COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

AGILE VISION-BASED FLIGHT FOR DRONES USING NEURAL NETWORKS

Autor: Pablo Paramio Valdés

Director: Jesús Tordesillas Torres

Madrid

Agosto de 2025

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Agile Vision-Based Flight for Drones Using Neural Networks

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2024-2025 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Pablo Paramio Valdés Fecha: 27/08/2025

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Jesús Tordesillas Torres

Digitally signed by Jesús Tordesillas Torres
Date: 2025.08.28 17:58:05 +02'00'

Fdo.: Jesús Tordesillas Torres Fecha: 28/08/2025

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

AGILE VISION-BASED FLIGHT FOR DRONES USING NEURAL NETWORKS

Autor: Pablo Paramio Valdés

Director: Jesús Tordesillas Torres

Madrid

Agosto de 2025

VUELO ÁGIL DE DRONES BASADO EN VISIÓN USANDO REDES NEURONALES

Autor: Paramio Valdés, Pablo.

Director: Tordesillas Torres, Jesús.

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

1. INTRODUCCIÓN

El proyecto aborda el problema del vuelo ágil autónomo de drones cuadricópteros basado en visión. Aunque este tipo de autonomía ya se ha alcanzado mediante tecnologías como el GPS, su uso resulta poco fiable en entornos complejos o a altas velocidades, donde la señal es lenta e imprecisa [1]. Como alternativa, surge el vuelo basado en visión con inteligencia artificial.

En este ámbito se distinguen principalmente dos enfoques:

1. **Políticas de red neuronal híbridas**, que dividen el problema en dos etapas: la percepción o estimación explícita del estado a partir de imágenes y el control o toma de decisiones en función de ese estado [2].
2. **Políticas de red neuronal de extremo a extremo**, en las que el agente aprende a generar señales de control directamente desde la cámara a bordo, imitando la forma en que un piloto FPV maneja el dron [3].

Este proyecto se centra en la segunda variante, ya que resulta más prometedora para vuelo ágil, donde la estimación explícita de estado presenta dificultades [4], y se evita recurrir a sensores costosos.

El objetivo principal es el desarrollo de un marco de simulación en Nvidia Isaac Lab [5] que sirva como base para investigaciones futuras en vuelo autónomo basado en visión. Para ello, se han implementado tres componentes clave:

1. **Un robot de cuadricóptero y su respectivo modelo aerodinámico** realista dentro del marco de Isaac Lab.
2. **Un controlador de vuelo** tipo Betaflight [6] paralelizable, que asegura la compatibilidad con drones FPV reales.
3. **Un entorno de entrenamiento** con obstáculos, en el que se prueban tanto políticas basadas en estados como en visión con técnicas de reinforcement learning (RL).

Más que buscar un nuevo estado del arte en arquitecturas neuronales, este trabajo proporciona herramientas y metodologías abiertas que faciliten futuros avances en el vuelo autónomo ágil basado en visión.

2. METODOLOGÍA

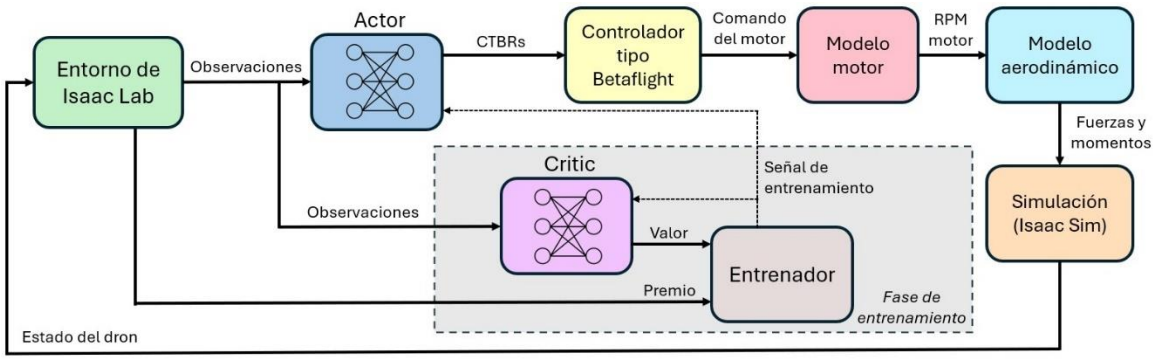

Figura 1: Esquema general del flujo de datos del proyecto

El esquema mostrado en la Figura 1 muestra el flujo de datos del proyecto completo. En él, se muestra cómo se utiliza una arquitectura simétrica de Actor-Critic (*actor-crítico*), en la que el actor aprende a generar un comando de empuje colectivo (collective thrust) y velocidades angulares (body rates) (CTBRs) a partir de las observaciones y el crítico aprende a estimar el valor para un estado dado y es utilizado durante el entrenamiento del agente.

1. Robot de cuadricóptero y modelo aerodinámico

En Isaac Lab, los entornos de cuadricópteros se basan por defecto en la clase *Articulation*, que simplemente aplica pares y fuerzas externas directamente sobre el dron, en lugar de modelar como cada motor individual genera fuerza y par de guiñada (yaw).

Para resolver esto, el proyecto introduce un nuevo tipo de asset denominado *Multicopter*, que recibe como entrada las velocidades angulares de cada motor y calcula las fuerzas y momentos aerodinámicos resultantes, reproduciendo fielmente el comportamiento que tendría el dron en la realidad. Esta parte del desarrollo se realizó con la ayuda de Mario Gómez Andreu, colaborador de la Universidad Pontificia Comillas.

El modelo aerodinámico se basa la suposición de una proporción cuadrática entre la velocidad angular del motor y la fuerza y momento que éste genera:

$$f_i = k_f \cdot \omega_i^2$$

$$\tau_z = \sum_{i=1}^{N} k_m \cdot d_i \cdot \omega_i^2$$

donde k_f es el coeficiente de empuje, k_m es el coeficiente del momento y d_i es una variable que vale 1 o -1 según el sentido en el que gira el motor i.

Todos los parámetros físicos y aerodinámicos (número de rotores, direcciones de giro, coeficientes de empuje y momento, inercia y posiciones de los rotores) se almacenan en una clase de configuración (*MulticopterCfg*). Esto permite reutilizar el mismo código de simulación en diferentes diseños de multirrotores simplemente modificando los ficheros de configuración.

La implementación se reparte en varios scripts:

- *multicopter.py*: lógica central de simulación y función de modelo aerodinámico.
- *multicopter_cfg.py*: parámetros aerodinámicos y geométricos.
- *quadcopter.py*: configuraciones específicas de cuadricópteros, con la definición de *CRAZYFLIE_CFG_MULTICOPTER* como sustituto del modelo basado en *Articulation*.

Este nuevo modelo de multirotor mejora notablemente el realismo de la simulación y reduce la brecha entre el entrenamiento en simulación y la implementación en drones reales.

2. Controlador tipo Betaflight

A diferencia de lo habitual en proyectos de investigación similares, en los que se utiliza un controlador de vuelo personalizado, uno de los objetivos de este proyecto es desarrollar uno que simule Betaflight, el controlador de vuelo más comúnmente utilizado en drones FPV. Esto permite que la política entrenada se use en cualquier dron con Betaflight instalado, sin necesidad de que se instale un controlador de vuelo personalizado. Aunque ya existe Betaflight SITL [7], que permite implementar el controlador en simulaciones, éste tiene la gran limitación de no soportar la computación paralela, esencial en aplicaciones de RL como ésta.

El controlador desarrollado utiliza la librería Pytorch para hacer las operaciones de control necesarias con tensores por lotes (*batches*). Se basa en un PID con acción diferencial sobre la salida:

$$\boldsymbol{pid}_{\text{sum}}(t) = \boldsymbol{p}(t) + \boldsymbol{i}(t) + \boldsymbol{d}(t)$$

$$\boldsymbol{pid}_{\text{sum}} \in \mathbb{R}^{\text{N}\times 3}$$

donde N es el numero de entornos paralelos simulados (*num_envs*). Para obtener la señal de control del motor (comando), se utiliza una matriz de mezcla (*mixer matrix*):

$$M = \begin{array}{c} \\ \text{FR} \\ \text{RR} \\ \text{RL} \\ \text{FL} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Roll} & \text{Pitch} & \text{Yaw} \end{array} \\ \begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & 1 & 1 \\ 1 & -1 & -1 \end{bmatrix} \end{array}$$

$$\boldsymbol{mix}(t) = \boldsymbol{pid}_{\text{sum}}(t) \cdot \boldsymbol{M}^{\top}$$

$$\boldsymbol{mix}(t) \in \mathbb{R}^{N \times 4}$$

Una vez se tiene los comandos, se aplica un modelo del motor utilizando la siguiente fórmula para obtener la velocidad de cada rotor:

$$\boldsymbol{\omega}(t) = \min\left(\boldsymbol{motor\_command}(t)^{\odot \alpha} \odot \boldsymbol{\omega}_{\max},\ \boldsymbol{\omega}_{\max}\right)$$

donde α es un parámetro experimental, $\odot\, \alpha$ representa una potencia elemento a elemento y ω_{max} es la velocidad máxima de cada rotor.

3. Entorno de Isaac Lab

El entorno utilizado para entrenar al agente consiste en un pasillo de 10 metros de largo, 2 metros de ancho y 1 metro de alto. El pasillo contiene 3 obstáculos en forma de cubo que cambian de posición aleatoriamente cada vez que el dron se choca o llega hasta el final, introduciendo *domain randomization*. La Figura 2 muestra una imagen exterior del pasillo con el techo eliminado para una mayor visibilidad. El objetivo del dron es atravesar el pasillo lo más rápido posible sin chocarse con ninguna pared u obstáculo.

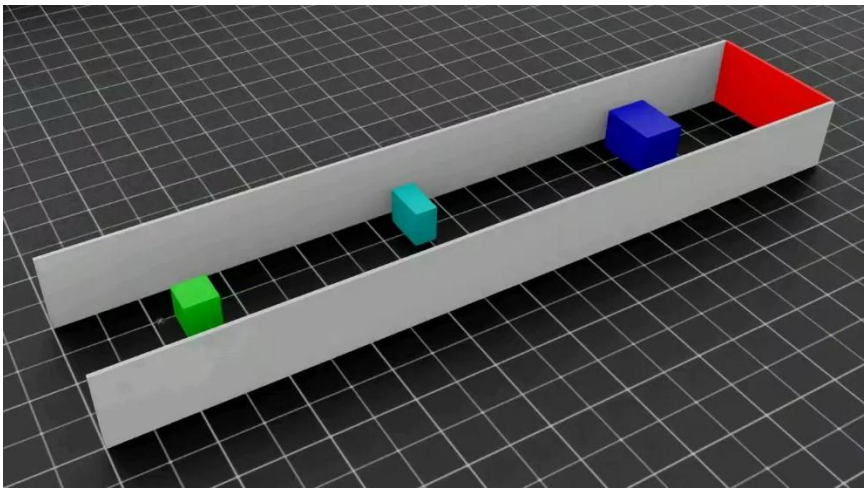

Figura 2: Imagen del pasillo utilizado para el entrenamiento del agente con el techo eliminado para una mejor vista.

Para evaluar tanto el entorno como el modelo aerodinámico y el controlador de vuelo, se realizan dos experimentos de entrenamiento de agente: uno con observaciones del estado actual de la simulación, y otro basado puramente en visión, con las imágenes de una cámara montada en el dron.

Para el caso del experimento basado en estados, las observaciones son las siguientes:

- **Orientación** (alabeo, cabeceo y guiñada):

$$\boldsymbol{\theta} = (\phi, \vartheta, \psi) \in \mathbb{R}^3$$

- **Velocidad angular:**

$$\dot{\boldsymbol{\theta}} = (\dot{\phi}, \dot{\vartheta}, \dot{\psi}) \in \mathbb{R}^3$$

- **Velocidad lineal:**

$$\boldsymbol{v_b} = (v_x, v_y, v_z) \in \mathbb{R}^3$$

- **Vector unitario de gravedad proyectada**, que indica la orientación con respecto al marco global:

$$\boldsymbol{g_b} \in \mathbb{R}^3, \quad \|g_b\| = 1$$

- **Vector de distancias a los límites del pasillo** (paredes, techo y suelo):

$$\boldsymbol{d}_{\text{right}} \in \mathbb{R}^3 \quad \boldsymbol{d}_{\text{left}} \in \mathbb{R}^3 \quad \boldsymbol{d}_{\text{ceil}} \in \mathbb{R}^3 \quad \boldsymbol{d}_{\text{ground}} \in \mathbb{R}^3$$

- **Vector de dirección hacia la posición objetivo** en coordenadas relativas**:**

$$\boldsymbol{d}_{\text{target}} \in \mathbb{R}^3, \quad \|\boldsymbol{d}_{\text{target}}\| = 1$$

- **Posición relativa del obstáculo** siguiente más próximo:

$$\boldsymbol{d}_{\text{obs}} \in \mathbb{R}^3$$

- **Acción previa** ejecutada por el agente, para aportar consistencia temporal:

$$\boldsymbol{a_{t-1}} \in \mathbb{R}^4$$

Por otro lado, en el caso del **experimento basado en visión**, todas las observaciones estarían disponibles en la implementación real:

- **Imagen RGB** de 60x60 captada por la cámara frontal del dron. Esta imagen es procesada por una MobileNetV3-small [8], una red neuronal previamente entrenada que extrae las características principales de la imagen. Esto genera 576 mapas de características que se reducen a un vector de dimensión 576 tras hacer *average pooling*:

$$\boldsymbol{f}_{\text{rgb}} \in \mathbb{R}^{576}$$

- **Orientación** (alabeo, cabeceo y guiñada):

$$\boldsymbol{\theta} = (\phi,\, \vartheta,\, \psi) \in \mathbb{R}^3$$

- **Velocidad angular:**

$$\dot{\boldsymbol{\theta}} = (\dot{\phi},\, \dot{\vartheta},\, \dot{\psi}) \in \mathbb{R}^3$$

- **Vector unitario de gravedad proyectada:**

$$\boldsymbol{g_b} \in \mathbb{R}^3, \quad \|g_b\| = 1$$

- **Vector de dirección hacia la posición objetivo** en coordenadas relativas:

$$\boldsymbol{d}_{\text{target}} \in \mathbb{R}^3, \quad \|\boldsymbol{d}_{\text{target}}\| = 1$$

- **Acción previa** ejecutada por el agente:

$$\boldsymbol{a_{t-1}} \in \mathbb{R}^4$$

Por último, para poder guiar correctamente el entrenamiento del agente y así conseguir que el dron atraviese el pasillo exitosamente, una función de premio densa es utilizada. En cada interacción de la política de control con el entorno, el premio es calculado con la siguiente función:

$$r_t = r_{\text{progress}} + r_{\text{end}} + r_{\text{collision}} + r_{\text{orientation}} + r_{\text{smooth}}$$

Donde cada parte del premio se calcula de la siguiente manera:

- **Progreso:**

$$r_{\text{progress}} = \Delta p \cdot \alpha_{\text{progress}}$$

  Donde Δp representa el progreso que se ha hecho a lo largo del pasillo en el último paso de simulación y $\alpha_{progress}$ es un factor de escala positivo.

- **Llegada al final:**

$$r_{\text{end}} = \mathbf{1}_{\text{end}} \cdot \alpha_{\text{end}} \left(1 - \frac{n_t}{N_{\text{max}}}\right)$$

  Donde $\mathbf{1}_{end}$ es un indicador que vale 1 cuando el dron ha llegado al final y 0 en caso contrario, n_t es el número de pasos que ha dado la simulación, N_{max} es el número de pasos máximo por intento y α_{end} es un factor de escala positivo.

- **Penalización por colisión:**

$$r_{\text{collision}} = -\mathbf{1}_{\text{collision}} \cdot \alpha_{\text{collision}}$$

Donde $\mathbf{1}_{collision}$ es un indicador que vale 1 cuando el dron ha chocado y 0 en caso de lo contrario y $\alpha_{collision}$ es un factor de escala positivo.

- **Orientación:**

$$r_{\text{orientation}} = e^{(4 \cdot d_{\text{target},x} - 3)} \cdot \alpha_{\text{orientation}} \cdot \Delta t$$

Donde $d_{target,x}$ es el componente x del vector de dirección hacia el objetivo, $\alpha_{orientation}$ es un factor de escala positivo y Δt es el intervalo de simulación.

- **Suavidad de control:**

$$r_{\text{smooth}} = -\left\|\boldsymbol{a_t} - \boldsymbol{a_{t-1}}\right\|^2 \cdot \alpha_{\text{smooth}} \cdot \Delta t$$

Donde a_t es la acción actual, a_{t-1} es la acción pasada, α_{smooth} es un factor de escala positivo y Δt es el intervalo de simulación.

Los valores de los factores de escala se obtienen de forma empírica y ofrecen un balance entre velocidad, estabilidad, suavidad y seguridad durante el vuelo:

$$\alpha_{\text{collision}} = 50.0, \quad \alpha_{\text{progress}} = 6.0, \quad \alpha_{\text{end}} = 80.0, \quad \alpha_{\text{orientation}} = 5.0, \quad \alpha_{\text{smooth}} = -5.0$$

4. Arquitectura de la red neuronal

Se utiliza un algoritmo simétrico de Actor-Critic, donde la estructura de ambas redes (actor y crítico) es casi igual y solo cambia su salida. Específicamente, el agente se entrena utilizando *Proximal Policy Optimization* (PPO) [9].

Política de control. La política está modelada como una distribución normal, parametrizada por una media y una desviación típica logarítmica.

$$\pi_\theta(a_t \mid s_t) = \mathcal{N}(\mu_\theta(s_t),\, \sigma^2),$$

donde s_t es el estado (vector observación) en t, a_t es la acción (CTBRs) y μ_θ se obtiene a través de la red neuronal del actor, formada por capas ocultas de 512, 256 y 128 neuronas y activaciones lineales exponenciales (ELU). Las arquitecturas para el actor para la política basada en estados y basada en visión se muestran en la Figura 3 y Figura 4 respectivamente. La desviación estándar logarítmica, en lugar de ser predicha por la red neuronal, es mantenida como un parámetro entrenable aparte.

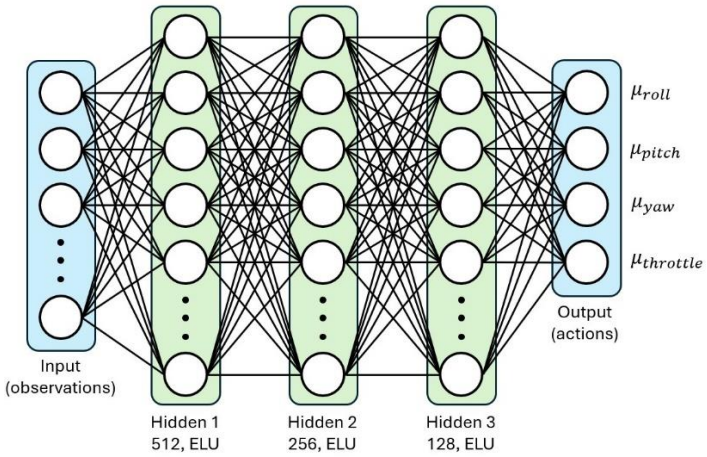

Figura 3: Red neuronal del actor para vuelo basado en estados

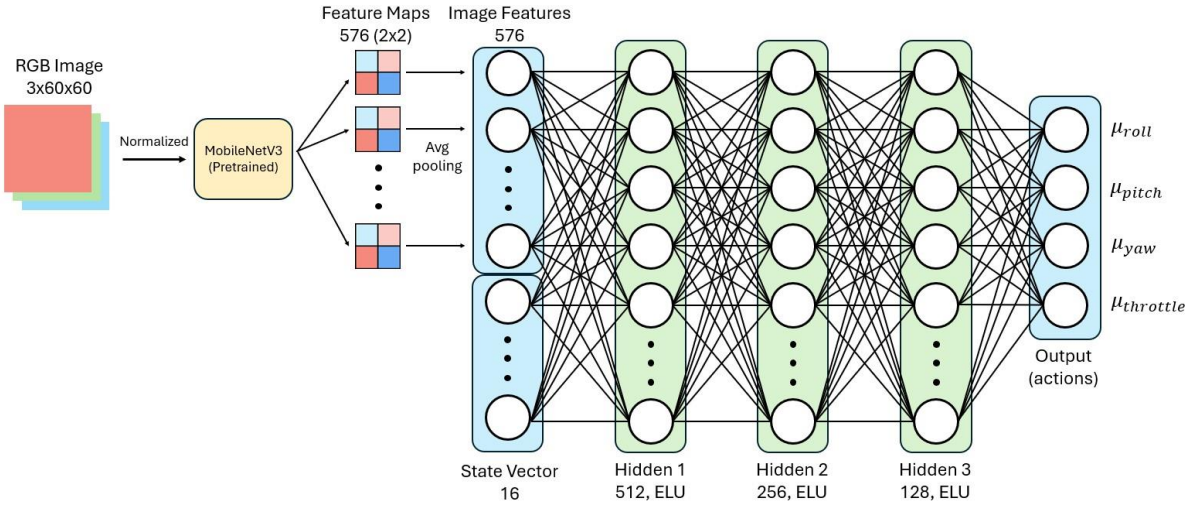

Figura 4: Red neuronal del actor para vuelo basado en visión

Función de valor. La función de valor, a diferencia de la política, es un escalar y no una distribución. Su valor se obtiene a partir de la red neuronal del crítico, que tiene una única salida, a diferencia del actor que tiene una por cada acción. La notación de la función de valor es la siguiente:

$$V_\phi(s_t) \in \mathbb{R},$$

donde s_t es el estado (vector observación) en t. La estructura de la red neuronal del crítico se muestra en la Figura 5 y Figura 6. Ésta también tiene 3 capas ocultas con 512, 256 y 128 neuronas.

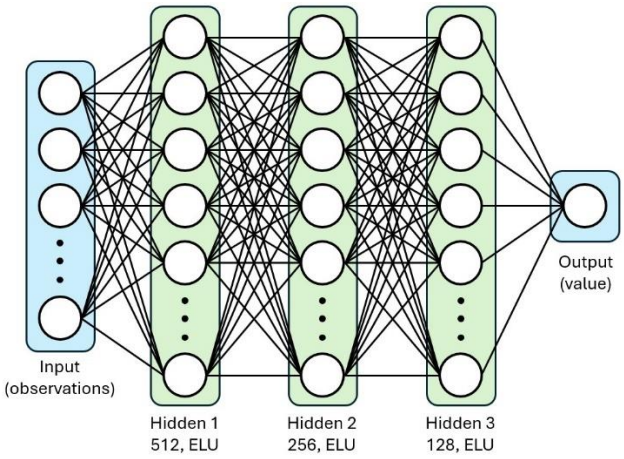

Figura 5: Red neuronal del crítico para vuelo basado en estados

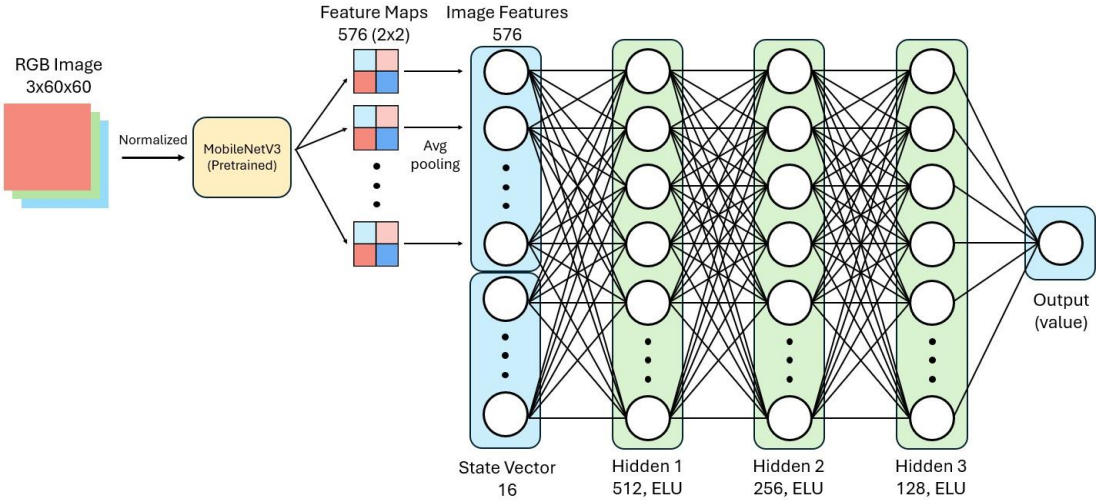

Figura 6: Red neuronal del crítico para vuelo basado en visión

La función valor estima el futuro premio total (descontado) en un estado dado considerando que se sigue la política del agente:

$$V_\phi(s_t) = \mathbb{E}_\pi \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t \right]$$

Cuando una acción a_t es tomada y se obtiene el verdadero premio r_t, una estimación más precisa del verdadero valor del estado anterior se puede calcular:

$$V_\phi\left(s_t \mid a_t\right) = r_t + \gamma\, \mathbb{E}_\pi \left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_{t+1} \right]$$

La función de ventaja (*advantage*), utilizada como señal de entrenamiento para ambas redes, se puede calcular de la siguiente manera:

$$A_t = V_\phi\left(s_t \mid a_t\right) - V_\phi(s_t)$$

Si A_t es positiva, significa que la acción tomada ha tenido un mejor efecto del esperado, reforzando la decisión tomada. Si, por el contrario, A_t es negativa, significa que la acción tomada ha tenido peor efecto del esperado y la decisión es penalizada.

3. RESULTADOS

Los objetivos de los experimentos son los siguientes:

1. Validar el modelo del cuadricóptero, demostrando que tanto el robot como el modelo aerodinámico funcionan como esperado.
2. Evaluar el controlador de vuelo tipo Betaflight paralelizable, asegurando que permite un vuelo estable.
3. Evaluar el entorno, confirmando que es adecuado para entrenar agentes de vuelo autónomo de drones.

Para el vuelo basado en estados, la política es entrenada en 4000 entornos paralelos, mientras que, en el vuelo basado en visión, tan solo se entrena con 400 entornos paralelos. Esto se debe a que la integración de cámaras en cada uno de los drones aumenta notablemente la carga computacional. La Figura 7 muestra la evolución del premio total medio por episodio a lo largo de una sesión en la que ambos agentes han tenido 30 millones de interacciones con el entorno. La Figura 8 muestra la evolución de la tasa de éxito a lo largo de las mismas sesiones.

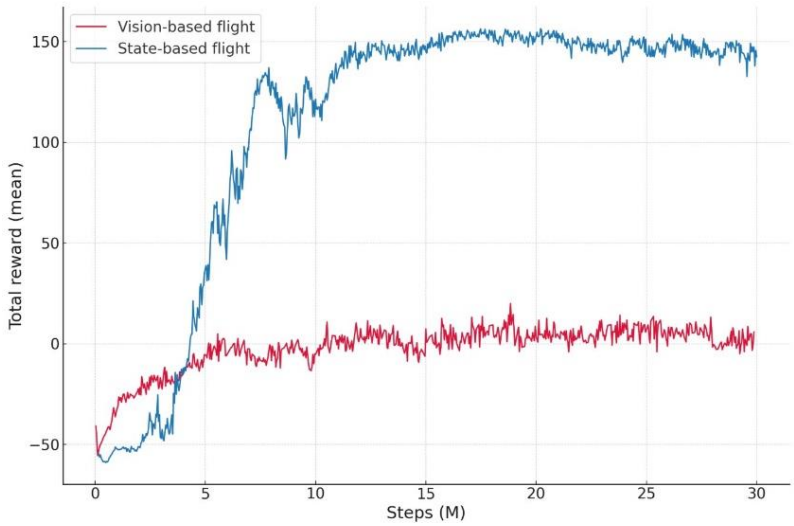

Figura 7: Mediana del premio total por episodio de ambos experimentos

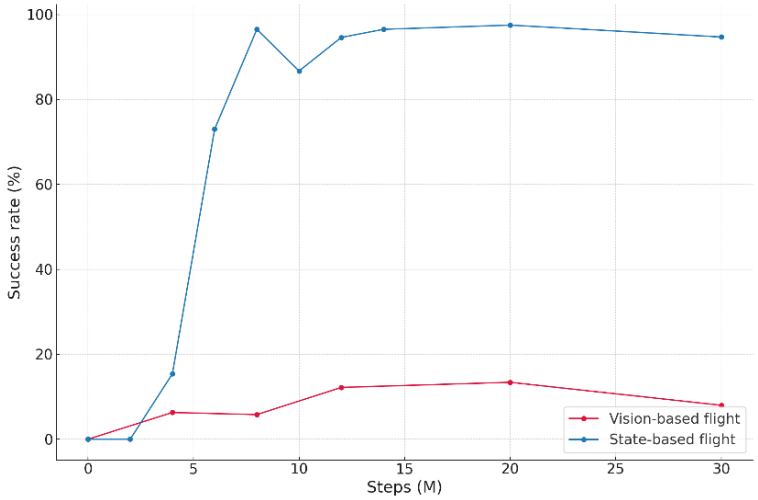

Figura 8: Tasa de éxito de ambos experimentos

La política basada en estados aprende exitosamente a atravesar el pasillo, convergiendo tras unas 15 millones de interacciones (unos 5 minutos en un ordenador de gama alta) y alcanzando una tasa de éxito de hasta el 98%. La Figura 9 muestra la trayectoria del mejor agente de la política basada en estados.

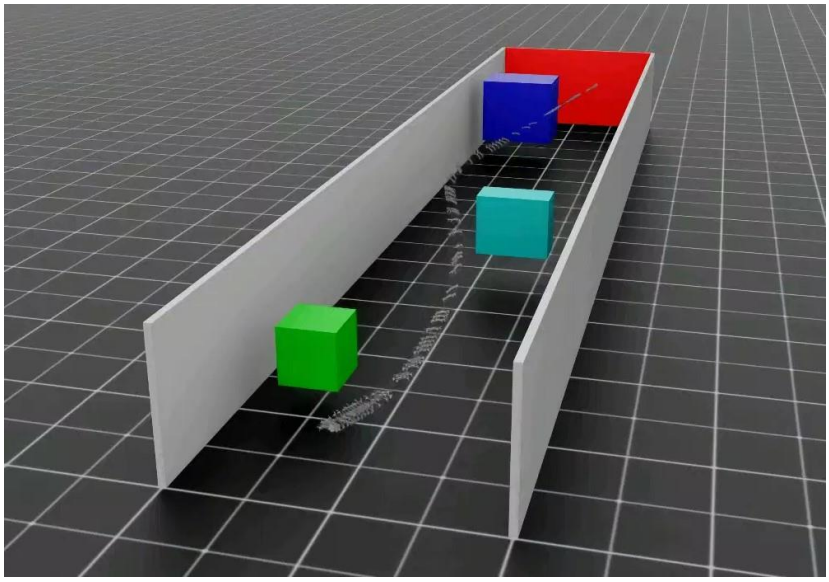

Figura 9: Trayectoria del mejor agente de la política de vuelo basada en estados. Techo eliminado para una mejor visualización.

Por otro lado, la política basada en visión tiene mucha más dificultad en aprender a atravesar el pasillo (como es esperado). La tarea de esta segunda es mucho más complicada: tiene que aprender a identificar no solo dónde está el dron, sino también dónde se encuentran los obstáculos. El agente apenas logra aprender a avanzar o esquivar de manera aleatoria, alcanzando el final del pasillo únicamente por suerte en los casos en que lo consigue.

Esto se puede deber a diversos factores, entre ellos la falta de una capa de memoria en la red neuronal que permita al agente retener información temporal, extractor de características de imagen inapropiado, el uso de una arquitectura simétrica en lugar de asimétrica, etc.

5. CONCLUSIONES

En conclusión, los resultados han demostrado que la metodología propuesta ha cumplido con éxito los tres objetivos planteados:

1. **Diseño e implementación de un dron cuadricóptero personalizado y su respectivo modelo aerodinámico** dentro del marco de Nvidia Isaac Lab. Esto proporciona una base para una simulación precisa, basada en física, de la dinámica de cuadricópteros.
2. **Implementación de un controlador de vuelo paralelizable basado en Betaflight** dentro de Isaac Lab. Esta solución permite un control estable a la vez que soporta entrenamiento paralelo a gran escala, superando las limitaciones de los enfoques SITL de Betaflight existentes, que se ejecutan de forma secuencial.
3. **Desarrollo de un entorno personalizado en Isaac Lab** que integra el dron cuadricóptero y el controlador basado en Betaflight en una tarea de navegación con obstáculos. En ella, el dron debe alcanzar un objetivo, lo que convierte este entorno en una plataforma adecuada para evaluar tanto políticas de refuerzo basadas en estado como basadas en visión.

En conjunto, estos desarrollos establecen un nuevo marco basado en Isaac Lab para el aprendizaje por refuerzo en control de drones mediante visión, proporcionando una base sólida para futuras investigaciones en vuelo autónomo ágil.

Aunque los resultados del experimento de la política basada en visión son peor de lo esperado, es importante reiterar que el objetivo de este proyecto no es el desarrollo de una arquitectura de vuelo autónomo, sino el desarrollo de herramientas que luego podrán ser usadas para ese mismo fin. Los resultados del experimento basado en estados prueban que el marco creado con este proyecto es una buena base para investigación sobre posibles futuras arquitecturas de control.

En caso de querer mejorar el resultado de vuelo con visión, algunas sugerencias son las siguientes:

- Implementación de una arquitectura de red neuronal más compleja, en la que se incluye al menos una capa de memoria como una GRU [10] o TCN [11]. Esto permitiría al dron tener mejor comprensión temporal para entender su movimiento, velocidad, etc.
- Cambio a una arquitectura de Actor-Critic asimétrica, donde el crítico tiene acceso a información privilegiada. Este tipo de arquitecturas han sido demostradas más eficientes en el pasado [12].
- Uso de un extractor de características personalizado pasado en capas de CNNs. Aunque codificadores previamente entrenados (como MobileNetV3-small) son

útiles, éstos son entrenados para clasificar, no identificar la ubicación exacta de un objeto en una imagen (como es crucial para tareas así).

- Añadir texturas a paredes y objetos. Esto no solo permitiría a red estimar su ubicación y movimiento de mejor manera, sino que también reduciría la brecha entre simulación y mundo real.

REFERENCIAS

[1] Sysnav. *Why GNSS and GPS Do Not Function Properly Indoors.* Blog post, published May 31, 2024. Available at: https://www.sysnav.fr/why-gnss-and-gps-do-not-function-properly-indoors/?lang=en (Accessed: August 26, 2025).

[2] Kaufmann, E., Bauersfeld, L., Loquercio, A., Müller, M., Koltun, V., & Scaramuzza, D. *Champion-level drone racing using deep reinforcement learning.* Nature, 620(7976), 982–987 (August 2023). doi:10.1038/s41586-023-06419-4. PMID: 37648758. PMCID: PMC10468397.

[3] Geles, I., Bauersfeld, L., Romero, A., Xing, J., & Scaramuzza, D. *Demonstrating Agile Flight from Pixels without State Estimation.* arXiv preprint arXiv:2406.12505 (2024). Available at: https://arxiv.org/abs/2406.12505 (Accessed: August 28, 2025).

[4] Lin, Y., Gao, F., Qin, T., Gao, W., Liu, T., Wu, W., Yang, Z., & Shen, S. *Autonomous aerial navigation using monocular visual-inertial fusion.* Journal of Field Robotics, 35, 23–51 (2018). Available at: https://api.semanticscholar.org/CorpusID:3670278.

[5] NVIDIA Corporation. *NVIDIA Isaac Lab.* 2025. Available at: https://developer.nvidia.com/isaac/lab (Accessed: August 22, 2025).

[6] Betaflight Community. *Betaflight.* 2025. Open-source flight controller firmware widely used for FPV quadcopters. Available at: https://betaflight.com/ (Accessed: August 22, 2025).

[7] Betaflight Contributors. *SITL (Software in the Loop) — Betaflight Documentation.* 2025. Available at: https://betaflight.com/docs/development/SITL (Accessed: August 26, 2025).

[8] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., & Adam, H. *Searching for MobileNetV3.* In: Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, South Korea, October 2019, pp. 1314–1324. IEEE.

[9] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. *Proximal Policy Optimization Algorithms.* arXiv preprint arXiv:1707.06347 (2017). Available at: https://arxiv.org/abs/1707.06347 (Accessed: August 28, 2025).

[10] Zhang, Y., Hu, Y., Song, Y., Zou, D., & Lin, W. *Learning vision-based agile flight via differentiable physics.* Nature Machine Intelligence, 7(6), 954–966 (June 2025). doi:10.1038/s42256-025-01048-0.

[11] Xing, J., Romero, A., Bauersfeld, L., & Scaramuzza, D. *Bootstrapping Reinforcement Learning with Imitation for Vision-Based Agile Flight.* arXiv preprint arXiv:2403.12203 (2024). Available at: https://arxiv.org/abs/2403.12203 (Accessed: August 28, 2025).

AGILE VISION-BASED FLIGHT FOR DRONES USING NEURAL NETWORKS

Author: Paramio Valdés, Pablo.

Director: Tordesillas Torres, Jesús.

Collaborating Entity: ICAI – Universidad Pontificia Comillas.

ABSTRACT OF THE PROJECT

1. INTRODUCTION

The project addresses the problem of the autonomous agile flight of vision-based quadcopter drones. Although this type of autonomy has already been achieved through technologies such as GPS, its use is unreliable in complex environments or at high speeds, where the signal is slow and imprecise [1]. As an alternative, flight based on vision with artificial intelligence emerges.

In this area, two main approaches can be distinguished:

1. **Hybrid neural network policies**, which divide the problem into two stages: the perception or explicit estimation of the state from images, and the control or decision-making based on that state [2].
2. **End-to-end neural network policies**, in which the agent learns to generate control signals directly from the onboard camera, mimicking the way an FPV pilot handles the drone [3].

This project focuses on the second variant, as it is more promising for agile flight, where explicit state estimation is difficult [4], and expensive sensors are avoided.

The main objective is the development of a simulation framework on top of Nvidia Isaac Lab [5] that will serve as a basis for future research in vision-based autonomous flight. To this end, three key components have been implemented:

1. **A quadcopter robot and its respective realistic aerodynamic model** within the framework of Isaac Lab.
2. **A parallelizable Betaflight-style [6] flight controller**, which ensures compatibility with real FPV drones.
3. **An obstacle-based training environment**, where both state-based and vision-based policies are tested with reinforcement learning (RL) techniques.

Rather than seeking a new state of the art in neural architectures, this work provides open tools and methodologies that facilitate future advances in vision-based agile autonomous flight.

2. METHODOLOGY

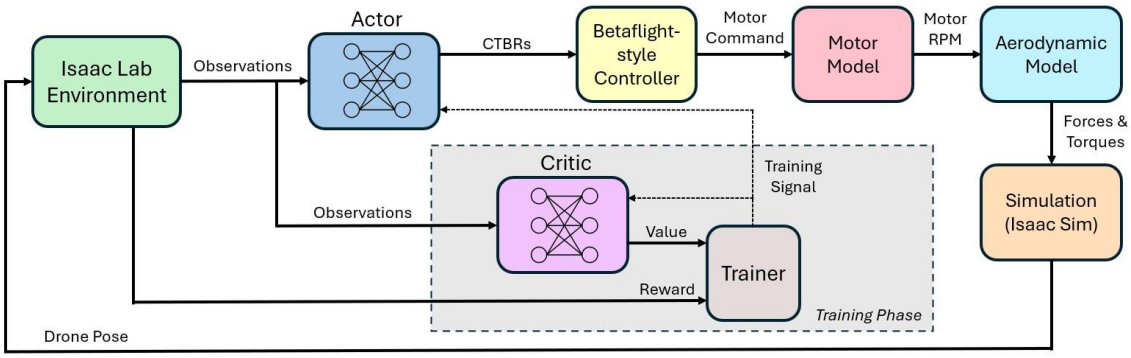

Figure 1: General schematic of the project data flow

The schematic shown in Figure 1 shows the data flow for the entire project. In it, it is shown how a symmetric Actor-Critic architecture is used, in which the actor learns to generate a collective thrust command and body rates (CTBRs) from the observations, and the critic learns to estimate the value for a given state and is used during agent training.

1. Quadcopter Robot and Aerodynamic Model

At Isaac Lab, quadcopter environments default to the *Articulation* class, which simply applies external torques and forces directly to the drone, rather than modeling how each individual motor generates yaw force and torque.

To solve this, the project introduces a new type of asset called *Multicopter*, which receives as input the angular velocities of each engine and calculates the resulting forces and aerodynamic moments, faithfully reproducing the behavior that the drone would have in reality. This part of the development was carried out with the help of Mario Gómez Andreu, a collaborator at the Comillas Pontifical University.

The aerodynamic model is based on the assumption of a quadratic ratio between the angular velocity of the motor and the force and momentum it generates:

$$f_i = k_f \cdot \omega_i^2$$

$$\tau_z = \sum_{i=1}^{N} k_m \cdot d_i \cdot \omega_i^2$$

where k_f is the coefficient of thrust, k_m is the coefficient of momentum and d_i is a variable that is worth 1 or -1 depending on the direction in which the motor i rotates.

All physical and aerodynamic parameters (number of rotors, directions of rotation, thrust and moment coefficients, inertia and rotor positions) are stored in a configuration class (*MulticopterCfg*). This allows the same simulation code to be reused in different multirotor designs by simply modifying the configuration files.

The implementation is spread over several scripts:

- *multicopter.py*: central simulation logic and aerodynamic model function.
- *multicopter_cfg.py*: aerodynamic and geometric parameters.
- *quadcopter.py*: Quadcopter-specific configurations, with the definition of *CRAZYFLIE_CFG_MULTICOPTER* as a substitute for the *Articulation*-based model.

This new multirotor model significantly improves the realism of the simulation and reduces the gap between simulation training and implementation in real drones.

2. Betaflight-Style Controller

Unlike what is usual in similar research projects, in which a custom flight controller is used, one of the goals of this project is to develop one that simulates Betaflight, the flight controller most commonly used in FPV drones. This allows the trained policy to be used in any drone with Betaflight installed, without needing a custom controller flashed onto it. Although there is already Betaflight SITL [7], which allows the controller to be implemented in simulations, it has the great limitation of not supporting parallel computing, essential in RL applications such as this one.

The developed controller uses the Pytorch library to perform the necessary control operations with tensors in a batching fashion. It is based on a PID with differential action on the output:

$$\boldsymbol{pid}_{\text{sum}}(t) = \boldsymbol{p}(t) + \boldsymbol{i}(t) + \boldsymbol{d}(t)$$

$$\boldsymbol{pid}_{\text{sum}} \in \mathbb{R}^{\text{N}\times 3}$$

where N is the number of simulated parallel environments (*num_envs*). To obtain the motor control signal (command), a mixer matrix is used:

$$\boldsymbol{M} = \begin{array}{c} \\ \text{FR} \\ \text{RR} \\ \text{RL} \\ \text{FL} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Roll} & \text{Pitch} & \text{Yaw} \end{array} \\ \begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & 1 & 1 \\ 1 & -1 & -1 \end{bmatrix} \end{array}$$

$$\boldsymbol{mix}(t) = \boldsymbol{pid}_{\text{sum}}(t) \cdot \boldsymbol{M}^{\top}$$

$$\boldsymbol{mix}(t) \in \mathbb{R}^{N\times 4}$$

Once the commands are in place, a model of the engine is applied using the following formula to obtain the speed of each rotor:

$$\boldsymbol{\omega}(t) = \min\left(\boldsymbol{motor\_command}(t)^{\odot\alpha} \odot \boldsymbol{\omega}_{\max},\ \boldsymbol{\omega}_{\max}\right)$$

where α is an experimental parameter, $\odot\,\alpha$ represents an element-by-element power and ω_{max} is the maximum speed of each rotor.

3. Isaac Lab Environment

The environment used to train the agent consists of a corridor 10 meters long, 2 meters wide and 1 meter high. The hallway contains 3 cube-shaped obstacles that randomly change position each time the drone crashes or makes it all the way, introducing *domain randomization*. Figure 2 shows an exterior image of the hallway with the roof removed for greater visibility. The goal of the drone is to traverse the hallway as quickly as possible without hitting any walls or obstacles.

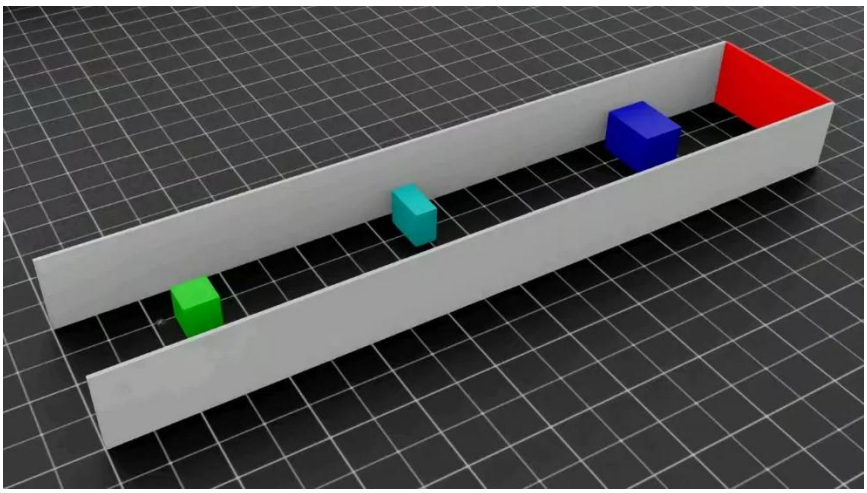

Figure 2: Image of the hallway used for agent training with the roof removed for a better view.

To evaluate both the environment and the aerodynamic model and the flight controller, two agent training experiments are conducted: one with observations of the current state of the simulation, and another based purely on vision, with the images from a camera mounted on the drone.

For the case of the state-based experiment, the observations are as follows:

- **Orientation** (roll, pitch and yaw):

$$\boldsymbol{\theta} = (\phi,\ \vartheta,\ \psi) \in \mathbb{R}^3$$

- **Angular velocity:**

$$\dot{\boldsymbol{\theta}} = (\dot{\phi},\, \dot{\vartheta},\, \dot{\psi}) \in \mathbb{R}^3$$

- **Linear Velocity:**

$$\boldsymbol{v_b} = (v_x, v_y, v_z) \in \mathbb{R}^3$$

- **Unit vector of projected gravity**, indicating orientation with respect to the global frame:

$$\boldsymbol{g_b} \in \mathbb{R}^3, \quad \|g_b\| = 1$$

- **Vector of distances to the limits of the corridor** (walls, ceiling and floor):

$$\boldsymbol{d}_{\text{right}} \in \mathbb{R}^3 \quad \boldsymbol{d}_{\text{left}} \in \mathbb{R}^3 \quad \boldsymbol{d}_{\text{ceil}} \in \mathbb{R}^3 \quad \boldsymbol{d}_{\text{ground}} \in \mathbb{R}^3$$

- **Vector of direction towards the target position** in relative coordinates**:**

$$\boldsymbol{d}_{\text{target}} \in \mathbb{R}^3, \quad \|\boldsymbol{d}_{\text{target}}\| = 1$$

- **Relative position of the next nearest obstacle** :

$$\boldsymbol{d}_{\text{obs}} \in \mathbb{R}^3$$

- **Prior action** executed by the agent, to provide temporal consistency:

$$\boldsymbol{a_{t-1}} \in \mathbb{R}^4$$

On the other hand, in the case of the **vision-based experiment**, all observations would be available in the actual implementation:

- 60x60 RGB image captured by the drone's front camera. This image is processed by a MobileNetV3-small [8], a pre-trained neural network that extracts the main features of the image. This generates 576 feature maps that are reduced to a vector of dimension 576 after *average pooling*:

$$\boldsymbol{f}_{\text{rgb}} \in \mathbb{R}^{576}$$

- **Orientation** (roll, pitch and yaw):

$$\boldsymbol{\theta} = (\phi,\, \vartheta,\, \psi) \in \mathbb{R}^3$$

- **Angular velocity:**

$$\dot{\boldsymbol{\theta}} = (\dot{\phi},\, \dot{\vartheta},\, \dot{\psi}) \in \mathbb{R}^3$$

- **Unit vector of projected gravity:**

$$\boldsymbol{g_b} \in \mathbb{R}^3, \quad \|g_b\| = 1$$

- **Vector of direction towards the target position** in relative coordinates:

$$\boldsymbol{d}_{\text{target}} \in \mathbb{R}^3, \quad \|\boldsymbol{d}_{\text{target}}\| = 1$$

- **Prior action** executed by the agent:

$$\boldsymbol{a_{t-1}} \in \mathbb{R}^4$$

Finally, in order to properly guide the agent's training and thus get the drone through the corridor successfully, a dense reward function is used. In each interaction of the control policy with the environment, the reward is calculated with the following function:

$$r_t = r_{\text{progress}} + r_{\text{end}} + r_{\text{collision}} + r_{\text{orientation}} + r_{\text{smooth}}$$

Where each part of the prize is calculated as follows:

- **Progress:**

$$r_{\text{progress}} = \Delta p \cdot \alpha_{\text{progress}}$$

  Where Δp represents the progress that has been made along the corridor in the last simulation step and $\alpha_{progress}$ is a positive scaling factor.

- **Arrival at the end:**

$$r_{\text{end}} = \mathbf{1}_{\text{end}} \cdot \alpha_{\text{end}} \left(1 - \frac{n_t}{N_{\max}}\right)$$

  Where $\mathbf{1}_{end}$ is an indicator that is worth 1 when the drone has reached the end and 0 otherwise, n_t is the number of steps taken by the simulation, N_{max} is the maximum number of steps per attempt and α_{end} is a positive scaling factor.

- **Collision penalty:**

$$r_{\text{collision}} = -\mathbf{1}_{\text{collision}} \cdot \alpha_{\text{collision}}$$

  Where $\mathbf{1}_{collision}$ is an indicator that is worth 1 when the drone has crashed and 0 if otherwise and $\alpha_{collision}$ is a positive scale factor.

- **Orientation:**

$$r_{\text{orientation}} = e^{(4 \cdot d_{\text{target},x} - 3)} \cdot \alpha_{\text{orientation}} \cdot \Delta t$$

  where $d_{target,x}$ is the x component of the direction-to-target vector, $\alpha_{orientation}$ is a positive scale factor and Δt is the simulation interval.

- **Smoothness of control:**

$$r_{\text{smooth}} = -\left\|\boldsymbol{a_t} - \boldsymbol{a_{t-1}}\right\|^2 \cdot \alpha_{\text{smooth}} \cdot \Delta t$$

Where a_t is the current action, a_{t-1} is the past action, α_{smooth} is a positive scaling factor, and Δt is the simulation interval.

The values of the scale factors are obtained empirically and offer a balance between speed, stability, smoothness and safety during flight:

$$\alpha_{\text{collision}} = 50.0, \quad \alpha_{\text{progress}} = 6.0, \quad \alpha_{\text{end}} = 80.0, \quad \alpha_{\text{orientation}} = 5.0, \quad \alpha_{\text{smooth}} = -5.0$$

4. Neural Network Architecture

A symmetric Actor-Critic algorithm is used, where the structure of both networks (actor and critic) is almost the same and only their output changes. Specifically, the agent is trained using *Proximal Policy Optimization* (PPO) [9].

Control policy. The policy is modeled as a normal distribution, parameterized by a mean and a logarithmic standard deviation.

$$\pi_\theta(a_t \mid s_t) = \mathcal{N}(\mu_\theta(s_t),\, \sigma^2),$$

where s_t is the state (observation vector) in t, a_t is the action (CTBRs) and μ_θ is obtained through the actor's neural network, formed by hidden layers of 512, 256, and 128 neurons and exponential linear activations (ELU). The actor-based architectures for state-based and vision-based policy are shown in Figure 3 and Figure 4 respectively. The logarithmic standard deviation, instead of being predicted by the neural network, is maintained as a separate trainable parameter.

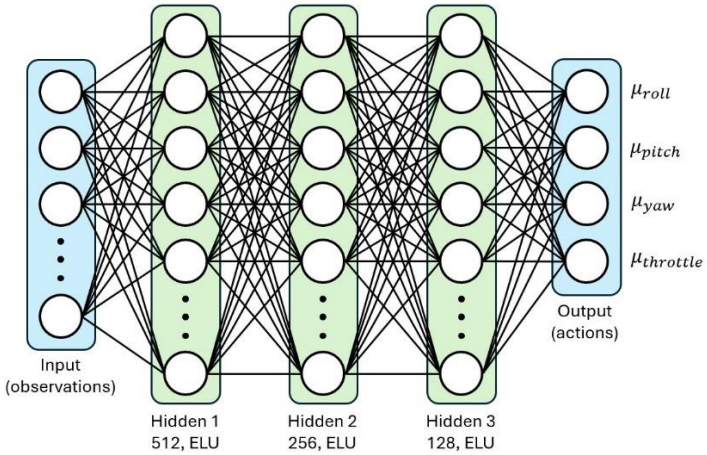

Figure 3: Actor neural network for state-based flight

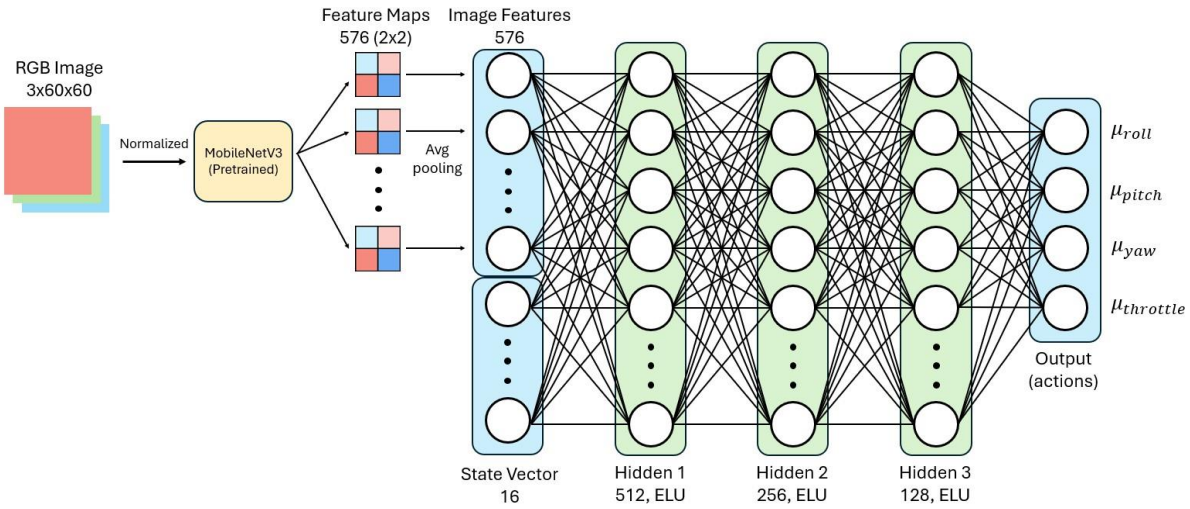

Figure 4: Actor neural network for vision-based flight

Value function. The value function, unlike the policy function, is a scalar and not a distribution. Its value is obtained from the neural network of the critic, which has a single output, unlike the actor who has one for each action. The notation of the value function is as follows:

$$V_\phi(s_t) \in \mathbb{R},$$

where s_t is the state (observation vector) in t. The structure of the neural network of the critic is shown in Figure 5 and Figure 6. It also has 3 hidden layers with 512, 256, and 128 neurons.

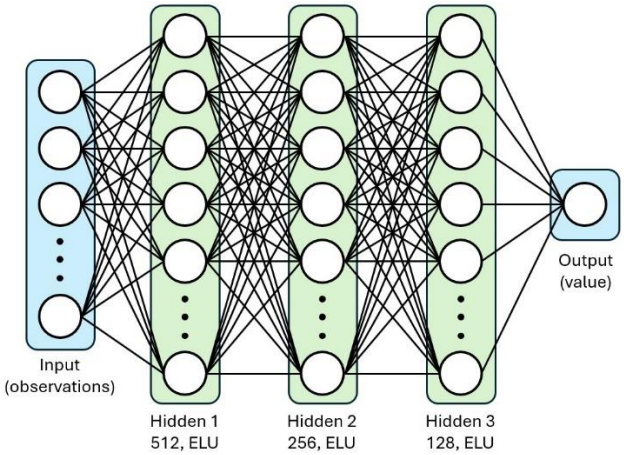

Figure 5: Critical neural network for state-based flight

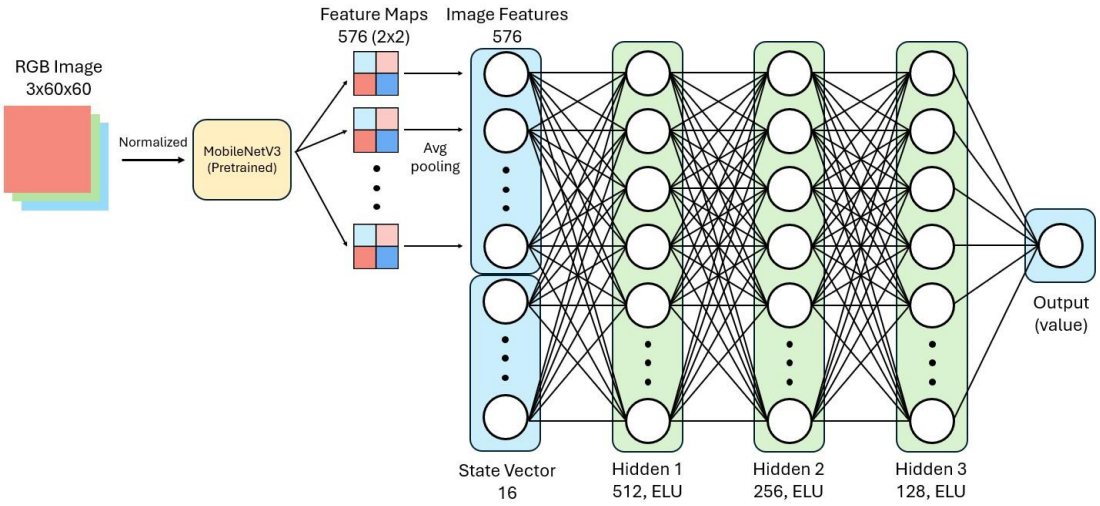

Figure 6: Critical neural network for vision-based flight

The value function estimates the total future reward (discounted) in a given state considering that the agent's policy is followed:

$$V_\phi(s_t) = \mathbb{E}_\pi \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t \right]$$

When an action a_t is taken and the true reward r_t is obtained, a more accurate estimate of the true value of the previous state can be calculated:

$$V_\phi\left(s_t \mid a_t\right) = r_t + \gamma\, \mathbb{E}_\pi \left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_{t+1} \right]$$

The advantage function, used as a training signal for both networks, can be calculated as follows:

$$A_t = V_\phi\left(s_t \mid a_t\right) - V_\phi(s_t)$$

If A_t it is positive, it means that the action taken has had a better effect than expected, reinforcing the decision made. If, on the other hand, A_t is negative, it means that the action taken has had a worse effect than expected and the decision is penalized.

3. RESULTS

The objectives of the experiments are to:

1. Validate the quadcopter model, demonstrating that both the robot and the aerodynamic model work as expected.
2. Evaluate the parallelizable Betaflight-style flight controller, ensuring that it allows a stable flight.
3. Assess the environment, confirming that it is suitable for training autonomous drone flight agents.

For state-based flight, the policy is trained in 4000 parallel environments, while in vision-based flight, it is only trained in 400 parallel environments. This is because the integration of cameras in each of the drones significantly increases the computational load. Figure 7 shows the evolution of the mean total reward per episode over a session in which both agents have had 30 million interactions with the environment. Figure 8 shows the evolution of the success rate over the same sessions.

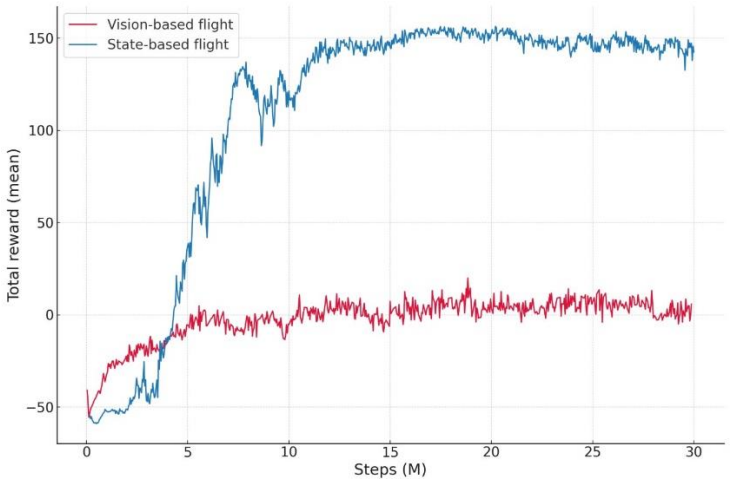

Figure 7: Median total award per episode of both experiments

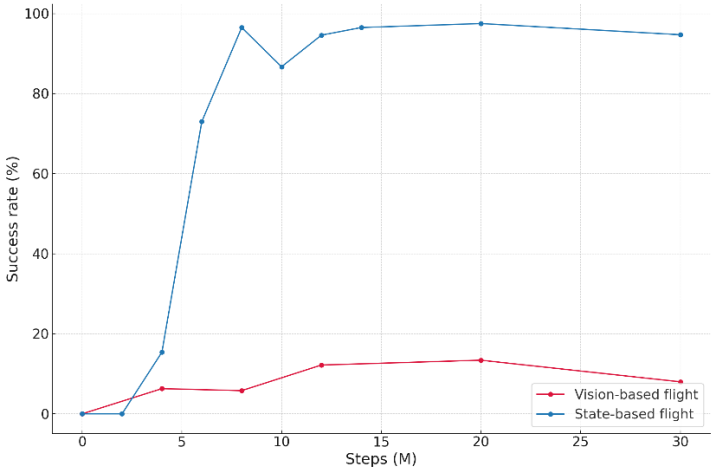

Figure 8: Success rate of both experiments

The state-based policy successfully learns to traverse the corridor, converging after some 15 million interactions (about 5 minutes on a high-end computer) and reaching a success rate of up to 98%. Figure 9 shows the trajectory of the best agent of the state-based policy.

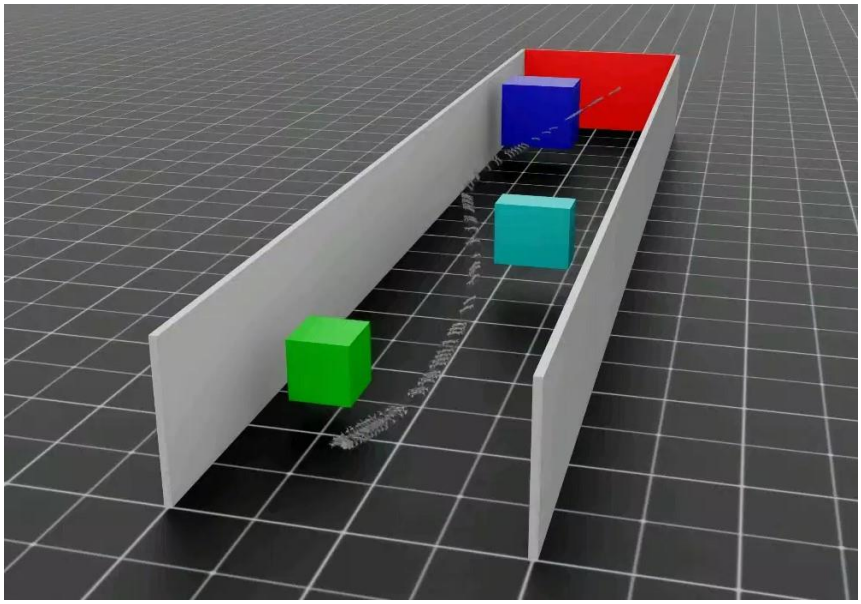

Figure 9: Trajectory of the best agent of state-based flight policy. Roof removed for better viewing.

On the other hand, the vision-based policy has much more difficulty learning to cross the corridor (as expected). The task of this second one is much more complicated: it has to learn to identify not only where the drone is, but also where the obstacles are and successfully navigate around them. The agent barely manages to learn how to advance and rarely dodge any obstacles, reaching the end of the corridor only by luck in cases where it succeeds.

This poor performance can be due to a number of factors, including the lack of a memory layer in the neural network that would allow the agent to retain temporary information, inappropriate image feature extraction, the use of a symmetric rather than asymmetric architecture, and so on.

5. CONCLUSIONS

In conclusion, the results have shown that the proposed methodology has successfully met the three objectives set:

1. **Design and implementation of a custom quadcopter drone and its respective aerodynamic model** based Nvidia's Isaac Lab framework. This provides a basis for an accurate, physics-based simulation of quadcopter dynamics.
2. **Implementation of a parallelizable flight controller based on Betaflight** within Isaac Lab. This solution enables stable control while supporting large-scale parallel training, overcoming the limitations of existing Betaflight SITL approaches, which run sequentially.
3. **Development of a custom environment at Isaac Lab** that integrates the quadcopter drone and Betaflight-based controller into an obstacle navigation task. In it, the drone must achieve a goal, which makes this environment a suitable platform for evaluating both state-based and vision-based reinforcement policies.

Together, these developments establish a new Isaac Lab-based framework for reinforcement learning in vision-based drone control, providing a solid foundation for future research in agile autonomous flight.

Although the results of the vision-based policy experiment are worse than expected, it is important to reiterate that the objective of this project is not the development of an autonomous flight architecture, but the development of tools that can then be used for that exact purpose. The results of the state-based experiment prove that the framework created with this project is a good basis for research on possible future control architectures.

If one were to improve the result of the vision-based policy, some suggestions are the following:

- Implementation of a more complex neural network architecture, including at least one memory layer such as a GRU [10] or TCN [11]. This would allow the drone to have a better temporal understanding to understand its movement, speed, etc.
- Shift to an asymmetric Actor-Critic architecture, where the critic has access to privileged information. This type of architecture has been shown to be more successful in the past [12].
- Use of a custom feature extractor based on CNNs. Although pre-trained encoders (such as MobileNetV3-small) are useful, they are trained to classify, not identify the exact location of an object in an image (as is crucial for such tasks).
- Add textures to walls and objects. This would not only allow the network to better estimate its location and movement, but it would also reduce the gap between simulation and the real world.

REFERENCES

[1] Sysnav. *Why GNSS and GPS Do Not Function Properly Indoors.* Blog post, published May 31, 2024. Available at: https://www.sysnav.fr/why-gnss-and-gps-do-not-function-properly-indoors/?lang=en (Accessed: August 26, 2025).

[2] Kaufmann, E., Bauersfeld, L., Loquercio, A., Müller, M., Koltun, V., & Scaramuzza, D. *Champion-level drone racing using deep reinforcement learning.* Nature, 620(7976), 982–987 (August 2023). doi:10.1038/s41586-023-06419-4. PMID: 37648758. PMCID: PMC10468397.

[3] Geles, I., Bauersfeld, L., Romero, A., Xing, J., & Scaramuzza, D. *Demonstrating Agile Flight from Pixels without State Estimation.* arXiv preprint arXiv:2406.12505 (2024). Available at: https://arxiv.org/abs/2406.12505 (Accessed: August 28, 2025).

[4] Lin, Y., Gao, F., Qin, T., Gao, W., Liu, T., Wu, W., Yang, Z., & Shen, S. *Autonomous aerial navigation using monocular visual-inertial fusion.* Journal of Field Robotics, 35, 23–51 (2018). Available at: https://api.semanticscholar.org/CorpusID:3670278.

[5] NVIDIA Corporation. *NVIDIA Isaac Lab.* 2025. Available at: https://developer.nvidia.com/isaac/lab (Accessed: August 22, 2025).

[6] Betaflight Community. *Betaflight.* 2025. Open-source flight controller firmware widely used for FPV quadcopters. Available at: https://betaflight.com/ (Accessed: August 22, 2025).

[7] Betaflight Contributors. *SITL (Software in the Loop) — Betaflight Documentation*. 2025. Available at: https://betaflight.com/docs/development/SITL (Accessed: August 26, 2025).

[8] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., & Adam, H. *Searching for MobileNetV3*. In: Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, South Korea, October 2019, pp. 1314–1324. IEEE.

[9] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. *Proximal Policy Optimization Algorithms*. arXiv preprint arXiv:1707.06347 (2017). Available at: https://arxiv.org/abs/1707.06347 (Accessed: August 28, 2025).

[10] Zhang, Y., Hu, Y., Song, Y., Zou, D., & Lin, W. *Learning vision-based agile flight via differentiable physics*. Nature Machine Intelligence, 7(6), 954–966 (June 2025). doi:10.1038/s42256-025-01048-0.

[11] Xing, J., Romero, A., Bauersfeld, L., & Scaramuzza, D. *Bootstrapping Reinforcement Learning with Imitation for Vision-Based Agile Flight*. arXiv preprint arXiv:2403.12203 (2024). Available at: https://arxiv.org/abs/2403.12203 (Accessed: August 28, 2025).

Contents

Chapter 1

Introduction and Project Approach

With the rapid evolution of robotics and artificial intelligence in recent years, a lot of work has been put into robot automation. More specifically, the field of drones has seen a lot of changes. These tiny but extremely agile robots have demonstrated capabilities that no other type of robot possesses. Their ability to freely move in the three-dimensional space allows them to reach previously unreachable places and transverse extremely complex environments that would otherwise be inaccessible with ground vehicles. As a consequence, there has been a lot of interest in these flying robots, with possible applications in areas including inspection, scouting or even rescue operations.

This three-dimensional freedom, however, comes at the cost of increased operational complexity of the drone. As a result, there has been a major effort to achieve the autonomous operation of these robots. Although GPS has been used to control drones in simple outdoors tasks, this technology becomes less useful in complex environments due to its high inaccuracy and slow updates [1]. For that reason, significant research has been done in the field of vision-based flight, which allows autonomous operation in complex and GPS-denied environments.

However, vision-based flight has turned out to be quite challenging, as computing proper controls based on pixels is very complex. As a way to combat this high-complexity problem, industry has turned towards artificial intelligence, more specifically machine learning. Although there have been multiple approaches to this problem, one of the most promising involves imitating how real FPV[1] pilots agilely fly these drones: directly from a video feed. If these pilots are able to maneuver drones through incredibly complex environments at an astonishingly fast speed just with an onboard video feed as input, why wouldn't an AI model be able to do the same?

This particular approach typically uses what is known as an end-to-end policy, which directly maps input sensor data to output controls. This differs from traditional modularized strategies, which break the problem into different stages such as perception, planning, and control. This decomposition of the problem can cause integration issues that lead to accumulation of errors across stages, as well as increased delays, making it harder to react to dynamic obstacles quickly.

[1] FPV: First Person View

As part of the effort to replace the pilot of these machines with artificial intelligence agents, this project aims to build upon existing and previously tested techniques for vision-based autonomous flight while introducing novel approaches. The objective is to integrate proven methods from prior research with new strategies to enhance the performance of vision-based low-cost platforms.

Chapter 2

State of the Art

Paper	Year	Sensor	Control Policy Input	Control Policy Output	Learning approach	Computation Location	Dynamics Simulator	Visual Engine	Deployment Environment
[2]	2021	Depth camera and VIO estimator	State estimation (velocity & attitude) based on VIO, depth frame and desired direction (based on position estimation)	Three possible trajectories described by 10 position points each	Imitation learning (DAgger)	Onboard (Nvidia Jetson TX2)	Flightmare with RotorS Gazebo plugin	Unity	General outdoors environments (varying). 3D mapping of the environment is needed before deployment.
[3]	2023	Monocular camera, VIO estimator	State estimation (position, velocity & attitude) based on VIO and PnP triangulation, relative pose of next gate and previous action.	Collective thrust and body rates	Reinforcement learning (symmetric PPO)	Onboard (Nvidia Jetson TX2)	Custom polynomial aerodynamic model	None	Specific gated circuit
[4]	2024	Monocular camera	Image abstraction of camera (current frame) & past 3 actions	Collective thrust and body rates	Reinforcement learning (asymmetric PPO)	Offboard	Custom based on quadratic thrust and torque propeller	Custom (simple image abstraction)	Specific gated circuit
[5]	2024	Monocular camera	Image abstraction of camera (last H frames)	Collective thrust and body rates	IL with RL fine tuning (asymmetric PPO)	Offboard	Custom based on BEM	Custom (simple image abstraction)	Specific gated circuit
[6]	2025	Monocular camera	Stochastic latent state (encoded-image) and recurrent state	Collective thrust and body rates	Model-based RL (DreamerV3)	Offboard	Flightmare with Agilicious dynamics	Habitat	Specific gated circuit
[7]	2024	Depth camera, IMU	Depth frame, attitude, (velocity) & target velocity	Desired thrust vector from which CTBR are calculated	Differentiable physics	Onboard ($21 arm-based computer)	Custom differentiable simulator with point-mass physics	Custom depth rendering engine	General outdoors and indoors environments (varying)
[8]	2024	Monocular camera, IMU	Optical flow observation, attitude & goal direction	Desired acceleration	Differentiable physics	Offboard	Custom differentiable simulator with point-mass physics	Custom ray-tracing based visual engine	General outdoors and indoors environments (varying)

Table 2.1: Comparison of agile vision-based flight papers with neural networks

Vision-based autonomous drone flight has been around for quite some time, with the first vision-based solution emerging over 15 years ago [9]. Since then, many similar and innovative solutions have arisen [2, 3, 10]. Many of these have relied on the decomposition of the problem into different stages like perception, planning, and control [11, 12, 13]. Recent research, however, seems to be turning towards end-to-end policies, which, with the use of deep neural networks, promise a higher robustness and simplicity compared to modular solutions [14].

This section provides a thorough analysis of the state of the art in end-to-end and near-end-to-end vision-based flight. The section will be divided into 8 subsections, each tackling a different aspect of the technology. Tab. 2.1 contains a comparison summarizing the main differences between some of the most significant articles in this research.

2.1 End-to-End vs. Hybrid Policies

Although not exactly end-to-end policies, hybrid end-to-end solutions that make use of a separate perception stage are very common in this area of research, as papers rarely use a fully end-to-end approach. Among these hybrid solutions, the most common is the addition of a perception module that utilizes either VIO[1] or SLAM[2] to obtain an explicit estimation of the drone's state.

VIO uses a combination of inertial and visual data to generate an estimation of the drone's state. This technique has proven to be useful in slow applications, where drift in IMU[3] measurements is less prominent. SLAM, on the other hand, is a technique in which the environment is actively mapped as it is being explored. This enables drift correction by the execution of loop closures (revisiting the same place on the map to eliminate the accumulated error).

In the context of agile flight, however, VIO becomes more problematic. Drift in IMU measurements, motion blur, and image noise are significantly greater at higher speeds, making accurate state estimations impossible without error correction [13]. SLAM loop closure is not possible, as revisiting the same place on the map would prevent swift flight. The high processing delay in SLAM would also complicate proper control. In [3], a PnP[4] algorithm is used to triangulate the drone's position based on the detection of a specific gate with a known position in the circuit, eliminating the accumulated error of VIO and correcting drift. However, this is only possible because the distribution of the circuit is already known and would not work in unknown environments. In addition, due to the consistent and extremely low latencies needed in VIO, these strategies require powerful and specialized hardware to be able to do the sensing and computing onboard [4], severely limiting their range and scalability for industrial applications.

On the other hand, as an alternative to the use of hybrid techniques that make use of explicit state estimation, true end-to-end policies (from pixels to CTBR[5]) have recently gained popularity [4, 5, 6] [7, 8][6]. These policies imitate the way FPV pilots fly: they take the pixels in a frame as input and directly output collective thrust and body rates. As one could probably imagine, these policies take a lot longer to train than those that use explicit state estimation, since mapping controls from pixels is much more challenging than from position estimates. Many solutions to this sample inefficiency have been found to help with this problem; these will be discussed later.

Although not focused on agile maneuvers, end-to-end policies have been applied to vision-based flight in earlier work. One of the first examples is [15], where the authors introduced CAD2RL. In this approach, the control policy takes as input a single RGB image from a monocular camera and outputs velocity commands directly, without relying on explicit state estimation or a separate perception stage. To enable transfer from simulation to the real world, domain ran-

[1] VIO: Visual-Inertial Odometry
[2] SLAM: Simultaneous Localization And Mapping
[3] IMU: Inertial Measurement Unit
[4] PnP: Perspective-n-Point
[5] CTBR: Collective Thrust and Body Rates
[6] Although doing state estimation of the drone's attitude, these approaches will also be considered end-to-end because of the simplicity of their estimations (direct integration of IMU sensor data).

domization was applied during training by varying textures, lighting, and scene layouts in CAD models. Despite being trained only on synthetic data, the policy successfully generalized to real indoor hallways. While not demonstrating agile flight, this work laid important groundwork for later research on end-to-end vision-based agile control.

However, true end-to-end policies can lead to much higher robustness compared to solutions with explicit state estimation. They also allow computation to be done offboard, serving as a direct replacement for a human pilot.

2.2 Sensors

Monocular Cameras

The most widely used sensors and the first that come to mind in vision-based flight are monocular cameras (regular cameras) [3, 4, 5, 6, 7, 8]. These are the same cameras that professional FPV pilots use to fly drones. They are quite inexpensive compared to other options, dramatically lowering the price of the platform. They can capture a lot of detail in the scene and can adjust to different brightness levels when needed. The main disadvantage of using monocular cameras is the difficulty the policy may have in identifying different objects in a frame, as patterns on certain surfaces can confuse it. Policies that directly use monocular cameras as inputs need to be trained in photorealistic simulators in order to reduce the sim2real gap, which increases the computational cost.

Depth Cameras

Another commonly used sensor is depth cameras [2, 7], with Intel RealSense stereo cameras being the norm. Imitating human vision, these use two monocular cameras to triangulate the position of pixels, providing a depth map. Providing the depth of each pixel not only allows the policy to focus on the closest objects but also allows it to identify the different objects in the scene more easily, as textures are eliminated. The use of depth maps instead of RGB images means that the policy can be trained without photorealistic simulators, greatly reducing computational cost in training. Although Intel RealSense cameras also output RGB images, generally only the depth map is used as input. The biggest drawback in depth cameras is price, with Intel RealSense depth cameras priced around $300. In addition, depth cameras are typically limited to detecting distances within a range of 10-20m.

Inertial Measurement Units

Although both monocular and depth cameras can be used independently, they are often paired with an IMU [2, 3, 7, 8], which can provide the policy with the acceleration and angular velocity

of the drone (which can be integrated to obtain velocity and attitude estimates). In the cases where an IMU is not used [3, 4, 5], the policy learns to estimate its speed based on the visual change between frames, the same way an FPV pilot would do it. However, using the IMU data when available seems to be the appropriate approach, especially considering that most commercial flight controllers already have an integrated IMU. The drift in attitude and velocity estimations, although present, is much lower than in position estimation with VIO.

VIO Estimators

Another sensor that may sometimes be used for state estimation is a VIO estimator. Although VIO estimation can be done in custom-built setups (one or two monocular cameras and IMU data), a common approach is to use prepackaged systems. In [2, 3], a RealSense T265 (now discontinued) is used to directly input the state estimation into the policy. This specific sensor is a stereo VIO estimator that uses two cameras to estimate the state of the drone. Though useful, as mentioned before, VIO solutions tend to be less reliable in situations with fast motion and accelerations. The introduction of this sensor also significantly increases the price of the platform, negatively impacting its scalability.

Other Technologies

Although other technologies, such as LiDAR, have been used in similar work [16], they are not relevant to this research, which focuses solely on vision-based flight. Although highly effective for environment perception, these LiDAR sensors are expensive and heavy, reducing the deployability of the platform.

2.3 Control Policy Input

The control policy input refers to the information that the agent neural network uses to output control commands. The most crucial piece of information is the camera's current frame. In the case of platforms that make use of depth cameras [2, 7], the usual practice is to input either the raw depth map or a downscaled version of it. This direct approach is possible due to the simplicity of a depth map, which allows for a much higher sample efficiency compared to applications with regular cameras.

On the other hand, in applications where regular cameras are used, many attempts have been made to simplify the observations before inputting them to the agent. In [4, 5], where a quadcopter flies through a gated racing circuit, a pretrained gate detector is used to generate a simple image abstraction of the gate. This simplified image, obtaining either the inner edges or the corners of the gate, is fed into the control network, which outputs the controls. The main problem with this approach is that an abstraction like that is not generalizable for real-world environments.

A very different but interesting approach is taken in [8], where the authors use optical flow obtained from the video feed as input for the control policy. Optical flow represents the motion of objects in the scene. This has proven to be extremely useful, as objects closest to the camera are highlighted, as they appear to move faster than the background. This provides an image that not only eliminates texture but also labels objects based on their distance to the drone, much like a depth camera, but with a longer range, enabling the policy to plan in advance. The way that optical flow eliminates texture is a key factor in its strong generalization capability. However, the main issue with optical flow is that, just like event cameras, it fails to capture anything if the drone is stationary.

In [6], where a model-based method is employed, the frame is encoded into a vector, which is then used as input for the policy. The encoder is trained to downscale the image into a vector that contains the relevant information of the image (stochastic latent state). More about this will be analyzed in the learning approach section.

Using the raw image as input presents multiple challenges. As mentioned earlier, textures on objects make it harder for the policy to recognize them and limit its ability to generalize well. It also involves training in a photorealistic simulator, which leads to higher computational costs and a longer training time. Due to these challenges, this is rarely practiced. However, inputting the raw image allows the policy access to all of the information available, potentially enabling more complex and refined solutions.

As mentioned before, in policies where an explicit state estimation is calculated, this state information is used as an input for the control policy. In [2, 3], velocity and attitude estimations from the pre-packaged VIO estimator (RealSense T265) are used as inputs. However, in the latter, a position estimate obtained using a Kalman filter, which fuses the VIO-estimated position with the position estimated from a PnP problem, is also used as an input.

In platforms where no explicit state estimation is calculated [4, 5, 6], additional information is provided to the policy for it to estimate the drone's state implicitly. In [4], the past three output actions are included as inputs to provide the policy with a sense of continuity. This is also done with the intention of getting smoother control outputs. In [5], the past H (tunable parameter) frames are used as inputs of a TCN[7], allowing the policy to estimate its velocity based on how fast objects move across the field of view, similar to how an FPV pilot perceives motion. In [6], where a model-based approach is used, this past state information is captured in the recurrent state, which, along with the stochastic latent state (encoded image), serves as the input to the policy. In [7, 8], although an attitude estimation based solely on IMU sensor data is used as input, since the velocity is not, the policy learns to estimate it internally using a GRU[8], which stores past information in a hidden state.

Finally, in papers where training is not task-specific [2, 7, 8], a goal direction or velocity is also used as input for the policy to follow. In [2], a global trajectory is obtained from a simulated privileged expert with access to a point cloud of the deployment environment. Once the global trajectory is obtained, it is used in deployment to generate desired normalized directions from

[7]TCN: Temporal Convolutional Network
[8]GRU: Gated Recurrent Unit

the drone's estimated position to a position in the global trajectory 1 second into the future. This desired direction is used as input to the drone. In [7, 8], a velocity target vector is used as input to the policy. The direction of this vector is based on the difference between the starting position and the final goal position, while the magnitude of the vector is determined by a predefined maximum speed.

2.4 Control Policy Output

There are multiple outputs that the control policy can output; however, according to [17], the drone's agility is maximized when collective thrust and body rates (CTBR) are used. These are the same outputs that a regular pilot would provide through his transmitter. These are commonly used in many papers [3, 4, 5, 6].

In [7, 8], due to the use of a simple model that allows for differentiable physics simulations, which will be discussed later, the policy outputs a desired thrust or acceleration vector. These are then used to compute the desired CTBR to achieve the desired attitude for that specific thrust vector.

A particular approach is taken in [2], where a mix of deep learning and optimal control is used to solve the problem. In this method, the agent outputs three collision-free trajectories in the form of 10 waypoints. These trajectories are then represented as a high-order polynomial. The trajectory requiring the least control effort is chosen to favor smoothness. A model predictive controller then solves an optimization problem to generate drone control inputs to closely follow that trajectory.

2.5 Learning Approach

Model-Free Reinforcement Learning

The first learning approach that will be analyzed is reinforcement learning using Actor-Critic, more specifically PPO[9] [18]. This method combines policy-based (actor) and value-based (critic) approaches. The use of two separate networks enables the generation of outputs in the continuous space with an improvement in stability during training. For each control, the actor outputs a distribution (mean and standard deviation). The critic generates the expected value for a given state. Both networks are trained simultaneously: the critic learns to predict state values based on environment rewards while the actor is trained to output the best possible action based on the critic's value estimation. This method can be implemented either symmetrically [3], where both networks have access to the same information during training, or asymmetrically [4, 5], where the critic has access to privileged information like the drone's state. This is possible

[9]PPO: Proximal Policy Optimization

since the critic is not actually used during deployment, but rather as a tool to train the actor in a more stable and optimal manner than with direct rewards. An asymmetrical architecture seems to have a better sample efficiency compared to a symmetrical one [4].

PPO has also been successfully demonstrated in non-vision-based applications. For instance, in [19], a state-based PPO approach is taken to train a quadcopter to execute highly dynamic aerodynamic maneuvers using the current state of the drone and waypoints as inputs. Although this work is not vision-based, it highlights the flexibility of the Actor–Critic approach, showing that the same algorithm can be applied both to raw visual inputs and to simpler state-based representations.

Model-Based Reinforcement Learning

Though rarely used in this context, model-based RL is a possibility for vision-based flight. In [6], DreamerV3 is used to achieve this. In this method, a world model is typically trained to predict future states from a given state-action pair, simulating the real environment. In this particular case, however, it is used to simulate the dynamics of a simulated environment, rather than the real world itself. Once the world model is trained, it enables the agent to learn directly from imagined rollouts, allowing policy optimization without direct interaction with the environment. This approach enables high sample efficiency once the world model is trained, as no actual simulations need to be performed to train the control policy. However, because of the high-dimensional visual input and complex dynamics, this sample efficiency is offset by the vast amount of simulation data needed to train the world model itself, effectively negating the benefits.

Imitation Learning

Another commonly used approach is imitation learning (IL) [20]. This technique is based on training a policy from expert demonstrations. Due to the high volume of data needed to train vision-based policies, obtaining human-labeled data is unfeasible. To solve this, labeled data is obtained from an expert policy trained with RL with access to privileged information like its state or a 3D map of the environment.

The most basic version of IL is Behavior Cloning (BC) [21]. This involves training based on a loss function that compares the action taken by the student policy with the action taken by the expert policy. Although highly sample-efficient compared to RL, BC has some major disadvantages. Firstly, the student policy lacks robustness due to limited exploration, as it is trained only on states across the optimal trajectory. This leads to compounding errors when it inevitably deviates from the optimal path and finds itself in unseen states, causing what is known as covariate shift. Secondly, when doing BC, the student policy's performance is limited to that of the teacher, preventing further improvement over the demonstrations provided.

As an improvement over plain BC, many papers use a technique called Dataset Aggregation

(DAgger) [2, 5]. In this technique, after each iteration of BC, the student policy is executed to explore new states. The expert policy then labels these newly explored states by indicating the correct action to take from each. These new state-action pairs are added to the original dataset. BC is then done again with the increased dataset. This process is repeated multiple times, progressively expanding the dataset with an increasing number of student-explored states in each iteration. This process reduces covariate shift and increases robustness. It also allows the policy to surpass the expert's performance, providing a solution that retains the sample efficiency of IL while achieving the effectiveness of RL.

In [5], robustness and generalization are further improved by fine-tuning the policy with RL after training with DAgger. In addition, during this fine-tuning step, adaptive learning rates are used, adjusting based on policy performance. This helps prevent catastrophic forgetting and stabilizes training.

Differentiable Physics

In [7, 8], a very particular approach is taken for training. Unlike in many other works that rely on high-fidelity simulators with complex dynamics, the authors use a simulator with a simplified point-mass dynamics model. This enables the computation of exact gradients of physics-based loss functions with respect to control policy parameters by directly differentiating through the simulation dynamics, enabling efficient optimization via backpropagation. In other words, a loss function (eg. distance to an obstacle) can be directly traced to each weight and bias in the control policy.

In model-free Actor-Critic methods where the simulation is not differentiable, an objective is set through rewards. A critic is then used to estimate the value from a given state (expected future rewards). The action taken by the actor is then either reinforced or penalized based on whether it was more beneficial or more detrimental than expected (advantage). The standard actor policy loss function for step k is the following:

$$J_{\substack{\text{actor} \\ \text{A-C}}} = \mathbb{E}_k \left[\log \pi_\theta(u_k \mid x_k) A_k \right]$$

Where A_k represents the advantage and $\pi_\theta(u_k \mid x_k)$ represents the probability of taking action u_k at state x_k under the policy parameterized by θ. The use of differentiable physics, however, eliminates the need for a critic, since the benefit of taking an action (A_k in Actor-Critic) can now be directly computed via backpropagation. The actor policy loss function for differentiable physics is the following:

$$J_{\substack{\text{actor} \\ \text{DP}}} = \mathbb{E}_k \left[\log \pi_\theta(u_k \mid x_k) B_k \right]$$

$$B_k = -\frac{\partial L_{\text{physics}}}{\partial u_k}$$

Where B_k is a substitute for A_k and represents the benefit of taking action u_k at state x_k. It is computed directly by differentiating the physics-based loss function.

In summary, due to the access to true gradients, the use of differentiable physics is more

efficient than model-free RL methods, which use high-variance policy gradient estimates. In addition, it simplifies the system by relying on a single network, as the critic is no longer necessary.

2.6 Computation Location

This section analyzes where the computation for the autonomous control of the drone takes place. While some approaches execute the control policy onboard the quadcopter [2, 3, 7], others transmit sensor data to a ground station, where the policy runs and sends the control outputs back to the drone.

Although onboard computation results in a reduced delay in control, it typically requires the use of a high-performance computer equipped with a GPU [2, 3]. This not only raises prices (with Nvidia Jetson computers priced at a few hundred US dollars) but also adds weight and power consumption of the platform, jeopardizing its scalability. However, this is sometimes necessary when using techniques such as VIO, where fusion of inertial and visual data requires extremely low latencies [4]. In other cases, the control policy architecture is greatly optimized and simplified to lower computational costs during deployment, enabling onboard computation on a simpler and more cost-effective computer [7].

Whenever possible, the use of offboard computation seems to be advantageous, as it reduces the need for high-performance computers and provides access to virtually unlimited computational power in a ground station. The use of offboard computing also mimics the setup of a traditional drone pilot, where the video feed is transmitted to the pilot's goggles, and the pilot sends control commands back to the drone. This is extremely beneficial, as it provides a one-on-one substitution to the pilot without any hardware changes to the rest of the system. Although still presenting the disadvantage of an additional delay in transmission, previous work using this approach has not found major issues associated with it [4, 5, 6, 8].

2.7 Simulator

Simulator	Rendering	Dynamics	Sensor Suite	RL API	Photorealistic	Betaflight SITL
RotorS [22]	OpenGL	Gazebo-based	IMU, RGB, Depth, LiDAR	No	No	No
FlightGoggles [23]	Unity	Flexible	IMU, RGB	No	Yes	Yes (unofficially)
AirSim [24]	Unreal Engine	PhysX	IMU, RGB, Depth, Segmentation	Yes	Yes	No
Flightmare [25]	Unity	Flexible	IMU, RGB, Depth, Segmentation	Yes	Yes	No
Gym-PyBullet-Drones [26]	OpenGL	PyBullet	IMU, RGB, Depth, LiDAR	Yes	No	Yes
OmniDrones [27] (Isaac Sim)	Omniverse RTX	PhysX	IMU, RGB, Depth, LiDAR	Yes	Yes	No
VisFly [28]	Habitat-Sim	Differentiable + Lightweight	IMU, RGB	Yes	Yes	No

Table 2.2: Comparison of different drone simulators and their features.

In order to be able to train these policies, a drone simulator must be used. Although the use of custom simulators is common [3, 4, 5, 7, 8], there are several simulator frameworks available

for use. Tab. 2.2 includes some of the most popular drone simulators.

Among the options in Tab. 2.2, only AirSim, Flightmare, Gym-PyBullet-Drones, and OmniDrones have an RL API. Having an RL API integrated in the simulator has multiple benefits, such as seamless interaction with RL frameworks and parallel rollouts for faster training. Gym-PyBullet-Drones follows OpenAI Gym's API [29], making it compatible with Gym-based RL frameworks like Stable Baselines3. AirSim and Flightmare provide a Gym-compatible wrapper that allows it to be used like an OpenAI Gym environment.

OmniDrones, on the other hand, is built on Nvidia's Isaac Lab [30], which has its own interface that also integrates with external RL frameworks, and promises great performance, as Isaac Sim is a powerful simulator that offers vast RL support through Isaac Lab. However, OmniDrones has lost support and is now outdated, making it incompatible with Isaac Lab and essentially unusable.

Recent work has been done to build an RL framework for high-speed drone racing on top of Isaac Lab. The Isaac Drone Racer [31] project is specifically tailored to racing scenarios, featuring accurate physics modeling, onboard sensor simulation, and a track generator to create custom race circuits. It provides a flexible platform for training reinforcement learning policies in drone racing tasks.

Also recently, VisFly [28] has been introduced as a lightweight, Habitat-Sim–based [32] simulator designed specifically for training vision-based flight policies. Unlike Unity or Unreal-based simulators, it achieves extremely high frame rates (over 10,000 FPS with 64×64 images) while supporting differentiable physics and Gym integration. VisFly uses large 3D scene datasets such as Replica [33] and Matterport3D [34], which makes it possible to train policies quickly while still exposing them to a variety of realistic environments. Since the simulator is built on simplified physics, it runs very efficiently, though this also means it is not as physically accurate as heavier frameworks like Isaac Lab.

Another factor to consider is that although all the simulators presented in Tab. 2.2 have rendering engines, not all of them offer photorealistic simulations. The use of photorealistic simulations is needed when training an RGB vision-based model in order to achieve a good sim2real transfer. Finally, some simulators like Gym-PyBullet-Drones [26] offer some extra features like integration with Betaflight System-in-the-Loop (SITL) [35], which enables running the Betaflight flight controller (which is the most widely used in deployment) as part of the simulation. However, a major drawback in Betaflight SITL is its lack of support for parallel computation. In the cases in which Betaflight SITL is not available, many simulators still offer the use of either an integrated or a custom controller.

2.8 Deployment Environment

Although all of the papers contained in Tab. 2.1 have the general objective of vision-based flight, the types of environments in which they are meant to be deployed are very different. While

some policies are trained to be deployed in a specific environment, others are able to generalize to fly in unseen environments.

In the case of [3, 4, 5, 6], the deployment environment is a gated racing circuit, similar to those used by FPV racing pilots. The training for these policies is specific to each circuit, requiring precisely mapping the environment and training the policy until it converges before deploying. Although incredibly high speeds and agility are achieved, these policies learn a track by “visual memory" instead of actual complex decision making. Slight perturbations in the circuit’s layout are enough to cause the policy to fail. In [4], a 40cm movement of one of the gates is enough to drop the success rates to under 20%. These works also depend on the presence of visually prominent gates in the deployment circuit, which is not generalizable to real-world environments.

In [2], the policy is trained to be deployed in unseen environments. However, it needs access to an ideal trajectory to follow. To obtain this trajectory, the environment must be mapped into a 3D point cloud, which a privileged expert policy uses to determine the ideal collision-free trajectory called the global trajectory. This means that although the policy has the ability to fly in environments it has never seen itself, the environment still needs to be mapped before deployment.

On the other hand, the policies in [7, 8] are trained on changing environments with randomized obstacles, allowing them to generalize to more than a specific scenario. This type of training is much more applicable to solving real-world problems.

2.9 Motivation

As mentioned before, due to their ability to traverse extremely complex environments and the agility they possess in doing so, the range of applications in which these flying robots can be used to solve real-world problems is limitless. However, their autonomous use for agile flight is still a work in progress, especially in the case of GPS-denied applications.

While some platforms rely on the use of complex sensors like LiDAR, these are often heavy and expensive, reducing their deployability and scalability for real-world applications. As an alternative, vision-based flight promises to solve these issues by offering a cheap and lightweight solution that imitates how FPV drone pilots fly.

Although significant progress has been made in this area and agile vision-based flight through certain unseen environments has been demonstrated [7, 8], considerable effort is still needed to perfect this, as there is much to improve before achieving the smoothness with which FPV pilots fly. These solutions also rely on the use of either a depth camera [7], which can be expensive, or optical flow [8], which depends on the movement of the drone and shows nothing when the quadrotor is simply hovering. A fully end-to-end vision-based policy is yet to achieve agile flight in unseen environments.

Chapter 3

Developed Model

3.1 Project Objectives

The main goal of this project is to create tools and explore new techniques that can support further research on this matter, rather than focusing on developing a state-of-the-art neural network architecture.

On that note, the specific problems this project looks to tackle are the following.

1. **Design and implementation of a custom quadcopter robot and its respective aerodynamic model** within Nvidia's Isaac Lab framework, built on Isaac Sim. This looks to solve the lack of an up-to-date and openly available drone simulator powerful enough to reproduce real-world physics.
2. **Implementation of a parallelizable Betaflight-based drone controller** within Isaac Lab. Training agents on a Betaflight-like controller enables deployment on standard FPV drones without requiring uncommon or custom firmware. Unlike existing solutions such as Betaflight SITL, which support only serial computation, this approach allows for high-speed simulation across multiple environments.
3. **Development of a custom Isaac Lab environment** integrating a quadcopter robot and a Betaflight-based controller in a scene where the drone has to maneuver between obstacles to reach a desired goal. The environment will first be evaluated using a state-based approach, where the agent receives ground-truth data such as the relative position of obstacles. It will then be evaluated in a vision-based setup, combining the drone's onboard camera feed with sensor data typically available on FPV drones, including gyroscope readings and altitude.

The completion of these three objectives will result in a new Isaac Lab–based framework for vision-based drone flight, providing a solid foundation that can be readily extended and used in future research.

3.2 Algorithms and Implementation

3.2.1 General Data Flow of the Project

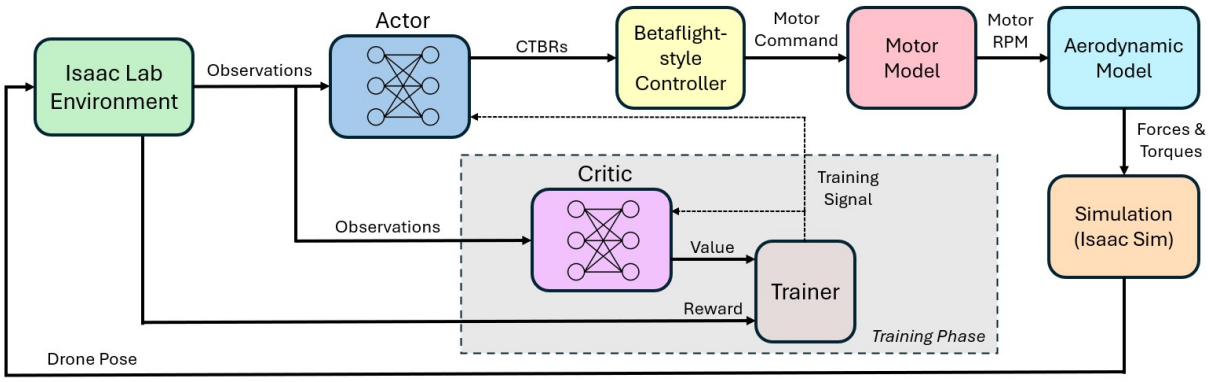

Figure 3.1: General data flow diagram for the entire project

Fig. 3.1 presents the general data flow of the project. The architecture follows a symmetric Actor-Critic structure, where the actor is trained to map observations—such as image pixels in the case of vision-based flight—to collective thrust and body rates (CTBRs). The critic, in turn, learns to estimate a value function representing the expected future reward of a given action. Together with the actual rewards received from the environment, this value estimate provides a training signal that enables both the actor and critic to improve through experience.

The CTBRs (collective thrust and body rates) produced by the actor are sent to a Betaflight-style controller, just like how a pilot's stick inputs would be processed by the Betaflight firmware on a real FPV drone. The controller then determines how much power each motor should produce, using values between 0 and 1. In practice, these values would go to the electronic speed controllers (ESCs), where a signal of 1 means full power and 0 means the motor is off. Since modeling the full electrical system of a drone is quite complex, a simplified motor model is used instead. It converts the 0–1 motor outputs into RPM values, which are then used to calculate the forces and torques acting on the drone. This is done with a custom aerodynamic model that assumes these forces are proportional to the square of each rotor's speed. The computed forces are applied in the Isaac Sim simulation, which updates the drone's pose. That new pose is then used by the Isaac Lab environment to generate observations, compute rewards, and move the simulation forward.

The following subsections provide a detailed explanation of each component of the project, describing how they work and contribute to the whole project.

3.2.2 Multicopter Implementation and Aerodynamic Modeling

In the current Isaac Lab framework, quadcopter environments are based on the `CRAZYFLIE_CFG` configuration, which uses the generic `Articulation` class. This class can handle a wide range of articulated robots, but does not natively represent how a multirotor produces lift or yaw. The simulator does not model individual motors or the way their spinning blades produce thrust and torque. Instead, the overall force and moment acting on the drone are simply fed into the simulation using the `set_external_forces_and_torques()` method.

This means that instead of modeling the physical process that generates these forces, it simply applies the force and torque wrench directly to the drone's body. For example, in the environment `Isaac-Quadcopter-Direct-v0`, the policy learns to output the forces and torques needed to move the drone towards a goal position. This type of simulation, however, is not helpful in cases like these, where a more complex aerodynamic simulation is needed.

To overcome this limitation, a new asset type called `Multicopter` was introduced. Unlike `Articulation`, it takes rotor speed commands as input and works out the resulting aerodynamic forces and torques inside itself. This brings the control interface closer to how a real drone behaves: rather than feeding in total forces, the controller sets the motors' rotational speeds, and the simulator calculates the aerodynamic effects they produce. This development was carried out in collaboration with Mario Gómez Andreu, a collaborator of Comillas Pontifical University.

The aerodynamic model calculates the thrust generated by each motor by assuming it is proportional to the square of its angular velocity:

$$f_i = k_f \cdot \omega_i^2 \tag{3.1}$$

where k_f is the thrust coefficient for rotor i and ω_i is its angular velocity in radians per second. The resulting force f_i is applied along the rotor's own local z-axis.

A similar calculation taking into account the direction of each rotor is done to obtain the yaw torque:

$$\tau_z = \sum_{i=1}^{N} k_m \cdot d_i \cdot \omega_i^2 \tag{3.2}$$

where k_m is the moment coefficient and $d_i \in \{-1, +1\}$ represents whether rotor i spins clockwise or counterclockwise. In a balanced hover, counter-rotating pairs produce equal and opposite yaw torques, which cancel unless a speed difference is commanded to produce rotation.

The drone parameters needed for this model are kept in the configuration class `MultiCopterCfg`. This class specifies characteristics such as:

- Number of rotors and their associated body names in the USD file.
- Spin direction of each rotor.
- Aerodynamic coefficients (k_f, k_m) and rotor speed limits.

- Mass and inertia of the drone's base body.
- Geometric offsets for the rotor positions.

By keeping these parameters in a configuration file, the same simulation code can be reused for different multirotor designs without much code modification.

This new functionality is spread across several files. The `multicopter.py` script contains the core simulation code, including the `_apply_aerodynamics_model()` function, which translates rotor speed inputs into aerodynamic forces and moments at each simulation step. The `multicopter_cfg.py` file defines the `MulticopterCfg` class, which stores all configuration parameters. In the `robots` directory, `quadcopter.py` specifies the quadcopter setups. Here, a new `CRAZYFLIE_CFG_MULTICOPTER` was added as a direct replacement for the existing `CRAZYFLIE_CFG`. It uses the same Crazyflie USD file as before but changes the simulation asset type from `Articulation` to `Multicopter`. This change keeps the same visual model and mass properties while introducing a rotor-speed-driven aerodynamic model.

In the `CRAZYFLIE_CFG_MULTICOPTER`, the actuators are configured so they do not apply any direct mechanical movement to the joints. Instead, all motion of the drone results from the aerodynamic forces generated by the rotor model. Commands are provided via `set_rotor_velocity_target()`, which sets the desired speed for each rotor. The aerodynamic model then enforces speed limits, computes the corresponding forces and torques, and applies them to the relevant rotor bodies and to the base body in the physics engine.

By modeling the way each motor generates forces, the simulation now represents the same fundamental process that causes a real quadcopter to move, narrowing the gap between simulated training and deployment on actual hardware and consequently improving the likelihood that the trained policy performs as expected on an actual drone.

3.2.3 Betaflight-Style Controller

In reinforcement learning works like this one, in which a drone is trained through thousands of episodes in simulation before being deployed in physical hardware, it is crucial that the drone's response to commands in simulation and reality is as close as possible. In order to achieve this, the same PID controller should be used for both environments.

Similar works on this line of research commonly use custom PID controllers to achieve this; however, this restricts the policy to only work with drones that have been flashed with that specific software, which defeats the purpose of the AI agent being able to control any regular FPV drone.

Betaflight [36] SITL aims to solve this by allowing the simulation to interface with Betaflight, the most widely used open-source firmware in recreational FPV drones. Although this is a great step forward in the normalization of PID controllers for reinforcement learning applications, a

major limitation is its serial-based nature: it does not allow parallelized computation, which is crucial for policy training.

This work addresses that limitation by implementing a vectorized, PyTorch-based PID controller that emulates Betaflight's control law and default gains, performing batched, parallel computation rather than computing each control step iteratively in a loop.

Controller Structure

The controller is designed to follow the same logic used in Betaflight, with proportional (P), integral (I), and derivative (D) terms evaluated for each rotational axis (roll, pitch, yaw). The main difference is that here everything runs in a vectorized PyTorch form, so the calculations for all simulated drones are done at the same time in a batch fashion rather than one at a time in a loop.

In the first place, after the desired rates $\boldsymbol{r}_{\text{target}}$ are given and current body rates $\boldsymbol{r}_{\text{actual}}$ are obtained from the drone's gyro, the controller calculates the rate error $\boldsymbol{e(t)} \in \mathbb{R}^{\text{N.}^1 \times 32}$ in radians per second using the following formula:

$$\boldsymbol{e}(t) = \boldsymbol{r}_{\text{target}}(t) - \boldsymbol{r}_{\text{actual}}(t) \tag{3.3}$$

It is important to note that due to the parallelized approach previously mentioned, each of the PID terms lives in $\mathbb{R}^{\text{N}\times 3}$, as they hold the PID values for the three axes of each environment.

The p-term is calculated directly from $\boldsymbol{e(t)}$, scaled by the proportional gains $\boldsymbol{k_p} \in \mathbb{R}^3$ (roll, pitch & yaw) and by Betaflight's constant PTERM_SCALE:

$$\boldsymbol{p}(t) = \boldsymbol{k_p} \cdot \text{PTERM\_SCALE} \odot \boldsymbol{e}(t) \tag{3.4}$$

Here $\odot$ means element-wise multiplication, since the three axes are handled separately. The use of scaling factors like PTERM_SCALE enables an easier PID tuning with more manageable parameters.

For the i-term, the controller adds up the error over time to ensure precision. Similar to the p-term, it is also scaled by gains $\boldsymbol{k_i} \in \mathbb{R}^3$ and its respective ITERM_SCALE:

$$\boldsymbol{i}(t) = \boldsymbol{i}(t - \Delta t) + \boldsymbol{k_i} \cdot \text{ITERM\_SCALE} \cdot \Delta t^3 \odot \boldsymbol{e}(t) \tag{3.5}$$

[4] Following Betaflight's original controller, the d-term is taken from the derivative of the measured rates ($\boldsymbol{r}_{\text{actual}}$) and not from the error, which is a common choice in flight control that

[2] N: Number of environments

[4] Δt: Simulation timestep

leads to smoother control and a more stable response:

$$\boldsymbol{\delta}(t) = -\frac{\hat{\boldsymbol{r}}_{\text{actual}}(t) - \hat{\boldsymbol{r}}_{\text{actual}}(t - \Delta t)}{\Delta t} \tag{3.6}$$

$$\boldsymbol{d}(t) = \boldsymbol{k_d} \cdot \text{DTERM\_SCALE} \odot \boldsymbol{\delta}(t) \tag{3.7}$$

The rates $\hat{\boldsymbol{r}}_{\text{actual}}$ are the result of the original rates passed through several low-pass filters in Betaflight. However, this functionality has been intentionally omitted ($\hat{\boldsymbol{r}}_{\text{actual}} = \boldsymbol{r}_{\text{actual}}$) for the sake of simplicity and a reduced computational load, although they could be integrated in the future to better emulate the behavior of Betaflight.

The feedforward term estimates the needed control action from the change in the target rate. In other words, it reduces the response time of the drone by acting based on how fast the pilot moves the stick on the RC controller. Although this term is present in the default Betaflight configuration, it has been left out in this setup because, with a neural network acting as the pilot, the desired rates are rarely as smooth as those from a human pilot. Unlike a real pilot, the AI agent can output very different target rates from one simulation step to the next, causing an extremely large PID impact.

Finally, the total PID signal $\boldsymbol{pid}_{\text{sum}} \in \mathbb{R}^{\text{N}\times 3}$, omitting the f-term, is obtained by adding each of the separate terms and then clamping it.

$$\boldsymbol{pid}_{\text{sum}}(t) = \boldsymbol{p}(t) + \boldsymbol{i}(t) + \boldsymbol{d}(t) \tag{3.8}$$

$$\boldsymbol{pid}_{\text{sum}}[roll, pitch] = \frac{clamp(\boldsymbol{pid}_{\text{sum}}[roll, pitch], -L_{rollpitch}, +L_{rollpitch})}{\text{PID\_MIXER\_SCALING}} \tag{3.9}$$

$$\boldsymbol{pid}_{\text{sum}}[yaw] = \frac{clamp(\boldsymbol{pid}_{\text{sum}}[roll, pitch], -L_{yaw}, +L_{yaw})}{\text{PID\_MIXER\_SCALING}} \tag{3.10}$$

The PID scaling factors from Betaflight used in this subsection are:

PTERM_SCALE $= 0.032029$

ITERM_SCALE $= 0.244381$

DTERM_SCALE $= 0.000529/1.5$[5]

PID_MIXER_SCALING $= 1000.0$

[5]Due to the reduced PID rate in simulation (200Hz) compared to deployment (>4000Hz), the D gain is slightly reduced from the original 0.000529 to prevent oscillations that crash the simulation.

Motor Mixing

The axis outputs are turned into motor commands using the quadcopter mixer matrix $\boldsymbol{M} \in \mathbb{R}^{4\times 3}$:

$$\boldsymbol{M} = \begin{array}{c} \\ \text{FR} \\ \text{RR} \\ \text{RL} \\ \text{FL} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Roll} & \text{Pitch} & \text{Yaw} \end{array} \\ \begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & 1 & 1 \\ 1 & -1 & -1 \end{bmatrix} \end{array} \tag{3.11}$$

$$\boldsymbol{mix}(t) = \boldsymbol{pid}_{\text{sum}}(t) \cdot \boldsymbol{M}^{\top} \tag{3.12}$$

$$\boldsymbol{mix}(t) \in \mathbb{R}^{N\times 4}$$

Here, each row of $\boldsymbol{M}$ contains the direction at which each motor should spin for a positive roll, pitch, or yaw. The motor labels are: FR (front right), RR (rear right), RL (rear left), and FL (front left).

To keep full control authority even when the throttle is low (as in Betaflight's Airmode[37]), the mixer outputs are normalized:

$$\text{norm\_factor} = \frac{1}{\max\left(mix_{\max} - mix_{\min}, 1\right)} \tag{3.13}$$

$$\tilde{\boldsymbol{m}}(t) = \text{norm\_factor} \cdot \boldsymbol{mix}(t) \tag{3.14}$$

This first step ensures that the range in $\tilde{\boldsymbol{m}}(t)$ is always capped to a maximum of 1. Next, the throttle input t is clamped to ensure that each motor output is in the range $[0, 1]$.

$$\tilde{t}(t) = clamp(t(t), -\tilde{m}_{\min}(t), 1 - \tilde{m}_{\max}(t)) \tag{3.15}$$

$$\boldsymbol{motor\_command}(t) = \tilde{\boldsymbol{m}}(t) + \tilde{t} \cdot \mathbf{1}(t) \tag{3.16}$$

Although the motor command should already be in the range of $[0, 1]$, it is clamped again to prevent a small decimal error from causing a negative value.

$$\boldsymbol{motor\_command}(t) = clamp(\boldsymbol{motor\_command}(t), 0, 1) \tag{3.17}$$

These formulas emulate the exact way that Betaflight's Airmode works. This enables the pilot to have complete authority over the drone even when no throttle input is provided. To better understand this, an example is useful:

Example 1:

$$\boldsymbol{mix}(t) = [-0.20.5, 0.3, 0.6]$$

$$\text{norm\_factor} = \frac{1}{max(0.6 - (-0.2), 1)} = \frac{1}{1} = 1$$

$$\tilde{\boldsymbol{m}}(t) = [-0.2, 0.5, 0.3, 0.6] \cdot 1 = [-0.2, 0.5, 0.3, 0.6]$$

$$\tilde{t}(t) = 0.1$$

$$\begin{aligned}\tilde{t}(t) &= \text{clamp}\big(t(t), -\tilde{m}_{\min}(t), 1 - \tilde{m}_{\max}(t)\big) \\ &= \text{clamp}(0.1, -(-0.2), 1 - 0.6) \\ &= \text{clamp}(0.1, 0.2, 0.4) = 0.2\end{aligned}$$

In this example, a flight controller without Airmode activated would result in the output of motor 1 being negative, which would get clipped to 0. This clipping would result in a loss of control authority. Instead, the throttle is forced to 0.2 instead of the requested 0.1 by the pilot. This shows how Betaflight Airmode prioritizes body rate control over throttle control.

Essentially, the controller provides the PID loop with all the motor power needed, and leaves whatever wiggle room there is available, if any, for throttle control. The range ($mix_{max} - mix_{min}$) of the mix represents how strong a PID signal is being executed. If $range = 1$, full command authority of the motors is dedicated to attitude control (PID), leaving no wiggle room for throttle movement. If $range < 1$, the wiggle room for throttle is $1 - range$. In this example, the range was 0.8, leaving a 0.2 wiggle room for throttle (0.2 to 0.4).

Rotor Speed Mapping

Finally, the normalized outputs are mapped to physical rotor speeds with a power-law curve:

$$\boldsymbol{\omega}(t) = \min\left(\boldsymbol{motor\_command}(t)^{\odot\alpha} \odot \boldsymbol{\omega}_{\max},\ \boldsymbol{\omega}_{\max}\right) \tag{3.18}$$

where α is an experimental motor parameter that shapes the thrust curve, $\odot\alpha$ represents an

element-wise power, and $\omega_{\max}$ is the maximum speed of each rotor.

3.2.4 Isaac Lab Environment

Scene Layout

The Isaac Lab scene used to train the policy consists of a 2-meter-wide, 1-meter-tall, and 10-meter-long corridor that the drone has to traverse while avoiding three cubes that act as obstacles. Fig. 3.2 shows an image of the corridor with the ceiling removed to allow a better view. Fig. 3.3 shows the perspective of the drone inside the corridor. The objective is to train the AI agent to go through this corridor as fast as possible without hitting any walls or obstacles until it goes through 9 meters of the corridor.

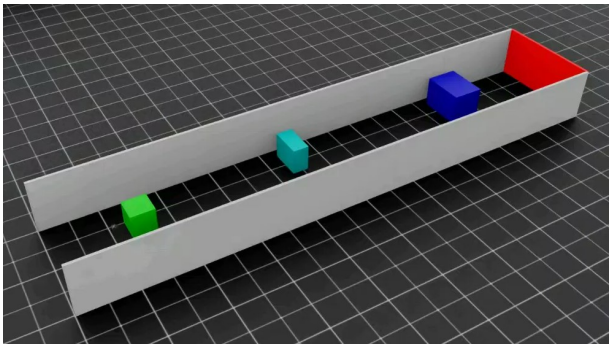

Figure 3.2: Image of the corridor used in the environment where the policy is trained. Ceiling removed for better visualization.

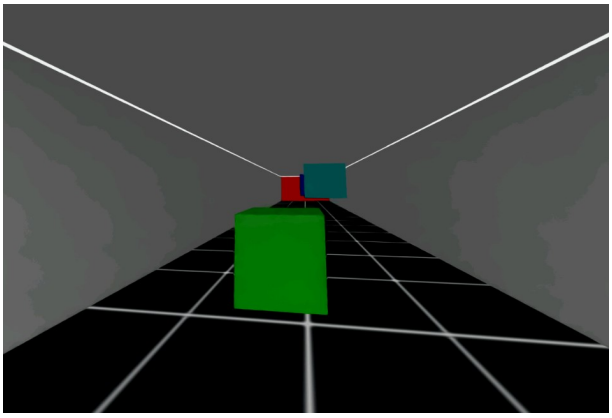

Figure 3.3: Image of the perspective of the drone inside the corridor

In order to prevent other drones from appearing in the view of each agent, a separate corridor with its respective obstacles is created for each environment, as shown on Fig. 3.4.

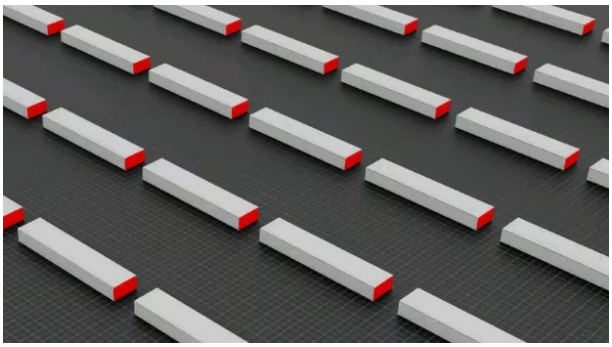

Figure 3.4: Image of multiple parallel environments used during training

Observations

As a way to test the newly developed aerodynamic model and controller on both vision- and state-based flight, experiments have been conducted for both cases. The observation space for each setting is defined as follows:

1. **State-based flight**
   In the case of state-based flight, where the policy has access to ground-truth simulation data, the observation vector s_t at each timestep is composed of:

   - **Orientation:** The drone's orientation expressed in Euler angles,

     $$\boldsymbol{\theta} = (\phi, \vartheta, \psi) \in \mathbb{R}^3$$

     where ϕ is roll, ϑ is pitch, and ψ is yaw. The values are normalized such that $1 = 2\pi$ rad (i.e. $360°$) and $-1 = -2\pi$ rad. Orientation is expressed in the *world frame*.
   - **Angular velocity:** Body angular rates expressed as

     $$\dot{\boldsymbol{\theta}} = (\dot{\phi}, \dot{\vartheta}, \dot{\psi}) \in \mathbb{R}^3$$

     where $\dot{\phi}$ is roll rate, $\dot{\vartheta}$ is pitch rate, and $\dot{\psi}$ is yaw rate. These values are normalized with the same scale as orientation ($1 = 2\pi$ rad/s). Angular velocity is expressed in the *body frame*.
   - **Linear velocity:**

     $$\boldsymbol{v_b} = (v_x, v_y, v_z) \in \mathbb{R}^3$$

     given in meters per second (m/s). This vector is expressed in the *body frame*.
   - **Projected gravity vector:**

     $$\boldsymbol{g_b} \in \mathbb{R}^3, \quad \|g_b\| = 1$$

     a unit vector pointing in the direction of gravity, expressed in the *body frame*.

- **Right wall vector:**
$$\boldsymbol{d}_{\text{right}} \in \mathbb{R}^3$$
pointing from the drone to the nearest point on the right wall. Its magnitude $\|d_{\text{right}}\|$ equals the distance to the wall. This vector is expressed in the *body frame*.
- **Left wall vector:**
$$\boldsymbol{d}_{\text{left}} \in \mathbb{R}^3$$
pointing from the drone to the nearest point on the left wall. Its magnitude $\|d_{\text{left}}\|$ equals the distance to the wall. This vector is expressed in the *body frame*.
- **Ceiling vector:**
$$\boldsymbol{d}_{\text{ceil}} \in \mathbb{R}^3$$
pointing from the drone to the closest point on the ceiling. Its magnitude $\|d_{\text{ceil}}\|$ equals the distance to the ceiling. This vector is expressed in the *body frame*.
- **Ground vector:**
$$\boldsymbol{d}_{\text{ground}} \in \mathbb{R}^3$$
pointing from the drone to the closest point on the ground. Its magnitude $\|d_{\text{ground}}\|$ equals the distance to the ground. This vector is expressed in the *body frame*.
- **Desired direction:**
$$\boldsymbol{d}_{\text{target}} \in \mathbb{R}^3, \quad \|\boldsymbol{d}_{\text{target}}\| = 1$$
a unit vector pointing from the drone towards the end of the corridor. This vector is expressed in the *body frame*.
- **Next obstacle vector:**
$$\boldsymbol{d}_{\text{obs}} \in \mathbb{R}^3$$
pointing from the drone towards the centroid of the nearest obstacle directly ahead. Its magnitude $\|d_{\text{obs}}\|$ encodes the distance to the obstacle. This vector is expressed in the *body frame*.
- **Previous action:**
$$\boldsymbol{a_{t-1}} \in \mathbb{R}^4$$
the last control output of the policy (collective thrust and body rates, CTBRs), included to promote smoother actions and temporal consistency.

2. **Vision-based flight**
In the case of vision-based flight, where the policy has access to the view of an onboard monocular RGB camera on the drone in addition to certain information that a real drone would have access to during deployment, the observation vector s_t at each timestep is composed of:

- **Extracted features from RGB frame:** At each timestep, the agent receives the output of a monocular onboard RGB camera. To make this input manageable and consistent with pretrained vision backbones, the raw RGB frame is preprocessed and passed through a convolutional feature extractor.
The processing steps are as follows:

  - Each 60×60 image is normalized channel-wise using the standard ImageNet [38] statistics:

$$\text{mean} = (0.485,\ 0.456,\ 0.406), \quad \text{std} = (0.229,\ 0.224,\ 0.225).$$

  - The normalized frames of dimension $(N, 60, 60, 3)$ are passed through the convolutional feature extractor of a pretrained MobileNetV3-Small [39] model initialized with ImageNet weights and kept frozen (i.e., not updated) during training. This produces a feature map of size $(N, 576, 2, 2)$.
  - To reduce the dimension of each feature map from 2×2 to a single scalar, global average pooling is done, resulting in a feature vector of dimension 576 for each environment:

$$\boldsymbol{f}_{\text{rgb}} \in \mathbb{R}^{576}$$

- **Orientation:** The drone's orientation expressed in Euler angles,

$$\boldsymbol{\theta} = (\phi,\ \vartheta,\ \psi) \in \mathbb{R}^3$$

  where ϕ is roll, ϑ is pitch, and ψ is yaw. The values are normalized such that $1 = 2\pi$ rad (i.e. $360°$) and $-1 = -2\pi$ rad. Orientation is expressed in the *world frame*. During deployment, this orientation is estimated in the flight controller by fusing data from the IMU sensors.

- **Angular velocity:** Body angular rates expressed as

$$\dot{\boldsymbol{\theta}} = (\dot{\phi},\ \dot{\vartheta},\ \dot{\psi}) \in \mathbb{R}^3$$

  where $\dot{\phi}$ is roll rate, $\dot{\vartheta}$ is pitch rate, and $\dot{\psi}$ is yaw rate. These values are normalized with the same scale as orientation ($1 = 2\pi$ rad/s). Angular velocity is expressed in the *body frame*.

- **Projected gravity vector:**

$$\boldsymbol{g_b} \in \mathbb{R}^3, \quad \|g_b\| = 1$$

  a unit vector pointing in the direction of gravity, expressed in the *body frame*.

- **Desired direction:**

$$\boldsymbol{d}_{\text{target}} \in \mathbb{R}^3, \quad \|\boldsymbol{d}_{\text{target}}\| = 1$$

  a unit vector pointing from the drone towards the end of the corridor. This vector is expressed in the *body frame*.

- **Previous action:**

$$\boldsymbol{a_{t-1}} \in \mathbb{R}^4$$

  the last control output of the policy (collective thrust and body rates, CTBRs), included to promote smoother actions and temporal consistency.

Rewards

In order to correctly guide the policy training and enable the drone to successfully follow the corridor while avoiding obstacles, a dense reward function is defined. eAt each timestep, the reward $r_t \in \mathbb{R}$ is given as the weighted sum of several components:

$$r_t = r_{\text{progress}} + r_{\text{end}} + r_{\text{collision}} + r_{\text{orientation}} + r_{\text{smooth}}$$

where each term is defined as follows:

- **Progress:**

  $$r_{\text{progress}} = \Delta p \cdot \alpha_{\text{progress}}$$

  where $\Delta p = p_t - p_{t-1}$ represents the progress made by the drone along the longitudinal axis of the corridor (world y direction) between consecutive timesteps, and α_{progress} is a positive scaling factor. This term encourages continuous forward movement along the corridor.

- **Reaching the end:**

  $$r_{\text{end}} = \mathbf{1}_{\text{end}} \cdot \alpha_{\text{end}} \left(1 - \frac{n_t}{N_{\max}}\right)$$

  where $\mathbf{1}_{\text{end}}$ is an indicator that equals 1 if the drone has reached the end of the corridor and 0 otherwise, n_t is the number of elapsed timesteps in the episode, $N_{\max}$ is the maximum timesteps per episode and α_{end} is a positive scaling factor. This term yields a larger reward for finishing earlier (it is zero when $n_t = N_{\max}$).

- **Collision penalization:**

  $$r_{\text{collision}} = -\mathbf{1}_{\text{collision}} \cdot \alpha_{\text{collision}}$$

  where $\mathbf{1}_{\text{end}}$ is an indicator that equals 1 if the drone has collided with an obstacle, wall, or ground (terminating the episode) and 0 otherwise. $\alpha_{\text{collision}}$ is a positive constant that penalizes failure.

- **Orientation:**

  $$r_{\text{orientation}} = e^{(4 \cdot d_{\text{target},x} - 3)} \cdot \alpha_{\text{orientation}} \cdot \Delta t$$

  where $d_{\text{target},x}$ is the x-component of the desired direction vector expressed in the body frame (which reflects the forward alignment of the drone), $\alpha_{\text{orientation}}$ is a positive scaling factor, and Δt is the simulation timestep. This reward encourages the drone to maintain its orientation aligned with the forward direction of the corridor.

- **Control smoothness:**

  $$r_{\text{smooth}} = -\left\|\boldsymbol{a_t} - \boldsymbol{a_{t-1}}\right\|^2 \cdot \alpha_{\text{smooth}} \cdot \Delta t$$

  where a_t and a_{t-1} are the current and previous actions (CTBRs) respectively, α_{smooth} is

a positive scaling factor, and Δt is the simulation timestep. This term penalizes large variations and aggressiveness in control, encouraging a smoother and more stable flight.

Here, $\alpha_{\text{progress}}, \alpha_{\text{end}}, \alpha_{\text{collision}}, \alpha_{\text{orientation}}, \alpha_{\text{smooth}}$ are tunable reward parameters defined in the configuration file. For the experiments presented in this work, the following values were used:

$$\alpha_{\text{collision}} = 50.0, \quad \alpha_{\text{progress}} = 6.0, \quad \alpha_{\text{end}} = 80.0, \quad \alpha_{\text{orientation}} = 5.0, \quad \alpha_{\text{smooth}} = -5.0$$

These values are chosen empirically to balance the speed, smoothness, and safety of the policy. The combination of these terms promotes steady forward progress, penalizes collisions, reinforces correct orientation, and encourages smoother control signals.

3.2.5 Neural Network Architecture

As previously mentioned, the training is done using a symmetric Actor–Critic algorithm, where both neural networks (actor and critic) have the same architecture and differ just in their output layers.

The policy network (actor) learns to output control commands in the form of collective thrust and body rates (CTBRs) based on the information from the observations, while the value network learns to output a value scalar that estimates the expected return from a given state.

Control policy. The output of policy is modeled as a Gaussian distribution, parameterized by a mean and a log-standard deviation:

$$\pi_\theta(a_t \mid s_t) = \mathcal{N}(\mu_\theta(s_t),\, \sigma^2),$$

where s_t is the state (observation vector) at time t, a_t is the action (CTBRs), and μ_θ is obtained from a three-layer feedforward network with 512, 256, and 128 neurons per layer and Exponential Linear Unit (ELU) activations. The neural network architectures for state-based and vision-based tasks are displayed in Fig. 3.5 and Fig. 3.6 respectively. The log-standard deviation, instead of being predicted by the network, is kept as a separate trainable parameter vector. It is initialized at zero (corresponding to $\sigma = 1$) and clipped to the range $[-20, 2]$ to avoid instability. The log-space representation of the standard deviation is used to ensure it remains strictly positive.

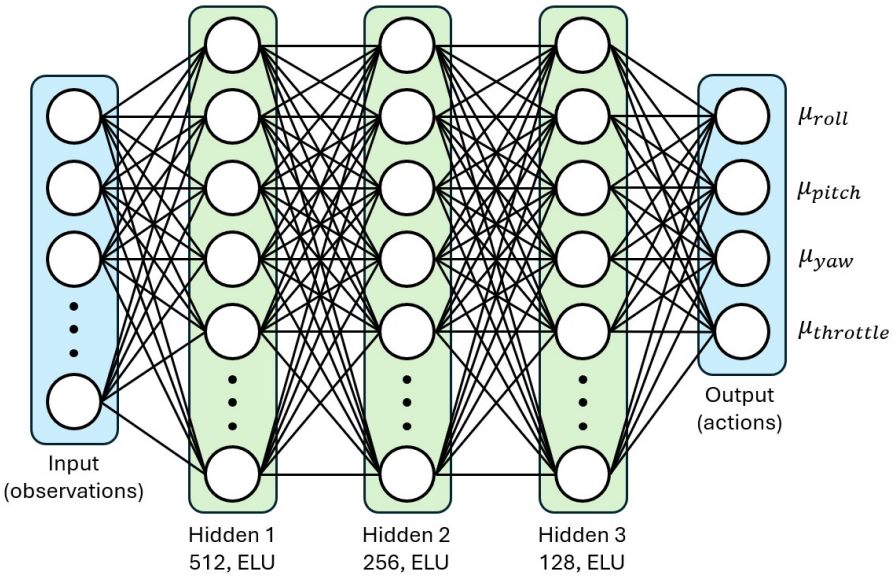

Figure 3.5: Policy network (actor) architecture for state-based flight

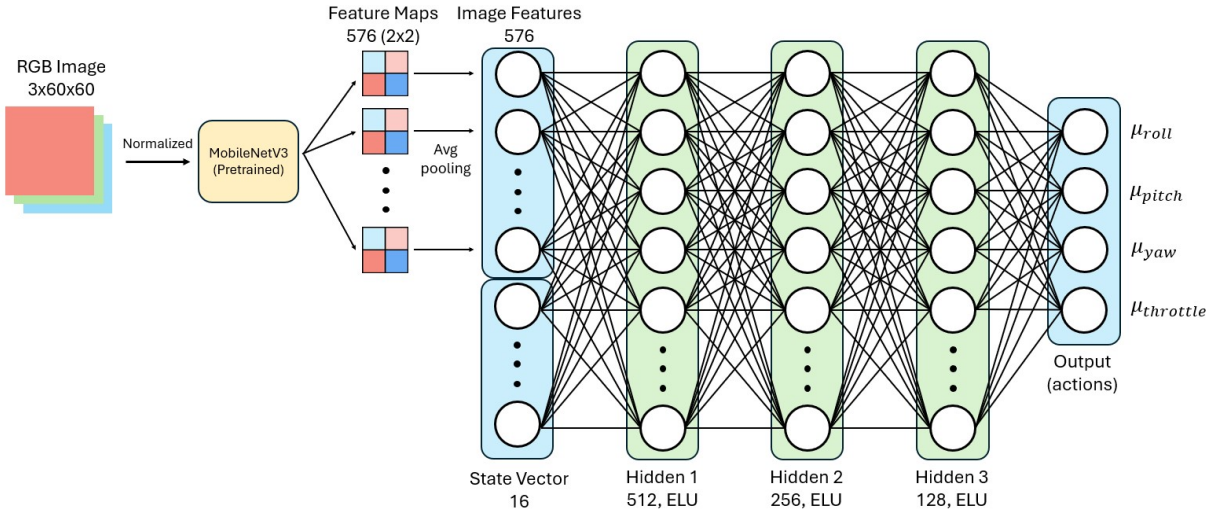

Figure 3.6: Policy network (actor) architecture for vision-based flight

Value function. Unlike the policy, the value function is just a scalar value instead of a distribution. This value is obtained from the critic neural network, which has a singular output instead of one per action. The following is the notation for the value function:

$$V_\phi(s_t) \in \mathbb{R},$$

where ϕ denotes the network parameters. It is made up of three fully connected layers with 512, 256, and 128 neurons each and ELU activations, mapping the observation vector to a scalar

estimate of the state value. The neural network architectures for both tasks are shown in Fig. 3.7 and Fig. 3.8.

This estimated value $V_\phi(s_t)$ represents the expected total future reward if the policy π is followed from that state s_t onward. In other words, it represents the highest possible value that the policy expects to receive from a given state.

$$V_\phi(s_t) = \mathbb{E}_\pi \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \;\middle|\; s_t \right]$$

When an actual action a_t is taken and the reward r_t is obtained in the next state s_{t+1}, the value function of the original state s_t following the taken action a_t can be calculated:

$$V_\phi\left(s_t \mid a_t\right) = r_t + \gamma\, \mathbb{E}_\pi \left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \;\middle|\; s_{t+1} \right]$$

The *advantage*, which is used as a training signal, can now be computed using these two terms:

$$A_t = V_\phi\left(s_t \mid a_t\right) - V_\phi(s_t)$$

If this newly calculated value $V_\phi\left(s_t \mid a_t\right)$ is higher than the expected value $V_\phi(s_t)$, the advantage is positive, which indicates that the action performed better than expected, encouraging the actor to increase its likelihood. The same applies to a negative advantage, which signals a worse-than-expected performance, discouraging similar actions in the future.

The advantage is also used by the critic as a training signal to learn to better predict the value of a given state assuming the future actions are taken on-policy[6].

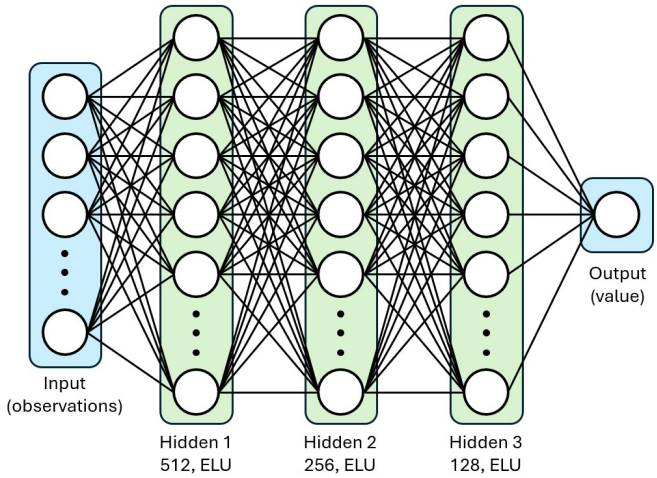

Figure 3.7: Value network (critic) architecture for state-based flight

[6]Although individual actions may not be taken strictly on-policy, as they follow a Gaussian distribution, the average of multiple actions does portray how the actual on-policy behavior would be, making the advantage a suitable signal to train the critic too.

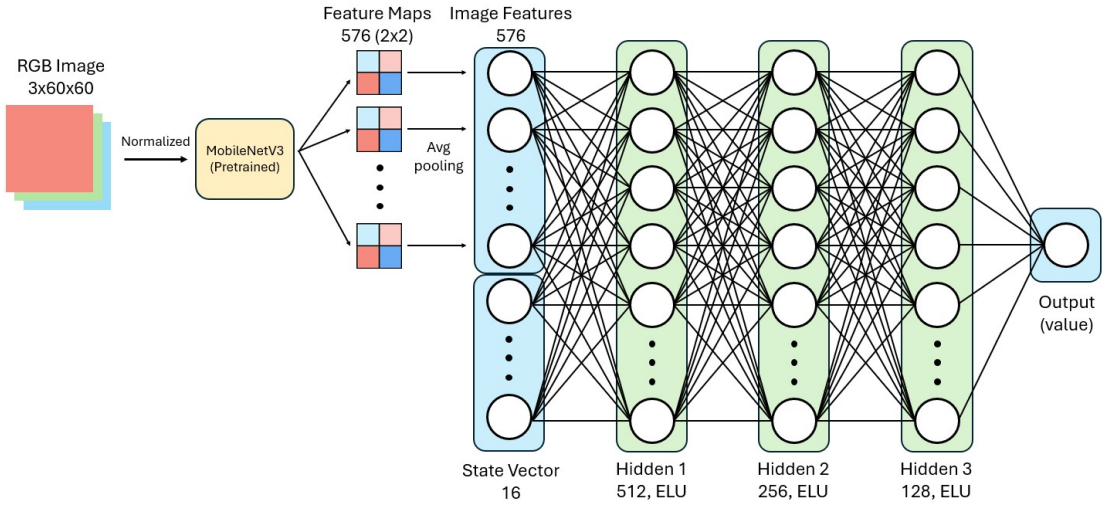

Figure 3.8: Value network (critic) architecture for vision-based flight

Agent configuration. Training is performed using the Proximal Policy Optimization (PPO) [18] algorithm with the following main hyperparameters:

- Discount factor $\gamma = 0.99$ and GAE parameter $\lambda = 0.95$, balancing short- and long-term rewards.
- Learning rate 5×10^{-4}, adapted dynamically with a KL-divergence–based scheduler (threshold 0.016).
- 24 rollouts per iteration, with 5 learning epochs and 4 mini-batches per epoch.
- Gradient clipping at 1.0 to avoid exploding gradients.
- Clipping parameters for PPO ratio and value function both set to 0.2, following standard PPO practice.

Chapter 4

Analysis of Results

After presenting the methodology in the previous chapter, the developed algorithms will now be evaluated. The experiments are designed with three main objectives:

1. Validate the quadcopter model: Demonstrate that the newly implemented quadcopter robot and aerodynamic model behave as expected.

2. Assess the controller: Verify that the Betaflight-style parallelizable controller provides stable and reliable low-level control, enabling the policy to maintain flight.

3. Evaluate the environment: Confirm that the developed Isaac Lab environment is suitable for training autonomous flight policies in drones.

4.1 State-Based Flight

For the state-based experiment, the policy is trained in 4000 parallel environments. The graph in Fig. 4.1 shows the evolution of the mean total reward during a training session with 30 million policy interactions with the environment. This specific experiment takes 10 minutes on a high-end desktop (NVIDIA GeForce RTX 4070 Ti, Intel Core i7-13700, 64GiB RAM), converging in about 5 minutes.

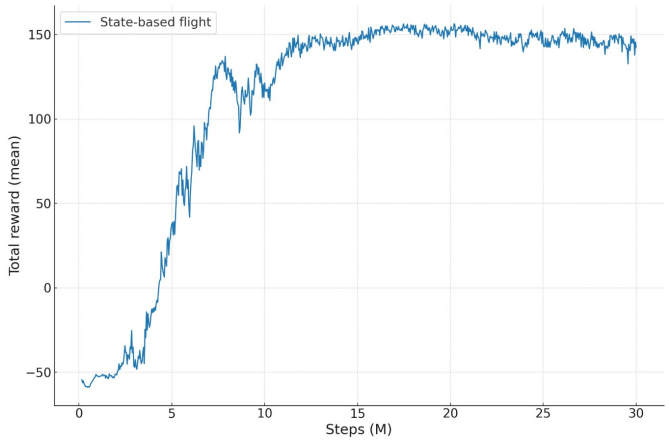

Figure 4.1: Mean of the total reward on the state-based training experiment

Fig. 4.2 shows the evolution of the success rate as the number of interactions with the environment increases. A run is considered successful when the drone reaches the 9-meter mark of the corridor. It is considered a failed run when it crashes into an object like a wall or an obstacle, or when it exceeds the time limit of 8 seconds. As the graph shows, the success rate rapidly increases in the first 10 million interactions, converging at about 97% at 15 million steps.

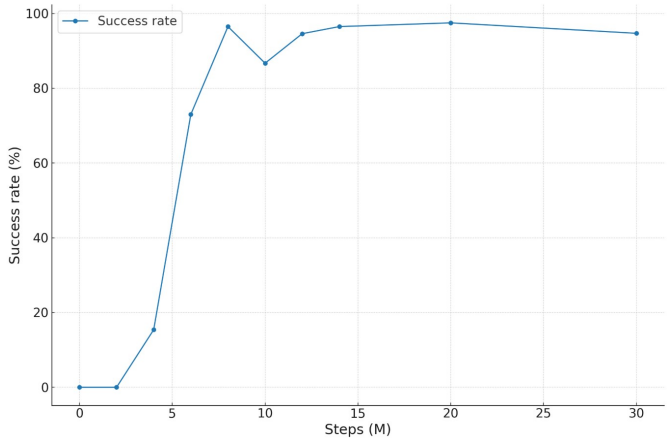

Figure 4.2: Success rate on the state-based training experiment

These results show how a state-based agent is able to learn in the environment and adapt to a changing obstacle course. Fig. 4.3 shows the trajectory of the best agent in one of the obstacle configurations.

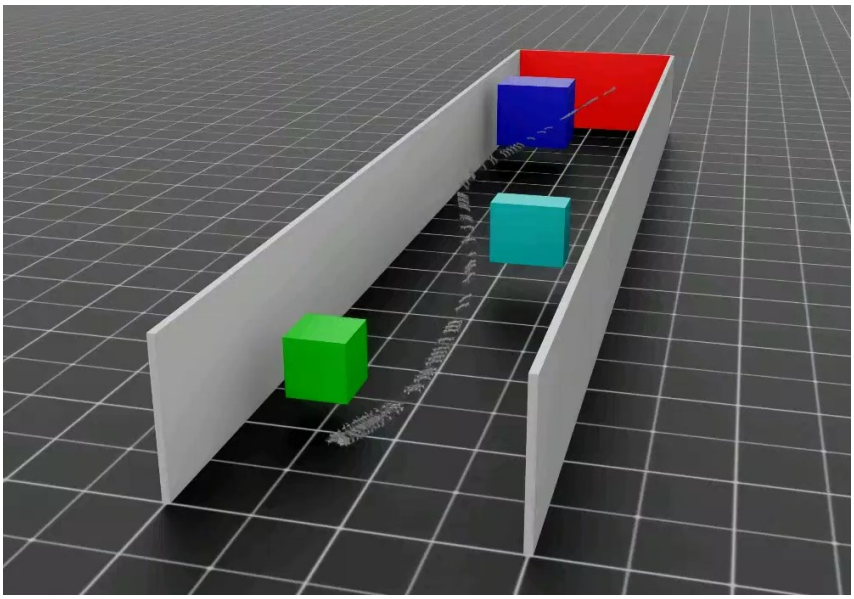

Figure 4.3: Best agent performance on the state-based environment. Ceiling removed for better visualization.

4.2 Vision-Based Flight

In the vision-based experiment, an onboard camera is set up in each of the environments. The use of these cameras heavily increases the computational load, limiting the number of parallel environments to be used to just 400. The evolution of the mean of the total reward of a 30 million interaction training session is displayed on Fig. 4.4. The figure also shows the results of the same network architecture with an added 512 neuron layer. This layer is added as an attempt to allow the network to better understand the complex nature of the problem; however, the results of both experiments are almost identical. This full session took 2 hours on a high-end desktop.

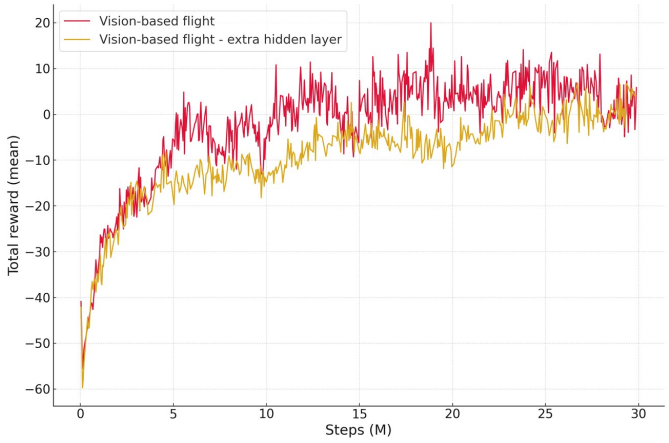

Figure 4.4: Mean of the total reward on the vision-based training experiment

Fig. 4.2 shows the evolution of the success rate as the number of interactions with the environment increases. As mentioned before, a run is considered successful when the drone reaches the 9-meter mark of the corridor. It is considered a failed run when it crashes into an object like a wall or an obstacle, or when it exceeds the time limit of 8 seconds. As the graph shows, the success rate barely surpasses the 12% mark at 20 million interactions and drops to 8% at 30 million.

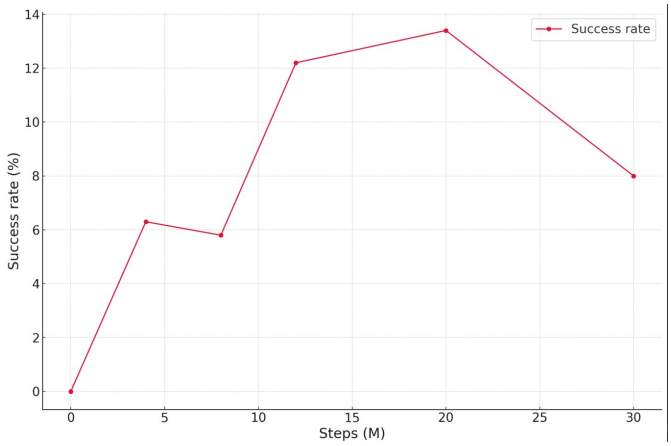

Figure 4.5: Success rate on the vision-based training experiment

Both Fig. 4.4 and Fig. 4.5 show how the vision-based policy is not able to correctly navigate through the environment. The times that it successfully navigates the whole corridor are likely due to luck or blind control. The policy roughly learns to dodge the first object, but fails to avoid the rest while maintaining stability. Fig. 4.6 compares the state-based and vision-based experiments, illustrating the weaker performance of the vision-based policy. These weak results are probably caused by the lack of a memory layer in the vision-based architecture, such as a GRU or other recurrent module, which would allow the policy to retain temporal information across frames and better estimate its speed.

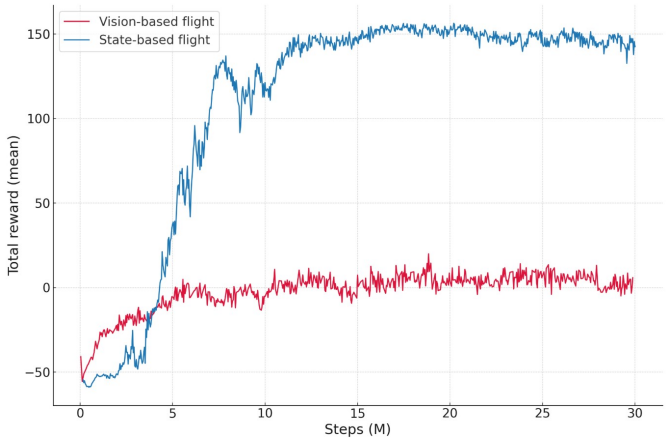

Figure 4.6: Comparison of the mean of the total reward for state and vision-based flight

Chapter 5

Conclusions

5.1 Conclusions on the Methodology

In conclusion, the results have demonstrated that the proposed methodology successfully accomplished all three objectives:

1. **Design and implementation of a custom quadcopter robot and its respective aerodynamic model** within Nvidia's Isaac Lab framework, built on top of Isaac Sim. This provides a foundation for accurate physics-based simulation of quadrotor dynamics.
2. **Implementation of a parallelizable Betaflight-based drone controller** within Isaac Lab. This enables stable control while supporting large-scale parallel training, overcoming the limitations of serial computation in existing Betaflight SITL approaches.
3. **Development of a custom Isaac Lab environment** that integrates the quadcopter robot and Betaflight-based controller in a task where the drone must navigate obstacles to reach a target goal. This environment serves as a platform for evaluating both state-based and vision-based reinforcement learning policies.

Together, these developments establish a new Isaac Lab–based framework for reinforcement learning in vision-based drone control, providing a solid foundation for future research in agile autonomous flight.

5.2 Recommendations for Future Research

As mentioned before, the objective of this work is to lay a foundation for future research in drone flight, not to develop a working neural network architecture for vision-based flight. Although

the presented experiment for pixel-based flight was unsuccessful, it is due to an unfit network architecture rather than a poor training framework. The great success of the state-based flight demonstrates the true potential for this RL environment. Some suggestions to achieve a better performance on vision-based policies are the following:

- Implementation of a more sophisticated neural network architecture that includes a memory layer, such as a GRU [7, 8] or a TCN [5]. This would allow the drone to have a better understanding of its current movement and speed.
- Switch to an asymmetric Actor-Critic architecture, where the critic has access to privileged information such as the position of the obstacles or the location of the drone in the corridor. This type of method has been shown to be quite successful in previous works [4, 5].
- Use of a custom CNN-based feature extractor. Although pretrained feature extractors like MobileNetV3 [39] or EfficientNet [40] offer great feature extraction, these encoders are trained for image classification, which is not useful in applications like these, where the exact position of an object on an image is much more important.
- Addition of textures to walls and obstacles. The use of textures would not only allow the policy to better understand its movement along the corridor, but is essential to achieve a small sim2real gap.

These changes, along with several others, are likely to produce a successful policy capable of navigating the corridor with performance comparable to its state-based counterpart.

Bibliography

[1] Sysnav. *Why GNSS and GPS Do Not Function Properly Indoors*. https://www.sysnav.fr/why-gnss-and-gps-do-not-function-properly-indoors/?lang=en. Blog post; published 31 May 2024; Accessed: 2025-08-26. May 2024.

[2] Antonio Loquercio et al. "Learning high-speed flight in the wild". In: *Science Robotics* 6.59 (Oct. 2021). ISSN: 2470-9476. DOI: 10.1126/scirobotics.abg5810.

[3] Elia Kaufmann et al. "Champion-level drone racing using deep reinforcement learning". In: *Nature* 620.7976 (Aug. 2023), pp. 982–987. DOI: 10.1038/s41586-023-06419-4.

[4] Ismail Geles et al. *Demonstrating Agile Flight from Pixels without State Estimation*. Accessed: 2025-08-28. 2024. arXiv: 2406.12505 [cs.RO]. URL: https://arxiv.org/abs/2406.12505.

[5] Jiaxu Xing et al. *Bootstrapping Reinforcement Learning with Imitation for Vision-Based Agile Flight*. Accessed: 2025-08-28. 2024. arXiv: 2403.12203 [cs.RO]. URL: https://arxiv.org/abs/2403.12203.

[6] Angel Romero et al. *Dream to Fly: Model-Based Reinforcement Learning for Vision-Based Drone Flight*. Accessed: 2025-08-28. 2025. arXiv: 2501.14377 [cs.RO]. URL: https://arxiv.org/abs/2501.14377.

[7] Yuang Zhang et al. "Learning vision-based agile flight via differentiable physics". In: *Nature Machine Intelligence* 7.6 (June 2025), pp. 954–966. ISSN: 2522-5839. DOI: 10.1038/s42256-025-01048-0.

[8] Yu Hu et al. *Seeing Through Pixel Motion: Learning Obstacle Avoidance from Optical Flow with One Camera*. Accessed: 2025-08-28. 2025. arXiv: 2411.04413 [cs.RO]. URL: https://arxiv.org/abs/2411.04413.

[9] Michael Blösch et al. "Vision based MAV navigation in unknown and unstructured environments". In: *Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. Anchorage, Alaska, USA, May 2010, pp. 21–28. DOI: 10.1109/ROBOT.2010.5509920.

[10] Shaojie Shen et al. "Vision-Based State Estimation and Trajectory Control Towards High-Speed Flight with a Quadrotor". In: *Proceedings of Robotics: Science and Systems (RSS)*. RSS Foundation. Berlin, Germany, June 2013, pp. 32–40. URL: https://api.semanticscholar.org/CorpusID:14706601.

[11] Sikang Liu et al. "High speed navigation for quadrotors with limited onboard sensing". In: *Proceedings of the 2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. Stockholm, Sweden, May 2016, pp. 1484–1491. DOI: 10.1109/ICRA.2016.7487284.

[12] Kartik Mohta et al. "Fast, autonomous flight in GPS-denied and cluttered environments". In: *Journal of Field Robotics* 35.1 (Dec. 2017), pp. 101–120. ISSN: 1556-4967. DOI: 10.1002/rob.21774.

[13] Yi Lin et al. "Autonomous aerial navigation using monocular visual-inertial fusion". In: *Journal of Field Robotics* 35 (2018), pp. 23–51. URL: https://api.semanticscholar.org/CorpusID:3670278.

[14] Elia Kaufmann et al. *Deep Drone Acrobatics*. Accessed: 2025-08-28. 2020. arXiv: 2006.05768 [cs.RO]. URL: https://arxiv.org/abs/2006.05768.

[15] Fereshteh Sadeghi and Sergey Levine. *CAD2RL: Real Single-Image Flight without a Single Real Image*. Accessed: 2025-08-28. 2017. arXiv: 1611.04201 [cs.LG]. URL: https://arxiv.org/abs/1611.04201.

[16] Yunfan Ren et al. "Safety-assured high-speed navigation for MAVs". In: *Science Robotics* 10.98 (2025), eado6187. DOI: 10.1126/scirobotics.ado6187. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.ado6187.

[17] Elia Kaufmann, Leonard Bauersfeld, and Davide Scaramuzza. *A Benchmark Comparison of Learned Control Policies for Agile Quadrotor Flight*. Accessed: 2025-08-28. 2022. arXiv: 2202.10796 [cs.RO]. URL: https://arxiv.org/abs/2202.10796.

[18] John Schulman et al. *Proximal Policy Optimization Algorithms*. Accessed: 2025-08-28. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[19] Zhichao Han et al. *Reactive Aerobatic Flight via Reinforcement Learning*. Accessed: 2025-08-28. 2025. arXiv: 2505.24396 [cs.RO]. URL: https://arxiv.org/abs/2505.24396.

[20] Maryam Zare et al. *A Survey of Imitation Learning: Algorithms, Recent Developments, and Challenges*. Accessed: 2025-08-28. 2023. arXiv: 2309.02473 [cs.LG]. URL: https://arxiv.org/abs/2309.02473.

[21] Dean A. Pomerleau. "ALVINN: An autonomous land vehicle in a neural network". In: *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 1. NeurIPS Foundation. Denver, Colorado, USA: Morgan Kaufmann, Dec. 1989, pp. 305–313.

[22] Fadri Furrer et al. "RotorS – A Modular Gazebo MAV Simulator Framework". In: *Studies in Computational Intelligence*. Vol. 625. Jan. 2016, pp. 595–625. ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_23.

[23] Winter Guerra et al. "FlightGoggles: Photorealistic Sensor Simulation for Perception-driven Robotics using Photogrammetry and Virtual Reality". In: *Proceedings of the 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ. Macau, China: IEEE, Nov. 2019, pp. 6941–6948. DOI: 10.1109/IROS40897.2019.8968116.

[24] Shital Shah et al. *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*. Accessed: 2025-08-28. 2017. arXiv: 1705.05065 [cs.RO]. URL: https://arxiv.org/abs/1705.05065.

[25] Yunlong Song et al. *Flightmare: A Flexible Quadrotor Simulator*. Accessed: 2025-08-28. 2021. arXiv: 2009.00563 [cs.RO]. URL: https://arxiv.org/abs/2009.00563.

[26] Jacopo Panerati et al. *Learning to Fly – a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control*. Accessed: 2025-08-28. 2021. arXiv: 2103.02142 [cs.RO]. URL: https://arxiv.org/abs/2103.02142.

[27] Botian Xu et al. *OmniDrones: An Efficient and Flexible Platform for Reinforcement Learning in Drone Control*. Accessed: 2025-08-28. 2023. arXiv: 2309.12825 [cs.RO]. URL: https://arxiv.org/abs/2309.12825.

[28] Fanxing Li et al. *VisFly: An Efficient and Versatile Simulator for Training Vision-based Flight*. Accessed: 2025-08-28. 2024. arXiv: 2407.14783 [cs.RO]. URL: https://arxiv.org/abs/2407.14783.

[29] Greg Brockman et al. *OpenAI Gym*. Accessed: 2025-08-28. 2016. arXiv: 1606.01540 [cs.LG]. URL: https://arxiv.org/abs/1606.01540.

[30] NVIDIA Corporation. *NVIDIA Isaac Lab*. https://developer.nvidia.com/isaac/lab. Accessed: 2025-08-22. 2025.

[31] Kousheek Chandra. *Isaac Drone Racer*. Accessed: 2025-08-22. URL: https://github.com/kousheekc/isaac_drone_racer.

[32] Manolis Savva et al. *Habitat: A Platform for Embodied AI Research*. Accessed: 2025-08-28. 2019. arXiv: 1904.01201 [cs.CV]. URL: https://arxiv.org/abs/1904.01201.

[33] Julian Straub et al. *The Replica Dataset: A Digital Replica of Indoor Spaces*. Accessed: 2025-08-28. 2019. arXiv: 1906.05797 [cs.CV]. URL: https://arxiv.org/abs/1906.05797.

[34] Angel Chang et al. *Matterport3D: Learning from RGB-D Data in Indoor Environments*. Accessed: 2025-08-28. 2017. arXiv: 1709.06158 [cs.CV]. URL: https://arxiv.org/abs/1709.06158.

[35] Betaflight Contributors. *SITL (Software in the Loop) — Betaflight Documentation*. https://betaflight.com/docs/development/SITL. Accessed: 2025-08-26. 2025.

[36] Betaflight Community. *Betaflight*. https://betaflight.com/. Open-source flight controller firmware widely used for FPV quadcopters; accessed August 22, 2025. 2025.

[37] Betaflight Team. *Modes — Betaflight Documentation: Airmode*. Accessed: 2025-08-26. 2025. URL: https://betaflight.com/docs/development/Modes#airmode.

[38] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. Miami, Florida, USA, June 2009, pp. 248–255.

[39] Andrew Howard et al. "Searching for MobileNetV3". In: *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE. Seoul, South Korea, Oct. 2019, pp. 1314–1324.

[40] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG]. URL: https://arxiv.org/abs/1905.11946.

[41] United Nations. *Goal 9: Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation*. https://sdgs.un.org/goals/goal9. Accessed: 2025-08-26. 2025.

[42] United Nations. *Goal 11: Make cities and human settlements inclusive, safe, resilient and sustainable*. https://sdgs.un.org/goals/goal11. Accessed: 2025-08-26. 2025.

Annexes

Alignment with the Sustainable Development Goals (SDGs)

This project aligns with Goal 9 of the United Nation's SDGs: Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation [41], as it contributes to the development of advanced, cost-effective technologies that can improve infrastructure and promote innovation in various industries.

It also aligns with goal 11: Make cities and human settlements inclusive, safe, resilient and sustainable [42], as it promotes the creation of technology that can one day help with tasks such as infrastructure inspection, urban safety, or emergency response in a sustainable and cost-effective manner, without relying on current resource-heavy methods.