar, menciona-se que se configurado controladores de posição, basta publicar os ângulos desejados em um tópico de comando conjunto [36]. Ajuste para o seu controlador específico.

- **Opção com torque:** Se optou por controladores de esforço, então você precisará converter o ângulo alvo em torque via um **controlador PD externo** no laço. Isto é: para cada junta, calcule `torque = Kp*(pos_d - pos_atual) + Kd*(vel_d - vel_atual)` [37]. Aqui pos_d é o ângulo alvo da policy, pos_atual e vel_atual vêm dos sensores, e Kp/Kd são ganhos escolhidos (idealmente semelhantes aos usados na simulação de treino). Então publique esses torques em tópicos de esforço das juntas. Essa replicação PD é o que o Isaac Gym fazia internamente, então mantém o comportamento consistente [38] [39].

- Mantenha atenção a *nomes e ordem das juntas*: a ordem do vetor de ação da policy deve corresponder exatamente à ordem das juntas no robô simulado. Pequenos erros de mapeamento ou unidade (graus vs radianos) podem causar comportamento explosivo [40] [41]. Verifique duas vezes que a política espera ângulos em radianos (provável) e seu controlador Gazebo também (URDFs usualmente usam radianos). Mapeie os índices corretamente – se necessário, crie um array com nomes das juntas na ordem da policy e publique de acordo.

- **Modo de Operação:** Se está integrando múltiplas policies (andar vs pular), o loop precisará saber qual policy usar em cada momento. Pode-se gerir um estado interno: `mode = WALK` ou `mode = JUMP`. Inicialmente `mode=WALK`. Quando jump_flag for acionado (Espaço pressionado) e o robô estiver pronto, troque para `mode=JUMP` e talvez ignore quaisquer comandos de movimento enquanto `mode=JUMP`. Mantenha `mode=JUMP` por, digamos, 1 segundo ou até detectar aterrissagem completada (pode usar um timer ou monitorar se o robô teve todos pés no chão de novo e velocidade vertical ~0 após um pico). Então automaticamente volte `mode=WALK`. Essa lógica garante que o salto seja executado por completo sem interrupção.

- Caso a política unificada, não há troca de modelo – você sempre usa a mesma rede. Nesse caso, você simplesmente alimenta o jump_flag junto com vel desejada no input. Quando espaço for pressionado, `jump_flag=1` e talvez você zere temporariamente os outros comandos de velocidade se for pretendido que o salto seja feito parado ou independente. A própria rede, se treinada apropriadamente, deve reconhecer o flag e produzir ações de salto. Após alguns passos, você pode resetar `jump_flag=0` (ex.: quando um certo tempo passou ou o pico de salto já ocorreu). O desafio aqui é temporal: o RL pode precisar do flag ativo durante toda a manobra. Uma abordagem é definir que ao apertar espaço, você mantenha `jump_flag=1` para, por exemplo, os próximos 1 segundo de controle, depois volte a 0. Assim a política sabe que está no "modo pulo" durante aquele período.

- **Frequência de Loop:** Mantenha ~100 Hz se possível, para correspondência com treino. Use timers do ROS2 ou um loop while com Rate. Muito lento (tipo 10 Hz) pode prejudicar equilíbrio [5], pois a política espera atualizações rápidas. Ajuste `RCLCPP_INFO` ou logs para ter certeza que está mantendo a taxa.

3. Teste e Depuração no Gazebo

Com o nó de controle rodando e o Gazebo com o robô ativo, realize os seguintes testes: - **Verificar Andar:** Pressione W (ou envie um /cmd_vel linear.x) e veja se o robô começa a marchar para frente. Se a política de andar foi bem treinada e integrada, o G1/G0 quadrúpede deve começar a caminhar de forma equilibrada na direção solicitada [42] [43]. Teste A/D para virar enquanto anda, S para ir para trás. Ajuste a escala do comando de velocidade se necessário (talvez 1.0 corresponda a 1 m/s – se for muito, reduza). - **Pressionar Espaço (Salto):** Traga o robô a uma parada (ou ao menos velocidade baixa), então pressione Espaço. O nó deve trocar para policy de salto. Observe o robô: idealmente deve agachar e então impulsionar-se para um salto. Com sorte, ele deixará o chão e pousará sem tombar. Se o salto foi pequeno ou tímido, talvez o comando não foi interpretado (verifique se jump_flag estava realmente ativado no input da rede). Se o robô pulou mas caiu, pode ser necessário refinar a política ou verificar discrepâncias (ver abaixo). - **Cenário de Agachamento:** Se houver uma tecla para agachar (C, por ex.), teste-a se implementado. O robô deve reduzir altura sem colapsar. - **Transições:** Teste sequência: andando -> espaço (pular) -> voltar a andar. O robô consegue retomar caminhada após o salto? Alguns segundos pós-pulo, envie comando de andar novamente, ou se sua lógica retorna auto para WALK mode, tente apertar W logo após o pouso. Se houver instabilidade na transição (ex.: ao aterrissar ele dá um passo em falso), considere implementar uma pequena espera ou usar a própria policy de andar para recuperar equilíbrio logo após pouso. Alternativamente, treinar a policy de salto até ela mesma aprender a estabilizar pós-pouso ajuda.

Se o robô no Gazebo **tremer muito ou cair frequentemente**, podem existir algumas causas e soluções: - **Frequência de controle baixa:** como mencionado, tentar rodar a inferência mais rápido (200Hz) pode ajudar na estabilidade [5]. - **Diferenças de física:** O Gazebo normalmente usa o motor ODE ou DART, enquanto o treino pode ter sido em PhysX (Isaac) ou MuJoCo [44]. Diferenças em contato, amortecimento, comportamento do solo, podem fazer a policy agir sub-otimizada. Mitigações: ajustar parâmetros do Gazebo (ex.: solver iterations, ERP, friction) ou adicionar leve amortecimento extra nos controladores de junta [44]. - **Modelo distinto:** Certifique-se que o modelo no Gazebo corresponde ao treinado. Exemplo real citado: se durante o treino não incluímos os braços/mãos do humanoide mas no Gazebo o URDF tem as mãos com física, isso altera a inércia e equilíbrio [45]. Solução: remover ou desabilitar certas partes no Gazebo para equivaler ao modelo de treino, ou treinar novamente considerando-as [46]. - **Calibração de ângulos:** Verifique offsets – às vezes o "zero" de uma articulação no URDF não era exatamente o mesmo ângulo considerado zero no modelo de treino [47]. Se houver essa discrepância, a posição estacionária aprendida pode não corresponder à posição real, causando inclinação. Você pode ajustar manualmente adicionando um offset nas ações ou refazendo a calibração do URDF.

Itere nesses ajustes até o comportamento no Gazebo ficar satisfatório. Uma vez que **o robô anda de forma inteligente guiado por RL e salta ao comando**, você terá validado a transferência simulação-para-simulação (Sim2Sim) do modelo [48]. Esse processo é análogo ao feito internamente no unitree_rl_gym ao passar do Isaac Gym para MuJoCo, mas aqui fizemos manualmente para o Gazebo/ROS [49].

4. Futuro: Integração com ROS2 e Expansão

Agora que o sistema responde a teclas como um "jogo", pense em como isso se integra no ecossistema ROS2: - Você pode criar um **Node de Teleop Personalizado** que publica tanto `cmd_vel` quanto um sinal de `jump` ao apertar espaço, encapsulando o controle em um só lugar (ou até usar um joystick/gamepad para controlar o robô). - Estruture os nodes de forma modular: por exemplo, um node `rl_walker` que executa a policy de andar (subscrito em cmd_vel), e um node `rl_jumper` para o salto (ativado por um

serviço ou tópico quando necessário). Em vez de alternar dentro de um mesmo node, poderia haver um coordenador que congela um e ativa outro. Dependendo do design, isso pode ou não ser mais complicado que a abordagem única. - Considere usar mensagens customizadas ou sinais de modo. Por exemplo, um tópico `/mode_switch` que pode ser "WALK" ou "JUMP" e o node de controle unificado sabe qual ação priorizar. Ou um serviço `/jump_now` que quando chamado, executa a sequência de salto. - **ROS2 Lifecycle:** Se pensando em um sistema robusto, você pode ter os nodes em estado de standby até o simulador estar pronto, etc., mas para protótipo não é crítico. - **Simulação vs Real:** O bacana é que o mesmo nó de controle usado no Gazebo pode, em teoria, controlar o robô real (como o Unitree A1/Go1 ou G1) – basta redirecionar os comandos para o SDK do robô real ao invés do Gazebo. No unitree_rl_gym, eles enviam comandos via UDP para o robô real usando a mesma policy treinada [34] [50]. Em nosso caso, usando ROS2, poderíamos mandar para um driver ROS do robô real.

Por fim, com o sistema funcionando no Gazebo, você pode testar cenários variados: andar e pular sobre obstáculos virtuais, agachar sob barreiras, etc. O grande benefício é que **todo o controle de alto nível está emergindo da política RL**, que aprendeu a rastrear comandos – se o treinamento considerou comandos de velocidade e salto na entrada e recompensou seguir corretamente, o resultado é um robô que **vai para frente quando você aperta W, vira com A/D e salta com Espaço, de forma natural**. Essa correspondência entre comando e ação ocorre porque no treino incluímos o comando na observação e demos recompensa pelo cumprimento dele [30] [43] – ou seja, a policy aprendeu a **associar "ir para frente" com os movimentos adequados** de pernas, sem precisarmos programar isso manualmente [51].

Conclusão

Neste guia, percorremos todos os passos para desenvolver um comportamento de salto controlado por tecla em um robô legged via aprendizado por reforço, incluindo considerações de treinamento (ambiente, observações, ações, algoritmo PPO, função de recompensa bem estruturada) e integração prática em ROS2/Gazebo para controle interativo. O processo requer cuidadosa definição de recompensas – especialmente para ações complexas como saltar – e possivelmente divisão em múltiplas policies para comportamentos distintos. Com uma função de recompensa adequada e treino suficiente, o robô aprende a **agachar, impulsionar e pousar em pé** ao receber o comando de pular, atendendo ao objetivo definido. Utilizando ROS2, conectamos a política treinada ao mundo simulado, permitindo comandar o robô por teclas WSAD (locomoção) e Espaço (salto) de forma semelhante a controlar um personagem de videogame, mas aqui aplicando comandos físicos a um robô virtual (e escalável ao robô real no futuro).

O resultado final é um sistema onde o **Aprendizado por Reforço fornece um controlador de alto nível** robusto: no Gazebo, o robô **não possuiria, por si só, lógica de andar ou pular**, apenas recebe torques nas juntas – mas ao plugarmos a política de RL treinada, ele ganha *inteligência motora* e reage aos nossos comandos de maneira estável e eficaz [31]. Isso demonstra o poder do RL em aprender habilidades motoras complexas e abre caminho para adicionar ainda mais comportamentos (corrida, subir escadas, etc.) seguindo metodologia similar.

Com essa base implementada, você pode expandir para incluir percepção (sensoriamento do ambiente) e comandos mais altos níveis. Por exemplo, integrar uma câmera e usar outra rede para decidir *quando* pular (detecção de obstáculo) – embora isso já entre em navegação autônoma. Para o escopo atual, você tem em mãos um robô simulado capaz de ser pilotado por comandos de teclado, combinando *políticas de andar e saltar aprendidas*. Boa sorte nos experimentos, e lembre-se que ajustes finos na simulação e inúmeras

iterações de treinamento podem ser necessários para atingir a performance desejada – faz parte do processo de desenvolvimento de robôs com aprendizado por reforço!

Referências Utilizadas: As informações e estratégias aqui apresentadas baseiam-se em orientações do guia *unitree_rl_gym* [2] [9], em pesquisas recentes sobre controle de salto com RL [17] [18], bem como na experiência prática de integração de políticas RL em ROS2/Gazebo [28] [29]. Essas referências reforçam a importância de um bom shaping de recompensa e mostram casos de sucesso de políticas aprendidas controlando robôs reais e simulados. Em especial, a demonstração da Unitree de controle via teclas confirma a viabilidade de nosso objetivo [28] [31], e trabalhos como ANYmal Parkour exibem a utilização de múltiplas policies coordenadas para locomover-se agilmente (andar, pular, escalar) [32] [26], inspiração direta para nossa abordagem multi-habilidade. Em suma, apoiamo-nos tanto na teoria quanto em implementações práticas existentes para delinear este guia completo de treinamento e deployment do comando de salto por RL.

---

[1] [2] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [27] [28] [29] [30] [31] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] Guia de Locomoção com __unitree_rl_gym__ e Aprendizado por Reforço.pdf
file://file-FVLJurZ9NvoHCTkZQdxrtr

[3] [PDF] gym-gazebo2, a toolkit for reinforcement learning using ROS 2 and ...
https://arxiv.org/pdf/1903.06278

[17] [18] [19] [20] [21] [22] [25] Stable Jumping Control Based on Deep Reinforcement Learning for a Locust-Inspired Robot
https://www.mdpi.com/2313-7673/9/9/548

[23] [24] Stage-Wise Reward Shaping for Acrobatic Robots: A Constrained Multi-Objective Reinforcement Learning Approach
https://arxiv.org/html/2409.15755v1

[26] [32] ANYmal Parkour: Learning Agile Navigation for Quadrupedal Robots
https://arxiv.org/html/2306.14874v1