Implementación Propia de PPO-C y Análisis Experimental en el Entorno FlappyBirdGymnasium-v0

Antonio Santiago Tepsich
Universidad de San Andrés
Buenos Aires, Argentina
atepsich@udesa.edu.ar

Michelle Chloe Berezovsky
Universidad de San Andrés
Buenos Aires, Argentina
mberezovsky@udesa.edu.ar

***Abstract*—Este trabajo presenta la implementación desde cero del algoritmo Proximal Policy Optimization con *clipping* (PPO-C) y su aplicación al entorno `FlappyBirdGymnasium-v0`. El objetivo fue desarrollar de manera manual el pipeline completo de aprendizaje por refuerzo, incluyendo la arquitectura actor-critic, la normalización de ventajas, el mecanismo de clipping y la actualización en minibatches, sin utilizar librerías de RL previamente programadas.**

La evaluación del agente se realizó mediante herramientas propias, análisis de métricas en TensorBoard y un conjunto de entrenamientos multiseed que permitieron estudiar la variabilidad inherente del algoritmo. Además, como paso de validación, se comparó la implementación propia con la versión de PPO de la librería Stable Baselines 3 sobre el entorno `CartPole-v1`, y se evaluó el desempeño del agente en entornos de prueba adicionales como `Acrobot-v1`, con el objetivo de comprobar la corrección y robustez del método más allá del dominio principal de Flappy Bird.

Los resultados muestran diferencias marcadas entre el desempeño estocástico, optimizado durante el entrenamiento, y el desempeño determinista, más estable al momento de ejecutar políticas aprendidas en entornos frágiles. El análisis cuantitativo final permitió identificar configuraciones robustas y distinguir entre modelos adecuados para evaluación algorítmica y aquellos que producen los mejores comportamientos observables. En conjunto, el trabajo demuestra la correcta implementación y funcionamiento de PPO-C, así como su capacidad para adaptarse tanto a entornos estándar de control clásico como a un entorno de control particularmente desafiante.

I. Objetivos

El objetivo general de este trabajo es desarrollar, entrenar y evaluar un agente inteligente capaz de resolver el entorno `FlappyBirdGymnasium-v0` mediante una implementación propia del algoritmo Proximal Policy Optimization [1] en su variante con clipping (PPO-C), sin utilizar librerías de aprendizaje por refuerzo previamente programadas.

De este objetivo principal se desprenden los siguientes objetivos específicos:

- Implementar desde cero una arquitectura actor-critic capaz de aprender políticas estocásticas parametrizadas y un valor estimado del estado.
- Programar el ciclo completo de PPO-C, incluyendo el cálculo de retornos, la estimación de ventajas, la actualización por clipping, la penalización por entropía y el entrenamiento en minibatches.
- Integrar correctamente la interacción entre la política implementada y el entorno `FlappyBirdGymnasium-v0`, comprendiendo su dinámica, recompensas y desafíos particulares.
- Realizar experimentación sistemática con hiperparámetros, buscando maximizar la estabilidad del entrenamiento y la calidad de la política resultante.
- Desarrollar métricas y scripts de evaluación que permitan comparar agentes entrenados bajo diferentes configuraciones, tanto bajo políticas estocásticas como deterministas.
- Analizar los comportamientos aprendidos por el agente, identificar limitaciones del algoritmo en este entorno y documentar los problemas de implementación y dead-ends surgidos durante el proceso.

El éxito del proyecto se evalúa en función de la capacidad del agente para superar consistentemente los obstáculos del entorno, así como en la solidez experimental y claridad metodológica del desarrollo.

II. Introducción

A partir de los objetivos planteados en la sección anterior, este trabajo se inserta en el marco del Aprendizaje por Refuerzo (RL), que permite entrenar agentes capaces de tomar decisiones secuenciales en interacción con un entorno. En este proyecto en particular, el foco se encuentra en el estudio e implementación desde cero de Proximal Policy Optimization con *clipping* (PPO-C).

Para poner a prueba el funcionamiento y estabilidad de la implementación, se seleccionó el entorno `FlappyBirdGymnasium-v0`, una adaptación del clásico juego *Flappy Bird* bajo la interfaz Gymnasium. Este entorno es particularmente sensible, los episodios son breves, el más mínimo error de política termina el juego, y la señal de recompensa es escasa y altamente dependiente de decisiones individuales. Estas características lo convierten en un caso de estudio para analizar el comportamiento de algoritmos actor-critic en entornos frágiles, donde la exploración es limitada y

la variación en las decisiones de la política estocástica puede tener grandes efectos en el desempeño.

En las siguientes secciones se detalla el proceso completo de desarrollo. Primero, se explica la implementación del algoritmo PPO-C, incluyendo las consideraciones técnicas y decisiones de diseño tomadas. Luego, se describe la dinámica del entorno y cómo esta afecta la estrategia de aprendizaje. Posteriormente, se presenta una reconstrucción cronológica del trabajo realizado, destacando problemas de implementación, decisiones metodológicas, configuraciones de entrenamiento y experimentación con hiperparámetros. Finalmente, se discute la evaluación cuantitativa del agente mediante métricas sistemáticas y la comparación entre políticas estocásticas y deterministas, seguida de las conclusiones y posibles líneas de trabajo futuro.

III. Desarrollo

A. Implementación del algoritmo PPO-C

La primera etapa del proyecto consistió en la implementación completa desde cero de un agente basado en Proximal Policy Optimization con *clipping* (PPO-C). La lógica principal del algoritmo se concentra en el módulo `ppo.py`, mientras que el bucle de entrenamiento vectorizado y la interacción con el entorno se encuentran en `train_vector_improved.py`. En esta sección se describen las decisiones de diseño más relevantes.

1) Arquitectura actor–critic: El agente se modela mediante una arquitectura actor-critic compartida, implementada en la clase `VectorActorCritic`. Esta red toma como entrada un vector de observaciones de dimensión 180 (salida preprocesada del entorno `FlappyBird-v0`) y produce, por un lado, los logits de una distribución categórica sobre las dos acciones posibles (`no-op` y `flap`) y, por otro, una estimación escalar del valor del estado $V_\phi(s)$.

La arquitectura consta de dos capas lineales completamente conectadas con activaciones ReLU, compartidas entre actor y critic, seguidas de dos cabezas independientes: una para la política (capa lineal que produce logits) y otra para el valor (capa lineal que produce un escalar). Esta estructura balancea capacidad de representación y simplicidad, reduciendo el riesgo de sobreajuste en un entorno con señal de recompensa escasa y episodios breves.

2) Entorno vectorizado y normalización de observaciones: El entrenamiento se realiza sobre un conjunto de entornos paralelos gestionados por la clase `VecEnv`. Esta clase instancia múltiples copias independientes de `FlappyBird-v0` (por defecto, 16 entornos) y una interfaz de tipo vectorizado para `reset` y `step`. Cada llamada a `step` devuelve un batch de observaciones, recompensas y señales de finalización, lo cual permite recolectar trayectorias más rápidamente y estabilizar las estimaciones de ventaja.

Adicionalmente, se implementa un `ObservationWrapper` denominado `NormalizeObservation` que mantiene una estimación online de la media y la desviación estándar de las observaciones, aplicando una normalización tipo

$$\hat{s} = \frac{s - \mu}{\sigma + \epsilon}.$$

Esta normalización puede activarse o desactivarse mediante el parámetro `normalize_obs`. En la práctica, resultó fundamental para evitar escalas muy dispares entre componentes de la observación y mejorar la estabilidad del entrenamiento.

3) Clase `PPODiagnostic`: núcleo del algoritmo: El corazón del algoritmo se encuentra en la clase `PPODiagnostic`, que implementa la actualización PPO-C y, al mismo tiempo, registra métricas de diagnóstico detalladas. Esta clase recibe el modelo actor–critic y los hiperparámetros principales: tasa de aprendizaje, ϵ de *clipping*, coeficientes de entropía y valor (`ent_coef`, `vf_coef`) y el límite de norma de gradiente (`max_grad_norm`).

En cada actualización, `PPODiagnostic` recibe los tensores de observaciones, acciones, retornos, valores antiguos, log-probabilidades antiguas y ventajas. Las ventajas se normalizan por batch para controlar su escala:

$$A_t^{\text{norm}} = \frac{A_t - \mu_A}{\sigma_A + \epsilon}.$$

A partir de esto, se construye el cociente de probabilidad

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)},$$

y se optimiza el objetivo PPO con *clipping*:

$$L^{\text{clip}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t^{\text{norm}},\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t^{\text{norm}} \right) \right].$$

En paralelo, el critic se actualiza minimizando un error cuadrático medio con *clipping* en el valor:

$$L_V(\phi) = \mathbb{E}_t \left[\max\left((V_\phi(s_t) - G_t)^2,\ (V_{\text{clip}}(s_t) - G_t)^2 \right) \right],$$

donde G_t son los *returns* descontados y V_{clip} limita el cambio del valor respecto a la estimación anterior.

El término de entropía $H(\pi_\theta(\cdot \mid s_t))$ se incorpora con un coeficiente que puede variar en el tiempo, incentivando la exploración al inicio del entrenamiento y permitiendo una política más concentrada en fases avanzadas.

4) Manejo de minibatches y épocas: El método de actualización recorre los datos recolectados en el rollout mediante múltiples épocas de gradiente estocástico. Para cada época, se barajan los índices y se forman minibatches de tamaño configurable. En cada minibatch se recalculan las log-probabilidades actuales, el ratio $r_t(\theta)$ y las pérdidas de política y valor, aplicando luego `backpropagation` y `gradient clipping`. Esta estrategia permite aprovechar al máximo cada interacción con el entorno, que es costosa, sin romper la hipótesis de cercanía entre π_θ y $\pi_{\theta_{\text{old}}}$ gracias al *clipping*.

5) Métricas de diagnóstico y estabilidad del entrenamiento: Una característica diferencial de `PPODiagnostic` es el registro detallado de métricas internas de PPO, entre ellas:

- `policy_loss`, `value_loss` y `entropy`, para monitorear el equilibrio entre optimización de la política, ajuste del critic y nivel de exploración.

- `kl_div` y `approx_kl`, que miden el cambio entre la política nueva y la antigua.
- `clip_frac`, fracción de muestras en las que el ratio fue recortado por el *clipping*.
- `ratio_mean` y `ratio_std`, que describen la distribución del cociente $r_t(\theta)$.
- `explained_var`, varianza explicada por el critic sobre los returns, como indicador de la calidad de la función de valor.

Estas métricas se promedian y se reportan en cada iteración de actualización, tanto por consola como a través de un logger integrado con TensorBoard. A partir de ellas se realizaron diagnósticos sobre colapso de entropía, insuficiente capacidad del critic, *overfitting* a trayectorias específicas y magnitud excesiva de las actualizaciones, temas que se desarrollan más adelante en la sección de historia del desarrollo y experimentación.

B. Entorno `FlappyBirdGymnasium-v0`

El agente se entrena en el entorno `FlappyBird-v0` provisto por la librería `flappy_bird_gymnasium` [2], una implementación del juego *Flappy Bird* compatible con Gymnasium. Este entorno modela un escenario de control continuo en tiempo discreto, donde un pájaro debe atravesar una secuencia infinita de tuberías evitando colisionar con ellas o con los límites superior e inferior de la pantalla.

1) Espacio de estados: En este proyecto se utilizó la variante del entorno basada en un sensor tipo *LIDAR*, que devuelve un vector de 180 observaciones numéricas por estado. Cada componente de este vector representa una lectura angular del "entorno visual" frente al pájaro, codificando la distancia relativa a obstáculos y límites de la pantalla [2].

Desde el punto de vista del agente, el estado en cada instante t puede verse como:

$$s_t \in \mathbb{R}^{180},$$

donde cada entrada describe información geométrica de la escena (posición relativa de tuberías, huecos y límites). Esta representación permite prescindir de imágenes crudas y reducir el problema a un espacio continuo de dimensión moderada, lo que hace viable el uso de una red fully-connected como arquitectura actor–critic, tal como se describe en la subsección anterior.

Antes de alimentar las observaciones a la red, se aplica una normalización online mediante el wrapper `NormalizeObservation`. Este wrapper mantiene estimaciones móviles de media y varianza y transforma cada observación según:

$$\hat{s}_t = \frac{s_t - \mu_t}{\sigma_t + \varepsilon},$$

con el objetivo de estabilizar la escala de entrada a la red y mejorar la eficiencia del entrenamiento, especialmente en presencia de componentes de estado con magnitudes muy dispares.

2) Espacio de acciones: El espacio de acciones es discreto y de muy baja dimensión:

$$\mathcal{A} = \{0, 1\},$$

donde:

- $a = 0$: no hacer nada (*no-op*), el pájaro continúa cayendo por efecto de la gravedad.
- $a = 1$: realizar un aleteo (*flap*), que incrementa momentáneamente la altura del pájaro.

La simplicidad del espacio de acciones contrasta con la dificultad del entorno, ya que pequeñas variaciones en la secuencia de decisiones pueden conducir rápidamente a una colisión, lo que hace que el problema sea particularmente sensible a errores de política.

3) Recompensas y finalización de episodios: El esquema de recompensas definido por el entorno `FlappyBird-v0` es el siguiente [2]:

- $+0.1$ por cada *frame* en el que el agente se mantiene con vida.
- $+1.0$ por cada tubería atravesada con éxito.
- -1.0 al morir (colisión con una tubería o con el suelo).
- -0.5 al tocar el límite superior de la pantalla.

Un episodio termina cuando ocurre una de las siguientes condiciones:

- El pájaro colisiona con una tubería.
- El pájaro sale de los límites verticales (suelo o techo).
- Se alcanza un límite máximo de duración establecido por el entorno.

Este esquema produce una señal de recompensa escasa y altamente dependiente de secuencias de acciones correctas: la mayor parte del tiempo el agente recibe recompensas pequeñas por sobrevivir y penalizaciones abruptas al fallar. Desde el punto de vista del algoritmo, esto se traduce en ventajas con alta varianza y episodios muy cortos en las primeras fases del entrenamiento, lo que pone a prueba la estabilidad del critic y del mecanismo de *clipping* de PPO-C.

4) Vectorización de entornos y episodios frágiles: Para mitigar la alta varianza inherente al entorno, el entrenamiento se realiza sobre múltiples copias paralelas de `FlappyBird-v0` mediante la clase `VecEnv`. Cada paso de entrenamiento avanza simultáneamente en N entornos (típicamente $N = 16$), recolectando una mayor cantidad de experiencias por actualización y promediando la información sobre episodios con distinta duración y resultado.

Esta combinación de:

- episodios frágiles y de corta duración,
- recompensas escasas y desbalanceadas,
- y un espacio de estados continuo de dimensión 180,

convierte a `FlappyBirdGymnasium-v0` en un banco de pruebas exigente para algoritmos actor–critic. En las secciones siguientes se detalla cómo estas características impactaron en la implementación de PPO-C, en el diseño de la experimentación y en los problemas concretos que surgieron durante el desarrollo.

C. Historia del desarrollo, problemas encontrados y decisiones tomadas

El desarrollo del proyecto siguió un proceso iterativo en el que se combinaron fases de implementación, experimentación, análisis de métricas y corrección de problemas prácticos surgidos durante las distintas pruebas. A continuación se presenta la reconstrucción precisa del proceso real seguido durante el trabajo, basada en el código desarrollado, los experimentos registrados y los intercambios documentados durante el proyecto.

1) Fase inicial: armado del algoritmo y del entorno vectorizado: El punto de partida fue la implementación manual del algoritmo PPO-C y la estructura básica del entrenamiento vectorizado. Se desarrolló la clase `VectorActorCritic` para modelar la arquitectura actor–critic y la clase `PPODiagnostic` para ejecutar la actualización del algoritmo, calcular métricas internas y registrar diagnósticos.

En paralelo se diseñó `VecEnv`, un entorno vectorizado que permite ejecutar múltiples copias independientes de `FlappyBird-v0`. Esta vectorización fue esencial para aumentar la eficiencia de recolección de experiencias y para reducir la varianza de las métricas. Durante esta etapa, el foco estuvo puesto en confirmar la coherencia dimensional entre observaciones, acciones, log-probabilidades y valores estimados, así como en verificar el correcto funcionamiento del *forward pass* de la red.

2) Primeros entrenamientos y validación funcional: Una vez armado el pipeline básico, se ejecutaron los primeros entrenamientos prolongados para verificar que:

- la política podía interactuar correctamente con el entorno,
- los rollouts contenían la información necesaria para PPO,
- las métricas internas (como pérdidas, entropía y KL) evolucionaban de manera consistente,
- el agente mostraba alguna mejora observable respecto al comportamiento aleatorio inicial.

Las primeras pruebas permitieron confirmar que la implementación del algoritmo funcionaba correctamente y que la política aprendía comportamientos básicos de supervivencia, aunque de manera todavía inestable.

3) Problema 1: registro incorrecto de experimentos en TensorBoard: Uno de los primeros problemas significativos surgió durante la fase de análisis de resultados. Se observó que TensorBoard mostraba decenas de curvas superpuestas en un mismo gráfico, aun cuando solo se había ejecutado un entrenamiento específico. Este comportamiento se debía a que múltiples ejecuciones estaban escribiendo en la misma carpeta `runs/`, lo que mezclaba todos los experimentos en un único conjunto de logs.

Aunque esto no afectaba el entrenamiento en sí mismo, complicaba severamente la interpretación de las métricas y dificultaba la comparación entre configuraciones.

La solución consistió en:

- generar un directorio único por experimento mediante nombres con timestamps,
- limpiar periódicamente la carpeta de logs,
- ajustar los scripts de entrenamiento para garantizar la separación de corridas.

A partir de este cambio, la visualización y el análisis de métricas se volvió consistente y confiable.

4) Fase 2: búsqueda exploratoria de hiperparámetros: Una vez establecida la implementación funcional del algoritmo y verificado el correcto desempeño del entorno vectorizado, la siguiente etapa consistió en realizar una búsqueda exploratoria de hiperparámetros con el objetivo de identificar regiones prometedoras del espacio de configuración. En esta fase se empleó un enfoque deliberadamente disperso, probando combinaciones que cubrieran rangos amplios de valores antes de proceder a una optimización más fina.

La búsqueda se ejecutó mediante el módulo `run_experiment.py`, configurado a través del archivo `phase1_quick_search.yaml`, el cual define un conjunto heterogéneo de opciones para parámetros clave: número de entornos vectorizados, tamaño del rollout, tasas de aprendizaje inicial y final, tipo de scheduler, tamaño de la red y coeficientes de entropía y valor. El objetivo fue explorar escalas contrastantes, por ejemplo tasas de aprendizaje desde 10^{-5} hasta 10^{-3}, con el fin de obtener una primera cartografía del comportamiento del algoritmo.

Esta fase permitió identificar rápidamente configuraciones inestables, como pérdidas de entropía, desviaciones abruptas del critic o valores explosivos de KL. Al mismo tiempo, permitió detectar combinaciones que mostraban un aprendizaje inicial más consistente. La Figura 1 muestra la evolución de la recompensa promedio por actualización para algunas de las configuraciones evaluadas. La dispersión observada en las curvas, donde algunas configuraciones divergen rápidamente mientras que otras exhiben progreso estable, motivó concentrar la fase siguiente de búsqueda en torno a las regiones más prometedoras del espacio de hiperparámetros.

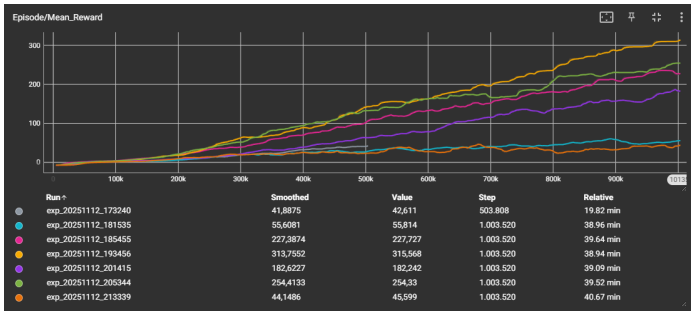

Fig. 1: Recompensa promedio por actualización en la búsqueda exploratoria de hiperparámetros. Cada curva corresponde a una configuración distinta definida en `phase1_quick_search.yaml`

5) Fase 3: búsqueda focalizada alrededor de las mejores configuraciones: A partir de los resultados de la búsqueda exploratoria, se identificaron varias configuraciones que mostraban un comportamiento estable y con crecimiento sostenido en la recompensa promedio durante el entrenamiento. Sobre esta base se diseñaron

dos búsquedas adicionales más focalizadas, definidas en los archivos `phase2_1_focused_search.yaml` y `phase2_2_focused_search.yaml`. En ambos casos se restringieron los valores de hiperparámetros a un vecindario cercano a las mejores configuraciones de la fase anterior.

Las Figuras 2 y 3 muestran la recompensa promedio a lo largo del entrenamiento para ambas búsquedas focalizadas. En términos de estabilidad, las curvas presentan un crecimiento progresivo y sin colapsos abruptos, lo que indica que todas las configuraciones probadas se encontraban en una región razonable del espacio de hiperparámetros. Sin embargo, el comportamiento de estas curvas no anticipó una mejora concreta en la capacidad final de los modelos de obtener episodios largos al ser evaluados fuera de entrenamiento.

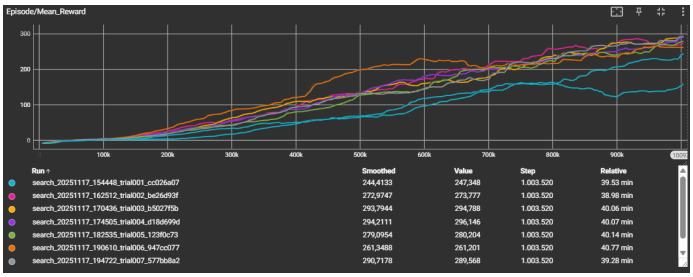

Fig. 2: Recompensa promedio durante el entrenamiento para la primera búsqueda focalizada.

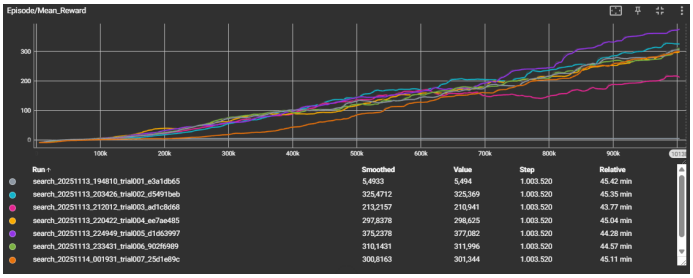

Fig. 3: Recompensa promedio durante el entrenamiento para la segunda búsqueda focalizada.

6) Problema 2: ausencia de mejoras a pesar de la búsqueda focalizada: Un resultado relevante de esta etapa es que, aunque las dos búsquedas focalizadas mostraron curvas de entrenamiento con recompensas crecientes y estables, este comportamiento no se reflejó en las evaluaciones finales de los modelos. En particular, al ejecutar cada uno de los modelos entrenados durante tres episodios independientes, los resultados obtenidos no superaron de manera sistemática al mejor modelo de la fase anterior (modelo base que usamos para buscar nuevos hiperparametros).

Las tablas de evaluación (archivo adjunto en el Drive [3] hoja de "Evaluación por Fases") muestran que algunos modelos lograron episodios individuales con recompensas muy altas, pero estas mejoras no fueron consistentes entre corridas. En promedio, los nuevos modelos presentaron un rendimiento equivalente o inferior al del mejor modelo inicial, lo que indica que las variaciones de hiperparámetros consideradas no lograron desplazar al algoritmo hacia una región más favorable del espacio de configuración.

Este desfasaje entre la recompensa promedio durante el entrenamiento y el rendimiento en las evaluaciones llevó a la necesidad de definir un criterio de ranking que sintetizara múltiples métricas. Finalmente se adoptó una combinación de:

- promedio de puntuación,
- máximo obtenido,
- variabilidad de episodios,
- desempeño estocástico vs. determinista.

Este criterio permitió seleccionar modelos robustos y evitar aquellos con desempeños engañosamente altos producto de un único episodio atípico.

7) Problema 3: el modelo con mejor recompensa en entrenamiento no era el mejor jugando: Durante la evaluación manual de distintos modelos, realizada mediante el script `watch_play.py`, se detectó un comportamiento repetido: el modelo que obtenía la mayor recompensa promedio durante el entrenamiento no era necesariamente el que mejor jugaba.

Este hallazgo motivó una serie de análisis adicionales que permitieron identificar la causa:

- durante el entrenamiento, la política se comporta de manera estocástica y optimiza el rendimiento bajo esa distribución;
- durante la evaluación manual, la política se ejecutaba usando tanto `argmax` sobre la distribución aprendida, es decir, de manera determinista, como así también de forma estocástica;
- estas dos dinámicas no siempre se alinean en entornos frágiles como Flappy Bird, donde una acción ligeramente subóptima puede terminar el episodio.

Como resultado, se concluyó que era necesario evaluar modelos tanto en modo estocástico como en modo determinista para obtener una medición más completa de su desempeño.

8) Implementación de herramientas específicas de evaluación: Para enfrentar los problemas anteriores, se desarrollaron herramientas específicas:

- una versión ampliada de `watch_play.py` con el parámetro `policy_mode`, permitiendo evaluar políticas en modo estocástico y determinista;
- el script `evaluate.py`, que automatiza la evaluación de múltiples modelos, calcula métricas como promedio, máximo y desvío estándar, y registra resultados en archivos CSV;
- la integración de estos resultados con un bucket en Google Cloud Storage para conservar todas las evaluaciones.

Estas herramientas permitieron construir un criterio sistemático para comparar agentes y fundamentar la selección del mejor modelo.

9) Fase 4: Analisis Profundo de los mejores resultados: Luego de estabilizar el pipeline de entrenamiento, aumentar la claridad del logging y desarrollar herramientas sólidas de evaluación, el proyecto llegó a una fase madura. En esta

etapa, con las métricas guardadas en el spreadsheet adjunto en el Drive (Hoja 2, `All evaluations`), se procede a un análisis más detallado.

Dado que existe una brecha importante entre los dos mejores modelos y el resto, se realiza un análisis profundo de ellos dos y se incluye en la tabla también el tercer modelo del ranking para mostrar la diferencia entre los primeros y el resto.

Métrica	**Modelos**		
	`a770b73c`	`exp205344`	`25d1e89c`
Stoch Avg	213.9	196.2	181.6
Stoch Max	753	597	626
Stoch Std	217.59	177.58	168.11
Stoch W	278.32	247.02	235.79
Det Avg	670.5	741.1	388.1
Det Max	3337	4970	847
Det Std	997.99	1443.80	281.53
Det W	954.30	1165.91	458.72

TABLE I: Comparación compacta de los tres mejores modelos evaluados en modo estocástico y determinístico.

La primera observación es que no hay un único ganador absoluto: el modelo `a770b73c` obtiene los mejores resultados en la política estocástica (la que PPO optimiza durante el entrenamiento), mientras que `exp205344` alcanza las mejores métricas en la política determinística (acción por `argmax`). El modelo `25d1e89c` aparece mucho más abajo en ambas métricas, lo que justifica concentrar el análisis en los dos primeros.

a) Evolución del rendimiento en entrenamiento: En la Figura 4 se muestra la curva de `Game/Mean_Pipes_Passed` para ambos modelos. Las dos corridas presentan una mejora monotónica y suave, sin oscilaciones fuertes, lo que indica un entrenamiento estable. Sin embargo, la curva de `a770b73c` se mantiene sistemáticamente por encima de `exp205344` y sigue creciendo hasta el final del entrenamiento, mientras que `exp205344` tiende a estabilizarse antes. Esto sugiere que, desde el punto de vista de la política estocástica, `a770b73c` logra una convergencia más sólida.

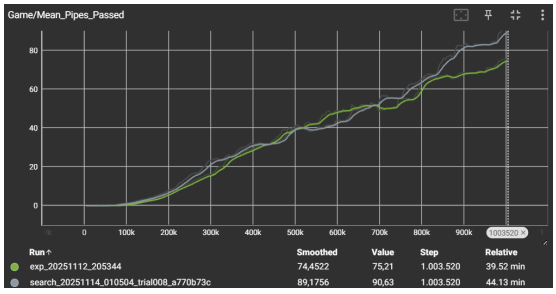

Fig. 4: Evolución del score medio (`Game/Mean_Pipes_Passed`) para los modelos `a770b73c` y `exp205344`.

b) Comportamiento de la política y del crítico: La Figura 5 resume los diagnósticos de PPO: `Policy_Loss`, `Value_Loss`, `Entropy` y `KL_Divergence`. En `Policy_Loss` se observa que `a770b73c` converge hacia una banda más estable alrededor de cero, con menos oscilaciones que `exp205344`. Algo similar ocurre en `Value_Loss`: el crítico de `a770b73c` termina con pérdidas más bajas y menos picos tardíos, lo que indica ventajas menos ruidosas y un valor mejor calibrado.

En cuanto a la entropía, ambos modelos descienden rápidamente desde valores altos hasta una franja baja (en torno a 0.02–0.03), pero `exp205344` pierde exploración aún más rápido. Finalmente, el `KL_Divergence` de `a770b73c` es más suave y decreciente, mientras que `exp205344` presenta picos más marcados. En conjunto, estos gráficos señalan que `a770b73c` tiene una dinámica de PPO más estable, con actualizaciones más controladas y un crítico mejor entrenado.

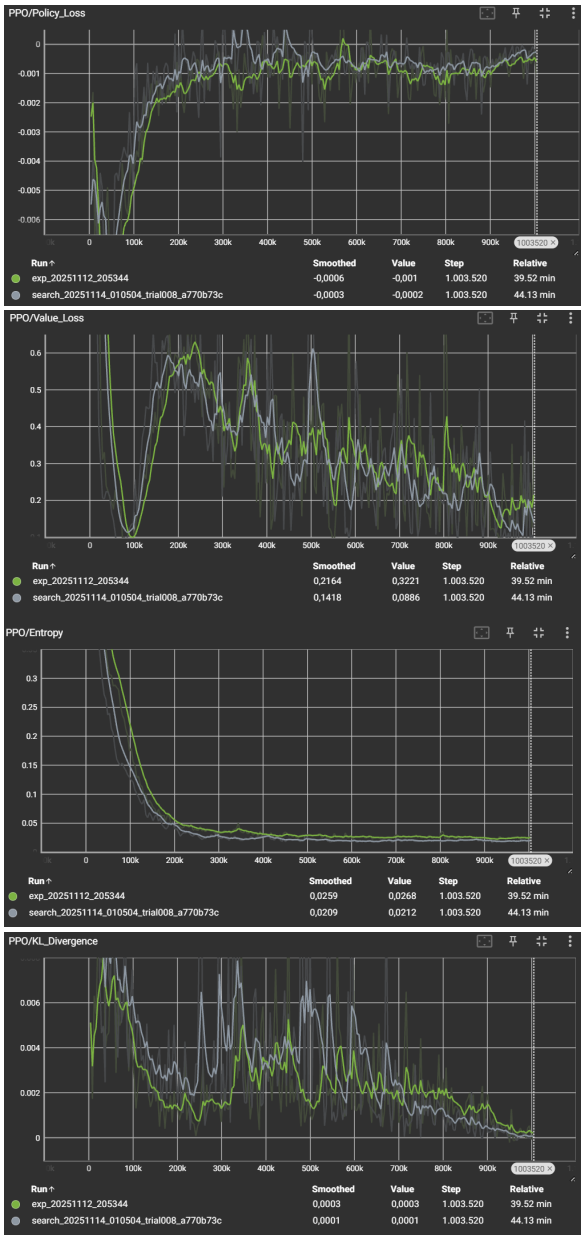

Fig. 5: Diagnósticos de PPO para los dos mejores modelos: pérdidas de política y valor, entropía y divergencia KL.

Desde la perspectiva de *aprendizaje de la política*, el modelo `a770b73c` es el mejor candidato ya que obtiene el mayor score estocástico, ademas presenta un crítico más estable y las curvas de PPO muestran una convergencia más limpia. Por eso se lo toma como referencia principal para seguir refinando la búsqueda de hiperparámetros.

Por otro lado, el modelo `exp205344` demuestra un desempeño determinístico notablemente superior (mayor `Det Avg` y `Det Max`), lo que lo convierte en un excelente candidato para despliegue o demostraciones donde la política se ejecuta en modo `argmax`.

10) Fase 5: Evaluación Multiseed: Con el objetivo de verificar la robustez de la mejor configuración de hiperparámetros obtenida en la búsqueda global (Fase 4), se llevó a cabo una evaluación *multiseed*. Este procedimiento consiste en reentrenar el agente varias veces manteniendo exactamente los mismos hiperparámetros, pero variando únicamente la semilla aleatoria utilizada en la inicialización de pesos y en los primeros rollouts.

En algoritmos de aprendizaje por refuerzo basados en políticas estocásticas, como PPO, la sensibilidad a la semilla es un aspecto relevante, ya que según la exploración inicial, el algoritmo puede converger hacia regiones distintas del espacio de políticas. Por ello, un conjunto de entrenamientos con diferentes semillas permite distinguir entre una configuración de hiperparámetros realmente estable y un resultado aislado producto de una combinación afortunada de inicialización y muestreo.

Para esta fase se empleó la configuración final seleccionada en la Fase 4, correspondiente al experimento `a770b73c`, considerada la de mayor estabilidad según las métricas estocásticas (*stochastic weighted* y *stochastic average*). Se realizaron cinco entrenamientos independientes variando únicamente el parámetro `seed`.

La Figura 6 muestra el *final mean reward* alcanzado por cada una de las cinco ejecuciones. Puede observarse que:

- Cuatro de las cinco ejecuciones convergieron satisfactoriamente, alcanzando valores finales en el rango aproximado de 310 a 385 puntos, consistente con los rendimientos esperados para esta configuración.
- Una de las ejecuciones obtuvo un valor final sustancialmente inferior ($\sim$ 52), evidenciando un caso de no convergencia atribuible a ruido estocástico, exploración inicial desfavorable o un colapso temprano de la política.

La presencia de un único *seed* fallido en cinco ejecuciones (**80%** de éxito) la consideramos evidencia sólida de que la configuración de hiperparámetros es robusta y estable. En consecuencia, la Fase 5 confirma la validez de la configuración final, la cual se adopta como **baseline definitiva** (`flappy_ppo_v1`) para posteriores experimentos y análisis comparativos.

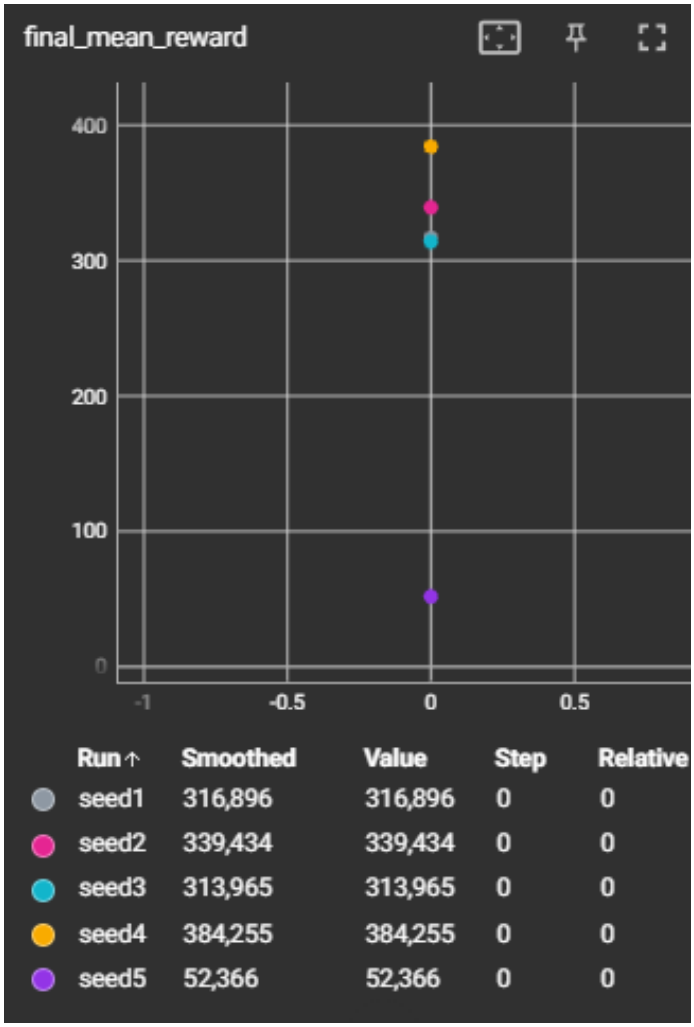

Fig. 6: Recompensa media final obtenida para cada semilla durante la evaluación multiseed. Se evidencia una alta consistencia entre cuatro de las cinco ejecuciones, con un único caso de no convergencia.

IV. Resultados en entornos de prueba y baseline en CartPole

Además del análisis detallado en `FlappyBirdGymnasium-v0`, se evaluó la implementación propia de PPO-C en los entornos de prueba provistos por la cátedra. En particular, se utilizó `CartPole-v1` como entorno de referencia para validar la implementación frente a una versión estándar del algoritmo.

A. Comparación con PPO de Stable Baselines 3 en CartPole-v1

Para validar la corrección de la implementación, se comparó el desempeño de PPO-C propio con la implementación de PPO de la librería Stable Baselines 3 (SB3) sobre el entorno `CartPole-v1`. En ambos casos se utilizó una arquitectura *MLP* con tres capas ocultas de 128 neuronas con activación ReLU, y se entrenó durante un total de $T = 204{,}800$ *timesteps*.

Se entrenaron 5 modelos independientes por algoritmo, cada uno con una semilla distinta. Al finalizar el entrenamiento, cada política se evaluó de forma determinista en 20 episodios de prueba, registrando la recompensa total por episodio. En la Tabla II se reporta el retorno medio y desvío estándar sobre todas las semillas y episodios de evaluación.

TABLE II: Desempeño en `CartPole-v1`: comparación entre PPO-C propio y PPO (SB3).

Algoritmo	**Retorno medio $\pm$ desvío estándar**
PPO-C (implementación propia)	481.3 ± 49.7
PPO (Stable Baselines 3)	500.0 ± 0.0

La Figura 7 muestra la comparación gráfica de ambos algoritmos a partir de la figura `cartpole_ppoc_vs_sb3.png` generada con `matplotlib`. Se observa que la implementación de referencia alcanza de manera consistente el máximo retorno posible en `CartPole-v1` (500 puntos por episodio), con varianza prácticamente nula entre semillas. La implementación propia de PPO-C se aproxima a este rendimiento óptimo, con un retorno medio ligeramente inferior y una mayor dispersión entre corridas.

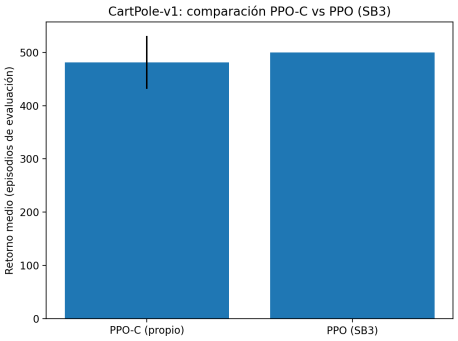

Fig. 7: Retorno medio de evaluación en `CartPole-v1` para PPO-C (propio) y PPO (SB3). Las barras representan la media y las líneas verticales el desvío estándar sobre 5 semillas y 20 episodios de evaluación por semilla.

Estos resultados sugieren que la implementación propia reproduce de manera razonable el comportamiento esperado de PPO en un entorno simple y bien estudiado, aunque con una estabilidad ligeramente menor que la implementación madura de SB3. Esta validación sirve como punto de partida para confiar en la implementación de PPO-C antes de aplicarla al entorno de Flappy Bird y al resto de los entornos de prueba considerados en este trabajo.

B. Resultados en `Acrobot-v1`

Como parte de los *probe environments* utilizados para validar la robustez del algoritmo, se evaluó PPO-C en el entorno `Acrobot-v1`. Este entorno representa un sistema de doble péndulo actuado en el segundo eslabón, donde el objetivo es generar suficiente impulso para elevar el extremo superior del péndulo por encima de cierta altura. A diferencia de `CartPole-v1`, la señal de recompensa es escasa, los episodios son más breves y la dinámica es altamente no lineal, lo que lo convierte en un entorno significativamente más desafiante para algoritmos basados en políticas.

Se entrenaron cinco modelos independientes variando únicamente la semilla. Tras el entrenamiento, cada política se evaluó de manera determinista en 20 episodios. Los resultados obtenidos fueron:

TABLE III: Desempeño de PPO-C en `Acrobot-v1`.

Entorno	**Retorno medio**	**Desvío estándar**
`Acrobot-v1`	-415.6	168.8

El desempeño obtenido es coherente con lo esperado para este entorno, donde las recompensas negativas dominan la dinámica y pequeñas diferencias en la política pueden conducir a colapsos tempranos del episodio. La alta varianza entre semillas refleja esta sensibilidad, pero aun así los resultados confirman que la implementación es capaz de aprender políticas razonables en un entorno de control clásico no trivial.

V. Conclusiones y trabajo futuro

El proyecto logró cumplir con el objetivo principal: implementar desde cero el algoritmo PPO-C, integrarlo con el entorno `FlappyBirdGymnasium-v0` y analizar rigurosamente su comportamiento mediante herramientas de diagnóstico, entrenamiento vectorizado y evaluaciones sistemáticas.

Como paso de validación adicional, requerido por la consigna, el algoritmo también se evaluó en entornos estándar de control clásico. En particular, la comparación con PPO de Stable Baselines 3 en `CartPole-v1` mostró que la implementación propia reproduce el comportamiento esperado del algoritmo, alcanzando un rendimiento cercano al óptimo e identificando diferencias en estabilidad entre semillas. Asimismo, la evaluación en `Acrobot-v1` permitió estudiar el desempeño de PPO-C en un entorno más desafiante, con recompensas escasas y dinámica no lineal, confirmando la robustez de la implementación y su capacidad para generalizar más allá del entorno principal de Flappy Bird.

El proceso permitió comprender en profundidad la dinámica del algoritmo, desde la construcción del surrogate con clipping hasta el papel de la normalización de ventajas, la penalización por entropía y el valor crítico del critic en entornos con recompensas escasas. Asimismo, el uso de un entorno frágil como *Flappy Bird* expuso de manera explícita la sensibilidad del agente a pequeñas variaciones en la política, lo cual se vio reflejado tanto en el entrenamiento como en la evaluación.

Otra contribución significativa del proyecto fue la realización de evaluaciones multiseed como fase final. Al ejecutar la misma configuración bajo distintas semillas, se observaron diferencias moderadas pero consistentes en episodios, estabilidad y velocidad de convergencia, lo cual confirmó la variabilidad inherente a PPO-C y reforzó la necesidad de análisis estadísticos para obtener conclusiones sólidas. Esta práctica permitió distinguir desempeños reproducibles de resultados dependientes de una semilla particular y mejoró la confiabilidad del análisis final.

La evaluación cuantitativa reveló dos patrones clave: el desempeño estocástico es el indicador más fiel del proceso de optimización del algoritmo, mientras que el desempeño determinista refleja mejor la calidad observable del comportamiento aprendido. Esta dualidad resultó especialmente relevante dado el entorno utilizado: pequeñas diferencias en la acción pueden producir resultados drásticamente distintos. El análisis sistemático de ambas métricas permitió identificar modelos con buen equilibrio entre consistencia y rendimiento máximo, y estableció un criterio claro para seleccionar modelos tanto para análisis como para demostración.

En conjunto, el trabajo evidencia que la implementación desarrollada es correcta, estable y capaz de aprender políticas útiles en un entorno desafiante. Además, la combinación de análisis de TensorBoard, evaluaciones multiseed y experimentación controlada permitió fundamentar las decisiones tomadas durante el desarrollo y construir una comprensión detallada del comportamiento del algoritmo.

Trabajo futuro

Existen varias direcciones para continuar y expandir este trabajo:

- **Entropía adaptativa o annealing programado:** ajustar dinámicamente el coeficiente de entropía podría mitigar tanto la exploración insuficiente inicial como el colapso prematuro de la política en fases tardías.
- **Optimización del critic y clipping del valor:** experimentar con pérdidas alternativas o arquitecturas más profundas podría mejorar la capacidad del critic para modelar correctamente la estructura del retorno.
- **Extender la arquitectura actor–critic hacia entornos más complejos:** aplicar esta implementación a entornos con observaciones más ricas (por ejemplo, basados en imágenes) permitiría evaluar la escalabilidad del método.

Estas líneas de trabajo permitirían no solo confirmar la robustez del algoritmo implementado, sino también explorar variantes y extensiones de PPO en entornos más complejos y desafiantes.

REFERENCES

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[2] M. Kubis, "Flappy Bird Gymnasium" (versión 0), GitHub repository, 2023. Disponible en: https://github.com/markub3327/flappy-bird-gymnasium.

[3] Drive con video y Evaluaciones. Disponible en: https://drive.google.com/drive/folders/1esKJS97e0Ws1jOa9FZXQD6lc1lXqItym.