Universidad de Buenos Aires

Facultad de Ingeniería

Trabajo Práctico:

Aprendizaje por refuerzo con PPO

(Donkey Kong)

Aprendizaje Automático

2° Cuatrimestre 2024

Alumno	**Responsabilidad**	**Padrón**	**E-mail**
Máximo Utrera	Investigación, desarrollo, ajuste, refinamiento y testing	109651	mutrera@fi.uba.ar
Felipe D'alto	Investigación y presentación	110000	fedalto@fi.uba.ar
Tania Friedenberger	Investigación y presentación	108823	tfriedenberger@fi.uba.ar

Fecha de Presentación: 28 de Noviembre de 2024

Índice

1 Introducción

El aprendizaje por refuerzo (Reinforcement Learning, RL), una rama del aprendizaje automático, ha ganado gran relevancia en los últimos años debido a su capacidad para resolver problemas complejos mediante la interacción con entornos dinámicos. A diferencia de otros enfoques, RL permite que un agente aprenda comportamientos óptimos a través de un proceso de prueba y error, utilizando recompensas como guía para maximizar su desempeño.

En este proyecto, se empleó el algoritmo Proximal Policy Optimization (PPO), para entrenar un agente capaz de completar el primer nivel del clásico videojuego Donkey Kong. Este desafío implica que el agente aprenda a superar obstáculos, como barriles rodantes, saltos y escaleras, con el objetivo final de rescatar a la princesa.

Para implementar este proyecto, se utilizó una combinación de herramientas modernas. La simulación del entorno del juego se llevó a cabo en Unity, aprovechando la biblioteca ML-Agents para integrar mecánicas del juego con algoritmos de aprendizaje por refuerzo. La gestión de dependencias y entornos de desarrollo se realizó con Conda, mientras que PyTorch fue la biblioteca principal para desarrollar y entrenar el modelo de aprendizaje por refuerzo. Esta integración permitió construir un flujo de trabajo eficiente para entrenar y evaluar el agente en un entorno de alta fidelidad.

El presente informe documenta el desarrollo del proyecto, desde la definición del problema hasta los resultados obtenidos.

2 Objetivos

El presente trabajo tiene como objetivo principal aplicar técnicas de aprendizaje por refuerzo para entrenar un agente capaz de superar el primer nivel del videojuego 'Donkey Kong'. Para lograrlo, se establecen los siguientes objetivos específicos:

- Diseñar y desarrollar una réplica funcional del videojuego 'Donkey Kong' utilizando Unity, asegurando que el entorno permita la interacción dinámica entre el agente y los elementos del juego.
- Implementar el algoritmo Proximal Policy Optimization (PPO) para entrenar agentes con un enfoque en la estabilidad, eficiencia y versatilidad de los modelos resultantes.
- Analizar y resolver los desafíos técnicos y conceptuales que surjan durante el proceso de desarrollo, a fin de profundizar en la comprensión de las técnicas de simulación y aprendizaje por refuerzo.
- Documentar el flujo de trabajo, las decisiones técnicas y los resultados obtenidos, con el objetivo de proporcionar una guía clara y estructurada sobre el uso de aprendizaje por refuerzo en entornos de videojuegos.

3 Elección del algoritmo a emplear (PPO)

El algoritmo **Proximal Policy Optimization (PPO)** es una técnica moderna de aprendizaje por refuerzo basada en políticas, desarrollada por OpenAI. Es ampliamente reconocido por su equilibrio entre simplicidad, estabilidad y eficiencia, lo que lo convierte en una opción ideal para tareas complejas como videojuegos.

¿Qué es PPO?

PPO optimiza directamente la política del agente mientras interactúa con el entorno, buscando mejorar su desempeño sin realizar cambios bruscos en sus decisiones. Para lograrlo, utiliza una función de pérdida que restringe dichas modificaciones, garantizando un aprendizaje más estable.

El objetivo principal de PPO se expresa como:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\text{mín} \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

Donde:

- $r_t(\theta)$ es el ratio entre la probabilidad de las acciones según la nueva política y la anterior:

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

- $\hat{A}_t$ es la ventaja estimada, que mide cuánto mejor fue una acción en comparación con la expectativa promedio.
- ϵ define un rango de confianza para limitar los cambios en la política.

Este enfoque asegura que el agente pueda mejorar su comportamiento de manera progresiva y controlada, evitando inestabilidades durante el entrenamiento.

Ventajas de PPO

- **Estabilidad:** Al limitar los cambios abruptos en la política del agente, PPO permite un aprendizaje más consistente.
- **Simplicidad:** Comparado con otros algoritmos avanzados, como TRPO, PPO logra resultados similares con menos complejidad computacional.
- **Versatilidad:** PPO es adecuado tanto para tareas con acciones continuas como discretas, lo que lo hace ideal para entornos dinámicos como videojuegos.
- **Eficiencia:** Su implementación eficiente lo hace una opción robusta incluso en simulaciones complejas.

Razones para elegir PPO

Se eligió PPO para este proyecto por su capacidad de manejar entornos dinámicos y complejos como el videojuego *'Donkey Kong'*. Su estabilidad durante el entrenamiento garantiza que el agente pueda aprender estrategias óptimas para superar desafíos, como esquivar barriles y escalar escaleras, sin riesgo de desestabilizarse. Además, su simplicidad y eficiencia lo convierten en una herramienta accesible para implementar en proyectos con recursos modestos, como los disponibles en este trabajo.

En resumen, PPO es la solución ideal para desarrollar un agente que pueda interactuar de manera efectiva con un entorno dinámico, maximizando su rendimiento mientras mantiene la estabilidad del modelo.

4 Desarrollo y entrenamiento

En esta sección documenta la etapa de desarrollo inicial del proyecto así como también su refinamiento iterativo. A su vez se abordan las dificultades surgidas y los diferentes modelos que se probaron y se descartaron hasta llegar al modelo que logró alcanzar los objetivos de este trabajo. La creación del entorno de desarrollo queda demostrada en ***el repositorio del proyecto***.

4.1. Implementación del entorno y mecánicas del juego

Previo a la implementación y entrenamiento de los agentes se creo una versión simplificada del juego 'Donkey Kong' utilizando el motor de videojuegos Unity. Esta replica desarrollada cuenta con las siguientes características y funcionalidades:

- Diseño del primer nivel del juego original
- Mecánica de movimiento horizontal y saltos
- Mecánica de subir y bajar escaleras
- Lanzamiento de barriles con un intervalo randomizado
- Probabilidad de caída de barriles a través de las escaleras
- Finalización y reinicio del nivel al colisionar con un barril, caer del mapa, o alcanzar a la princesa

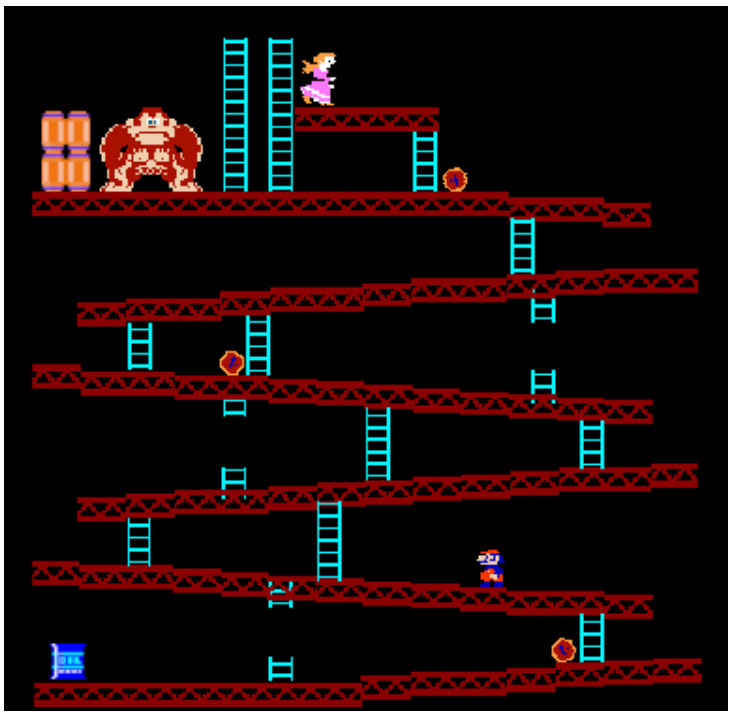

Figura 1: Replica desarrollada.

Luego de desarrollar esta versión básica jugable del videojuego se incluyo la biblioteca Ml-agents la cual permitiría conectar el flujo de juego a un servidor de python que se estaría ejecutando esperando a esa comunicación para poder comenzar el entrenamiento del modelo. Esta comunicación proporciona al 'Entrenador' la información necesaria del entorno para que el modelo siendo entrenado pueda interactuar con este y recibir recompensas o penalidades dependiendo de sus acciones.

4.2. Definición de capas de entrada y salida

Una de las partes mas centrales de la creación de una red neuronal es definir la forma y tamaño de esta. En este caso se debe definir la cantidad de unidades que estarán en la primera capa (la de entrada) ya que estas son los 'ojos' del modelo, es decir, es la forma en la que el modelo obtiene información de su entorno. Y también se debe definir la ultima capa (la de salida), cuyas unidades representan una posible acción que el agente puede llevar a cabo.

En esta sección se han probado varias combinaciones de capas de entrada y salida. Algunos ejemplos respecto a la capa de entrada fueron, agregar/quitar información sobre velocidades de las entidades, sobre distancias verticales, normalización de datos. En el caso de la capa de salida fueron, diferentes ramas para movimientos verticales y horizontales (permitiendo dos acciones concurrentes), permitir o no la posibilidad de elegir no moverse, entre otros.

Las observaciones de la capa de entrada y las acciones de la capa de salida que llevaron a los mejores resultados y con las que se entrenaron los modelos finales se detallan a continuación:

Observaciones [16 unidades]	**Acciones [6 unidades]**
Posición y velocidad del agente	No accionar
Si está en el piso	Mover derecha/izquierda
Si está escalando	Saltar
Distancia al próximo checkpoint (o plataforma)	Escalar arriba/abajo
Distancia a la próxima escalera	
Distancia al borde del mundo	
Posición y velocidad de los 3 barriles más cercanos	

Cuadro 1: Observaciones y acciones utilizadas en el modelo.

Lo que resulta en una red neuronal *Fully-Connected Feed-Fordward* de la forma que se puede observar en el siguiente diagrama:

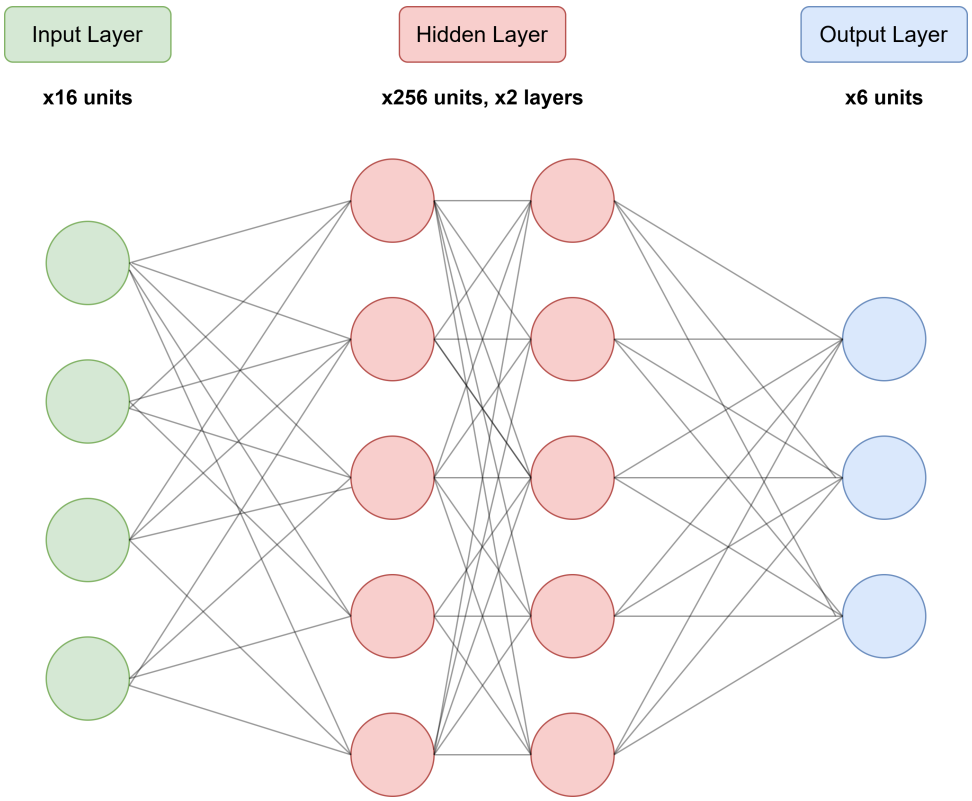

Figura 2: Forma de la red neuronal creada

4.3. Sistema de recompensas

En el contexto de aprendizaje por refuerzo es muy importante definir y probar diferentes sistemas de recompensas hasta encontrar el que mejor guie al agente hacia su objetivo, pero con ciertas restricciones de tal manera que el modelo pueda explorar y generalizar sin recaer siempre en la misma estrategia.

Para este proyecto se eligió un sistema de recompensas y penalidades que normaliza estas en un rango entre -1 y $+1$ (excepto la recompensa por ganar que es de 2), evitando así un des-balance, que afecte negativamente al entrenamiento. Este sistema es el detallado a continuación:

Acción	**Puntuación**
Ganar el nivel	2.0
Escalar escalera	0.5
Entrar a una nueva zona sin descubrir	0.5
Moverse una distancia horizontal significativa	0.2
Saltar barril	0.15
Hacer progreso vertical (incremental)	0.02

Cuadro 2: Recompensas.

Acción	**Puntuación**
Bajar la escalera	-0.1
Saltar muchas veces consecutivas	-0.05
Permanecer quieto por mucho tiempo	-0.25
Colisionar con barril	-1.0
Caer del mundo	-1.0

Cuadro 3: Penalizaciones.

Observación: la penalidad por colisión con barril es atenuada relativo al progreso vertical. A su vez las recompensas por entrar a una nueva zona aumentan relativo a la distancia entre esa zona y la meta.

4.4. Hiperparámetros óptimos

El ajuste de los hiperparámetros es clave para el rendimiento del agente en el algoritmo PPO. A continuación, se describen los valores seleccionados para los principales hiperparámetros basados en pruebas experimentales.

Hiperparámetros del algoritmo PPO

- **Tasa de aprendizaje (learning rate):** 0,0003, que permitió una actualización controlada de los parámetros del modelo.
- **Tamaño de mini-lote (*batch_size*):** 512, lo que permitió un entrenamiento eficiente sin sobrecargar la memoria.
- **Tamaño del buffer (*buffer_size*):** 20480, para almacenar suficiente experiencia y realizar actualizaciones efectivas.
- **Número de épocas (*num_epoch*):** 3, lo que permitió optimizar la política de forma eficiente sin sobreajuste.

- **Coeficiente de valor (λ):** 0,95, equilibrando la explotación y la exploración.
- **Coeficiente de entropía (β):** 0,01, incentivando la exploración sin perder estabilidad.
- **Clip (ϵ):** 0,2, para restringir los cambios en la política y asegurar un aprendizaje estable.
- **Descuento (γ):** 0,99, permitiendo al agente considerar tanto recompensas inmediatas como futuras.
- **Fuerza de la señal de recompensa ($strength$):** 1,0, asegurando el impacto adecuado de las recompensas en el aprendizaje.

Resultados con los hiperparámetros seleccionados

Con estos valores, el agente fue capaz de aprender a superar obstáculos como barriles y escaleras, logrando completar el primer nivel de 'Donkey Kong' con una mejora progresiva en el rendimiento durante el entrenamiento.

Otras configuraciones

Además de los hiperparámetros mencionados, se utilizaron las siguientes configuraciones:

- **Pasos máximos (max_steps):** $2,000,000$ pasos de entrenamiento, con la capacidad de terminar el entrenamiento manualmente en cualquier punto.
- **Horizonte temporal ($time\_horizon$):** 128 pasos.
- **Frecuencia de resúmenes ($summary\_freq$):** Cada 1000 pasos.

4.5. Entrenamiento del modelo

Luego de realizar varios intentos de prueba y error para determinar la configuración optima del modelo analizando y comparando el rendimiento de cada uno, se realizo el entrenamiento final (cuyos resultados se podrán observar en la siguiente sección) luego del cual se logro obtener un agente capaz de llegar a la meta saltando barriles y escalando las escaleras mas cercanas a este. Este entrenamiento se realizo con 10 agentes concurrentes (que comparten la misma red neuronal) aprovechando la aceleración de GPU haciendo uso de la plataforma CUDA de Nvidia a través de PyTorch, lo que permitió disminuir los tiempos de ajuste.

5 Resultados y análisis de métricas

Utilizando la herramienta TensorBoard se puede observar información sobre el desempeño de los modelos entrenados durante su entrenamiento, de esta manera se obtuvieron los siguientes gráficos que muestran la comparación entre los dos mejores modelos resultantes de este trabajo. El primero (en naranja) es un modelo que a pesar de lograr llegar a la meta no logro generalizar lo suficiente para ganar cuando hay cambios en el entorno, por ejemplo cuando se elimina una escalera, esto se debe a que no logro aprender a saltar barriles efectivamente, este defecto se logro corregir en el segundo modelo (en celeste). Por lo tanto en el gráfico celeste se ve un decremento de la recompensa acumulada en un punto, ese es el punto en el que se elimina la escalera mencionada para forzar al agente a aprender a saltar barriles. También se observa como los episodios duran cada vez menos a lo largo del entrenamiento, hasta llegar a un punto donde se mantienen constantes.

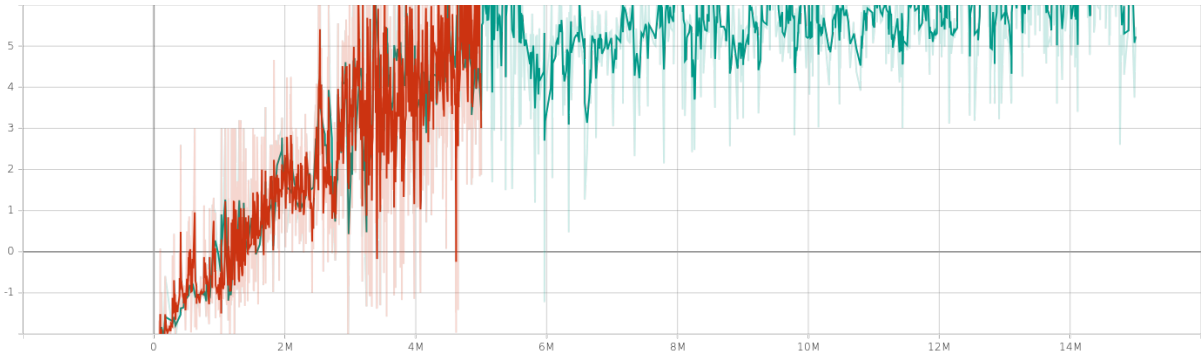

Figura 3: Recompensa acumulada.

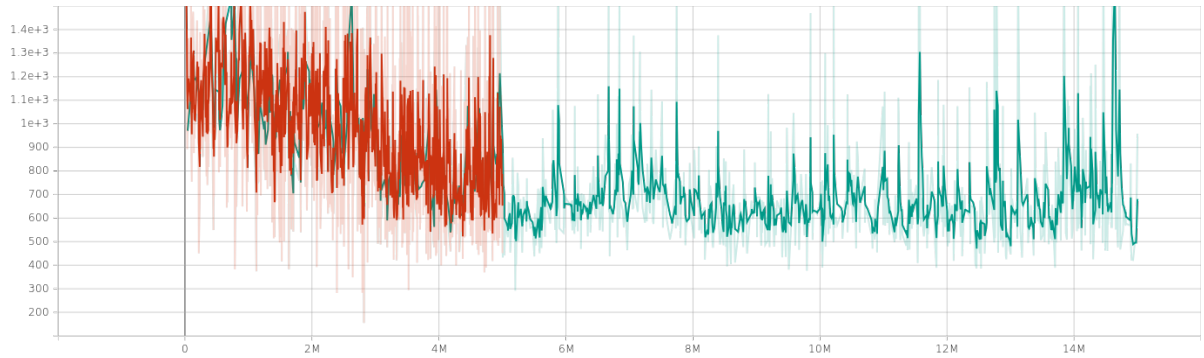

Figura 4: Duración de episodios.

En ***este video***, se presenta al modelo entrenado en acción, jugando el nivel de 'Donkey Kong' y demostrando su desempeño tras el proceso de aprendizaje.

6 Conclusión

Este proyecto demostró la eficacia del aprendizaje por refuerzo, utilizando el algoritmo Proximal Policy Optimization (PPO), para entrenar un agente capaz de superar el primer nivel del videojuego 'Donkey Kong'. Gracias a herramientas como Unity, ML-Agents y PyTorch, se creó un entorno de simulación robusto y un modelo eficiente que aprendió a esquivar barriles, escalar escaleras y alcanzar la meta.

El diseño adecuado de observaciones, acciones y recompensas fue clave para guiar el aprendizaje del agente. A pesar de los desafíos técnicos, como la configuración de hiperparámetros y el ajuste del sistema de recompensas, el modelo logró un rendimiento estable y adaptativo frente a cambios en el entorno.

En resumen, este trabajo resalta el potencial del aprendizaje por refuerzo en entornos dinámicos y abre la puerta a futuras mejoras, como su aplicación en niveles más complejos o en escenarios con mayor variabilidad.

7 Referencias

[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. Disponible en: `https://arxiv.org/pdf/1707.06347`

[2] Unity Technologies. (s.f.). *Unity - Plataforma de desarrollo de videojuegos*. Disponible en: `https://unity.com/es`

[3] Anaconda, Inc. (s.f.). *Anaconda: Distribución de Python para Ciencia de Datos*. Disponible en: `https://www.anaconda.com/`

[4] PyTorch. (s.f.). *PyTorch: Deep Learning Framework*. Disponible en: `https://pytorch.org/`

[5] Unity ML-Agents. (s.f.). *Repositorio de ML-Agents en GitHub*. Disponible en: `https://github.com/Unity-Technologies/ml-agents`

[6] *Repositorio del proyecto en GitHub*. Disponible en: `https://github.com/maxogod/AI-Donkey-Kong`

[7] *Video del modelo entrenado jugando Donkey Kong*. Disponible en: `https://www.youtube.com/watch?v=5XQuDkWKL-M&ab_channel=MaxoGod`