
MAESTRÍA EN INTELIGENCIA ARTIFICIAL

APRENDIZAJE POR REFUERZO II

# TP1: Double DQN, Dueling DQN

**Autor:**
**Maximiliano Torti**

Docente:
Miguel Augusto Azar

*Este trabajo fue realizado en mayo de 2025.*

# Índice general

# 1 Desarrollo

## 1.1. Introducción

En el presente trabajo práctico, se desarrolla la implementación de los algoritmos de Double-DQN y Dueling-DQN para que aprendan a jugar en forma automática al Tetris. El objetivo es poder evaluar la implementación de los algoritmo y compararlos en sus beneficios, defectos y rendimiento en un juego complejo.

Para lograrlo, se adaptó de [1] el ambiente de Tetris. El repositorio original ya contenía la definición de bloques, grilla, lógica de juego, puntaje y visualizaciones utilizando la libraría Pygame. Sobre esta base se agregó una clase GameUI que gestiona el flujo de visualizaciones y la interacción con los diferentes tipos de jugadores, y se agregó un método que permite obtener el estado del juego en forma codificada, necesario para entrenar modelos de aprendizaje por refuerzo basados en valor de estado. En la sección 1.2 se comentan mas detalles del ambiente.

Para los agentes, se adaptaron del repositorio [2] los algoritmos de Double-DQN y Dueling-DQN. Esta adaptación se describe con mayor detalle en la sección 1.3.

Finalmente, se agrego un script principal main.py que facilita la ejecución de los casos de uso principales: entrenar los algoritmos mencionados o comenzar un nuevo juego de Tetris con interfaz visual donde el jugador es un humano o un modelo entrenado.

## 1.2. Entorno

El juego comienza luego de definir el tipo de jugador entre las 2 posibilidades (humano o modelo de aprendizaje por refuerzo), instanciar la clase *GameUI* y ejecutar el método *play*. En ese instante se imprime en pantalla el tablero de la figura 1.1.

Se permiten las siguientes acciones:

- Abajo: desplaza el bloque actual una posición hacia abajo.
- Derecha: desplaza el bloque actual una posición hacia la derecha.
- Izquierda: desplaza el bloque actual una posición hacia la izquierda.
- Rotar: rota el bloque actual 90° en sentido horario.
- Omitir: no realiza ninguna acción y permite que el juego continúe por si solo.

Independientemente de las acciones ejecutadas el bloque desciende una posición cada 200 ms, y cuando el bloque actual toca el piso, este queda fijo y comienza a caer un nuevo bloque seleccionado aleatoriamente.

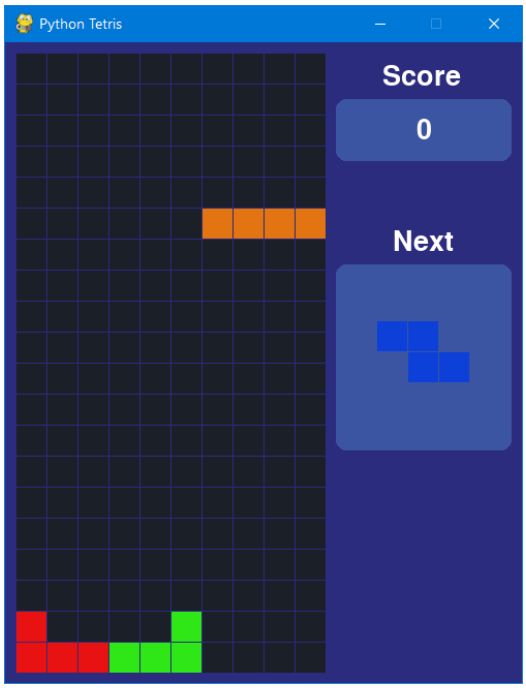


FIGURA 1.1. Tablero de Tetris.

Para observar el estado del ambiente, se agregó un método que retorna una matriz de 20 filas y 10 columnas (tamaño del tablero de Tetris), donde cada elemento de la matriz es 1 si la posición respectiva en el tablero está ocupada por un bloque (ya sea el bloque actual o uno fijado) o 0 si la posición está libre. Por simplicidad no se incluyó el bloque siguiente o el puntaje actual en la observación del estado. Bajo estas condiciones, el Tetris presenta mas de $1{,}6 * 10^{60}$ estados diferentes.

Para los puntajes, el juego base asigna 1 punto por cada ejecución de la acción abajo (de manera que el bloque descienda mas rápido), 100 puntos por completar una línea, 300 por completar 2 líneas en simultáneo y 500 por completar 3. Sobre esta lógica se agrego una quita de 5 puntos por perder el juego.

## 1.3. Agentes

El juego admite agentes humanos o modelos de aprendizaje por refuerzo. El jugador humano puede pulsar cualquiera de las flechas para realizar la acción correspondiente (la flecha arriba ejecuta la acción rotar) cuantas veces crea conveniente. En cuanto a los modelos de aprendizaje por refuerzo, dada la enorme cantidad de estados posibles no se pueden implementar algoritmos como Monte Carlo ES o Q-Learning, ya que estos requieren crear una tabla en memoria con el seguimiento de los valores de todos los pares estado-acción. Por este motivo, se implementan los algoritmos de Doouble-DQN y Dueling-DQN que reemplazan esta tabla en memoria por una función ajustable de aproximación de los valores de estado-acción.

El objetivo de los agentes DQN es aprender a realizar movimientos favorables en forma automática. Por simplicidad se encuentran restringidos a ejecutar un solo movimiento cada 200 ms. Es decir, en cada turno (cada 200 ms) el agente observa el estado actual del tablero, toma una de las 5 acciones posibles según su política

aprendida, el ambiente se actualiza ejecutando la acción y luego moviendo el bloque actual una posición hacia abajo.

Para la arquitectura de los modelos, inicialmente se hicieron pruebas con redes densamente conectadas, pero el rendimiento conseguido fue muy pobre. En su lugar, se decidió utilizar primero capas convolucionales seguido de capas densas, como se muestra en las figuras 1.2 y 1.3. Estas arquitecturas recuerdan a la arquitectura básica LeNet-5. Dado que la primer capa es una capa convolucional, el estado se codificó entonces como una imagen de (20,10) pixeles con 1 solo canal.

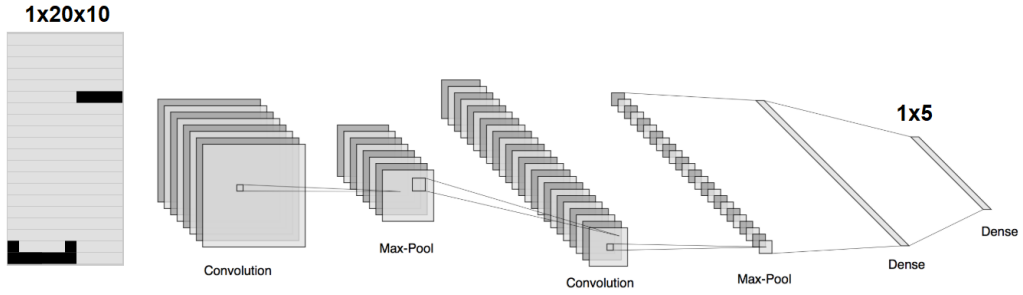


FIGURA 1.2. Arquitectura Double-DQN.

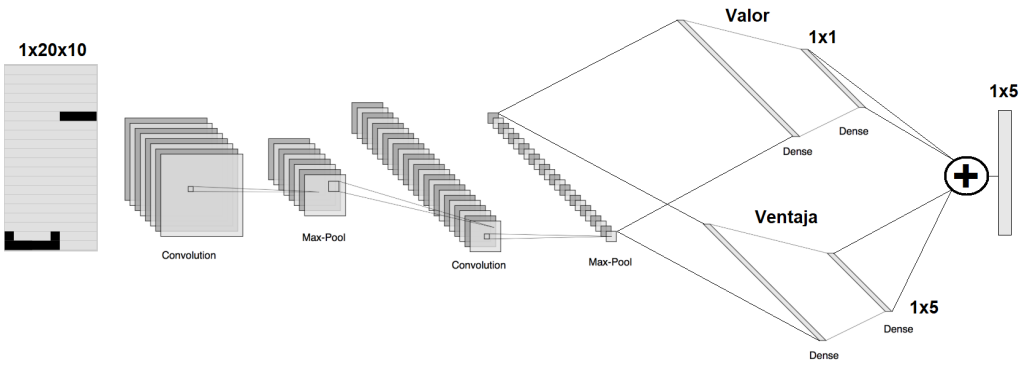


FIGURA 1.3. Arquitectura Dueling-DQN.

## 1.4. Entrenamiento

### 1.4.1. Double-DQN

Para entrenar el agente de Double-DQN, se ejecutaron múltiples juegos consecutivos de Tetris con la participación del modelo en entrenamiento y sin impresión de la interfaz gráfica. Se implementó el algoritmo de [3] que se muestra en la figura 1.4, con los siguientes detalles:

- $a_t$ puede tomar cualquiera de las 5 acciones posibles.
- $\pi(a_t, s_t)$ sigue una política $\epsilon$-greedy son $\epsilon_{inicial} = 1$ y $\epsilon_{final} = 0{,}001$.
- $Q_\theta$ y $Q_{\theta\prime}$ son redes DQN convolucionales que implementan la arquitectura de la figura 1.2.
- Las recompensas $r_t$ son las mencionadas en la sección 1.2.

- $s_t$ es una matriz de dimensiones (20,10).
- El replay buffer tiene un tamaño de 10.000 muestras.
- Se fijo un $\gamma$ (factor de descuento) de $0{,}9$.
- Para el entrenamiento de gradiente descendiente se utilizó un batch de 64 muestras y un lr de $0{,}001$.
- La red objetivo se actualizó cada 100 actualizaciones de gradiente.
- El entrenamiento se repitió durante 1.000 episodios, demorando aproximadamente 1 hora.

**Algorithm 1 : Double Q-learning (Hasselt et al., 2015)**

Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
**for** each iteration **do**
  **for** each environment step **do**
    Observe state $s_t$ and select $a_t \sim \pi(a_t, s_t)$
    Execute $a_t$ and observe next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
    Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
  **for** each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_{\theta'}(s_{t+1}, argmax_a Q_\theta(s_{t+1}, a))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
  After n steps update target network parameters:
    $\theta' \leftarrow \theta$

FIGURA 1.4. Algoritmo de Double-DQN.

### 1.4.2. Dueling DQN

Para entrenar el agente de Dueling-DQN, se replicó el mismo proceso que en la sección 1.4.1, con el mismo algoritmo de figura 1.4, difiriendo solamente en la implementación de $Q_\theta$ y $Q_{\theta\prime}$ que en este caso son redes convolucionales que implementan la arquitectura de la figura 1.3 y calculan por separado el valor general de un estado y las ventajas de cada acción en ese estado.

## 1.5. Resultados

### 1.5.1. Double-DQN

Luego de entrenar el agente Double-DQN por 1.000 episodios se obtuvo la curva de recompensa de la figura 1.5.

Se observa que la media móvil de la recompensa aumenta lentamente con el paso de los episodios desde un valor de 30 hasta aproximadamente 100, lo que indica que el modelo está entrenándose. No obstante, dado el puntaje, es de suponer que la política que se está aprendiendo no es la de completar líneas, ya que cada línea simple completada otorga 100 puntos. Si observamos la evolución de la recompensa sin filtrar, si notamos que cada ciertos episodios obtenemos recompensas mayores a 200 e incluso 300, por lo que en ciertos casos el agente logra completar líneas.

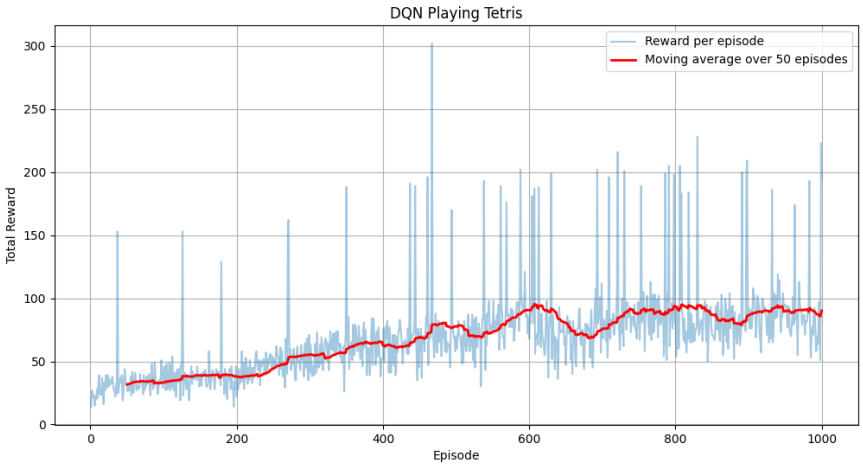


FIGURA 1.5. Recompensa por episodio para algoritmo de Double-DQN, suavizado con una media móvil de 50 episodios.

Al ejecutar el juego con el modelo entrenado como jugador, se observa que intenta completar lo máximo posible la grilla pero con una política muy básica. Solo distribuye los bloques a lo ancho del tablero (para extender perder el juego) con pocas o nulas rotaciones, e intenta bajar los bloques lo más rápido posible con la acción abajo para obtener los puntos de recompensa que esto otorga, como muestra la figura 1.6. Ocasionalmente, la política anterior completa una línea, como en la figura 1.7, pero solo por coincidencia, el modelo no busca activamente completar lineas. En la sección 1.6 se detallaran las razones de este resultado poco satisfactorio.

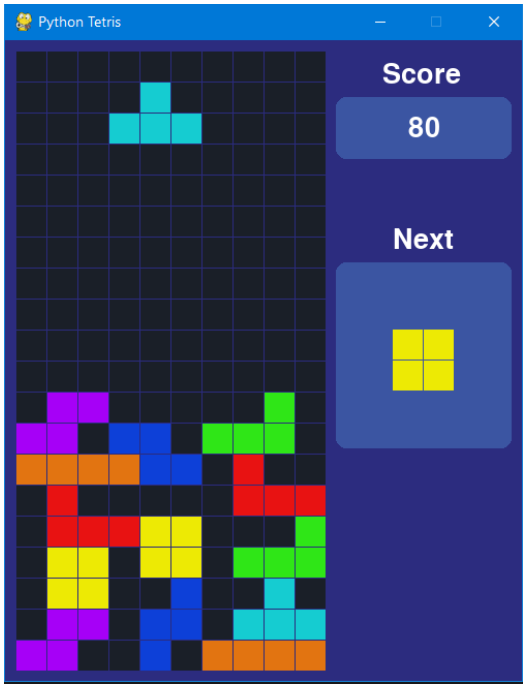


FIGURA 1.6. Agente Double-DQN distribuyendo bloques a lo ancho del tablero.

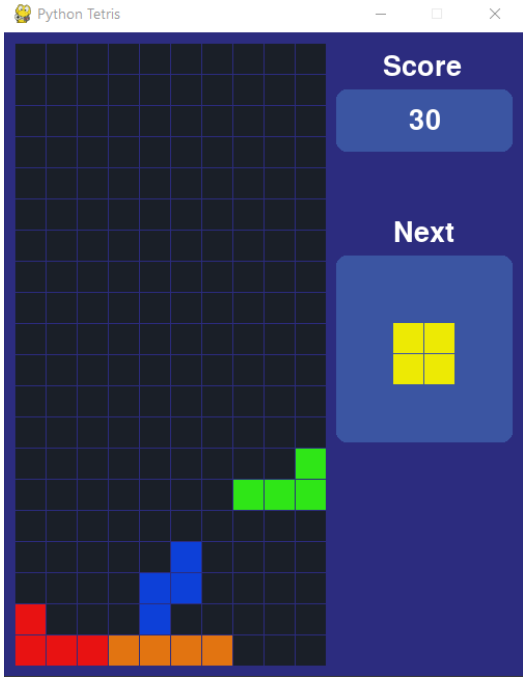


FIGURA 1.7. Agente Double-DQN completando una línea.

### 1.5.2. Dueling-DQN

Al entrenar el agente Dueling-DQN por 1.000 episodios, se obtuvo la curva de recompensa de la figura 1.8.

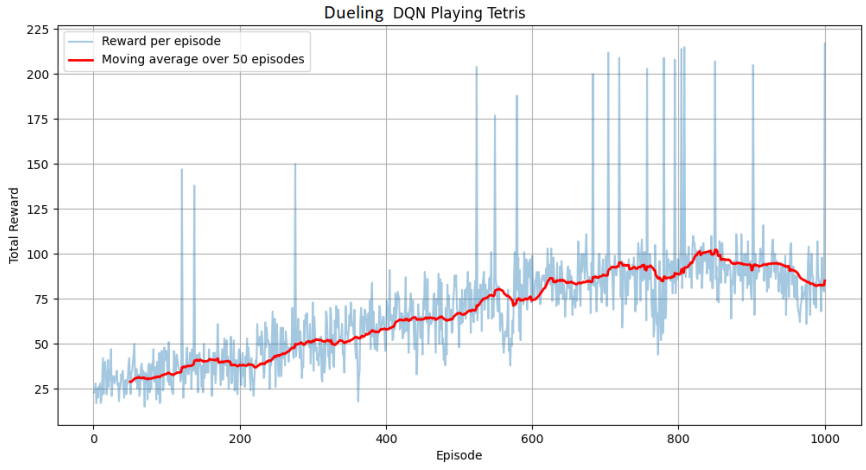


FIGURA 1.8. Recompensa por episodio para algoritmo de Dueling-DQN, suavizado con una media móvil de 50 episodios.

La curva muestra leves mejoras, aunque poco apreciables. Al ejecutar el juego con el modelo entrenado como jugador, se observa que este modelo realiza una mejor distribución de los bloques a lo ancho del tablero, como se muestra en figura 1.9, pero tiene las mismas limitaciones que el algoritmo anterior.

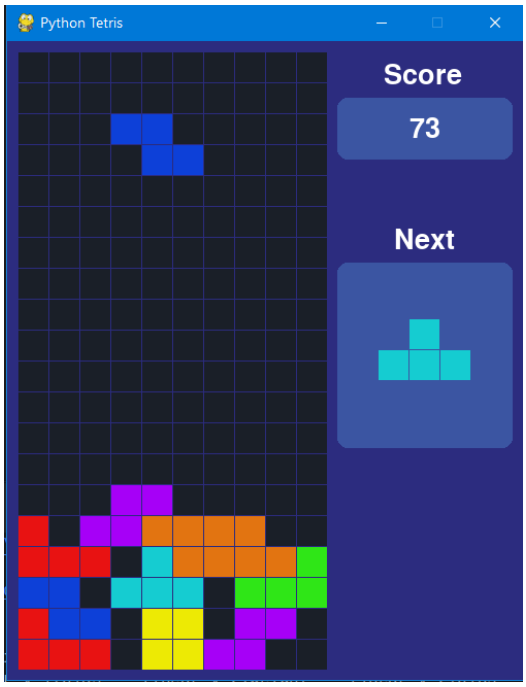


FIGURA 1.9. Agente Dueling-DQN distribuyendo bloques a lo ancho del tablero.

A fin de verificar que el rendimiento no esté siendo limitado por la cantidad de episodios, se continuó el entrenamiento de este caso durante 10.000 episodios más, demorando aproximadamente 10 horas. La curva de recompensa obtenida se muestra en la figura 1.10.

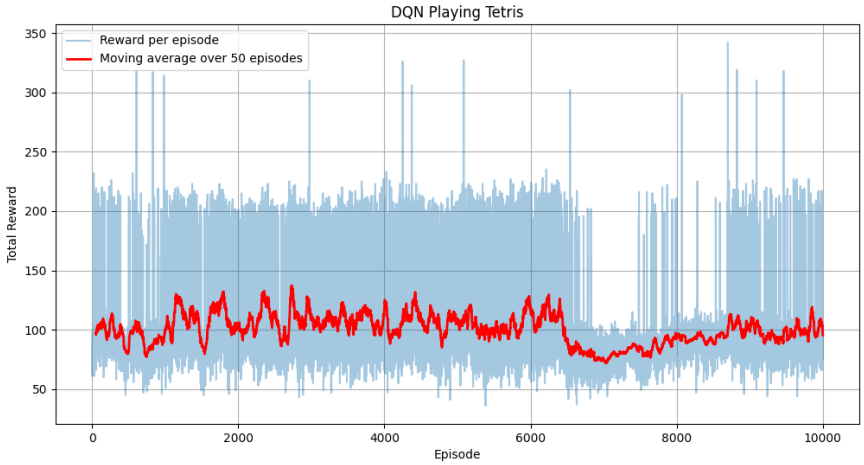


FIGURA 1.10. Recompensa por episodio para algoritmo de Dueling-DQN luego de 10.000 episodios, suavizado con una media móvil de 50 episodios.

Ahora, en la mayoría de episodios el modelo completa una o dos líneas, pero nuevamente la media no supera la barrera de los 150 puntos. Es decir, el modelo sigue aprendiendo simplemente a distribuir bloques y el aumento de la cantidad de episodios no mejora sustancialmente los resultados obtenidos.

## 1.6. Conclusiones

Ambos algoritmos mostraron resultados poco satisfactorios, aprendiendo solamente una política básica de distribuir bloques y extender el juego lo máximo posible pero no de completar líneas.

Comparando, Dueling-DQN obtuvo resultados levemente mejores gracias a que separa el cálculo del valor general para el estado actual y de la ventaja que se conseguiría ejecutando cada acción, lo cual es beneficioso en un caso como el del juego del Tetris donde todas las acciones tienen aproximadamente la misma recompensa (valor general) y algunas acciones representan un pequeño valor agregado (ventaja).

Al analizar en detalle la limitación de los algoritmos para este caso, se concluyó lo siguiente:

- Ambos algoritmos se basan en $TD(0)$, lo que provoca que sean extremadamente codiciosos y busquen la recompensa inmediata. Esto es contraproducente en un juego como el Tetris donde la recompensa real (completar líneas) se obtiene luego de más de 50 pasos futuros correctos.
- Estas variantes de DQN basadas en $TD(0)$ realizan estimaciones de $Q(t)$ y $Q(t+1)$ que pueden presentar un gran error de bias en entornos complejos. Por lo tanto, es de esperar que el incremento del valor de estado obtenido por completar una línea luego de 50 pasos futuros no se vea reflejado correctamente en las estimaciones de valor del estado actual o inmediato próximo que realizan la redes neuronales.
- El Tetris requiere ejecutar correctamente múltiples pasos consecutivos para llegar a estados favorables. Es muy poco probable que la política $\epsilon$-greedy logre una buena exploración del espacio de estados y acciones posibles. Entonces, el modelo tendrá muy pocas instancias de movimientos correctos para ajustar su política, y muchas instancias de movimientos que no aportan valor en su entrenamiento.

Para poder resolver estas problemáticas se puede o bien usar otro algoritmo que tenga menor error de bias en la estimación de los valores de estado (como Actor-Critic), o modificar los algoritmos utilizados de DQN. Como ejemplo, [4] plantea utilizar una variante de DQN con los siguientes cambios:

- Agregar pre-entrenamiento con episodios jugados por humanos (llamado demostración), que poseen mayor concentración de muestras con pares estado-acción favorables.
- Realizar un muestreo ponderado del replay buffer al obtener los batch de entrenamiento de la red neuronal, priorizando los pasos de episodios que hayan conseguido altos puntajes.
- Modificar la función de pérdida de la red neuronal:

  $J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q)$

  Donde $J_{DQ}(Q)$ es el error clásico $TD(0)$, $J_n(Q)$ es el error TD(n) qu permite disminuir el error de bias mencionado, $J_E(Q)$ está solo presente en el pre-entrenamiento y es un término de error cross-entropy que fuerza a

la red a seguir los pasos que siguió el humano, y $J_{L2}(Q)$ es un término de regularización L2 para evitar el overfitting.

Lamentablemente la implementación de ambas soluciones escapan del alcance de este proyecto y se enuncian como trabajo futuro.

## 1.7. Trabajo futuro

Como trabajo futuro se plantea implementar y comparar las soluciones a las limitaciones encontradas en los algoritmos de Double-DQN y Dueling-DQN en el juego de Tetris, mencionadas en la sección 1.6:

1. Implementar un algoritmo que posea menor error de bias en la estimación del valor de estado como, por ejemplo, Actor-Critic.
2. Implementar las técnicas de demostración humana, muestreo ponderado sobre el replay buffer y función de pérdida modificada sobre Dueling-DQN.

# A Script principal

El script principal del proyecto contiene un procesador de líneas de comando que facilita la ejecución de los diferentes casos de uso. Los usos básicos del procesador se detallan a continuación:

- Mostrar la ayuda.

```
$ python -m src.main --help
```

- Jugar al Tetris con el teclado.

```
$ python -m src.main play --human
```

- Entrenar un nuevo agente de Double-DQN.

```
$ python -m src.main train --model=ddqn
```

- Entrenar un nuevo agente de Dueling-DQN.

```
$ python -m src.main train --model=dueling-dqn
```

- Ejecutar el juego Tetris con un agente Double-DQN entrenado.

```
$ python -m src.main play \
    --model-file="./resources/ddqn/model_ddqn.pkl" --model=ddqn
```

- Ejecutar el juego Tetris con un agente Dueling-DQN entrenado.

```
$ python -m src.main play \
    --model-file="./resources/dueling_dqn/model_dueling_dqn.pkl" \
    --model=dueling-dqn
```

```
>python -m src.main --help
pygame 2.6.1 (SDL 2.28.4, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org/contribute.html
This script allows you to play Tetris game or to train a DQN model to play it and then run the game.

Usage:
    main.py play [--model-file=<model-file> --model=<model_type>] [--human]
    main.py train --model=<model-type>

Options:
    -h --help                       Show this screen.
    --human                         Play tetris using keyboard.
    --model-file=<model-file>       Specify the model used to play the game.
    --model=<model_type>            The type of model to play/train. Valid options are: ddqn or dueling-dqn.
```

FIGURA A.1. Uso del script principal por línea de comando.

# B Calidad de código

Para mantener una buena calidad de código, el repositorio cuenta con documentación extensiva y un archivo README.md donde se detalla el uso del proyecto. Adicionalmente se ejecutaron análisis estáticos de calidad de código mediante la herramienta SonarLint y se resolvieron las malas prácticas y los problemas de seguridad encontrados.

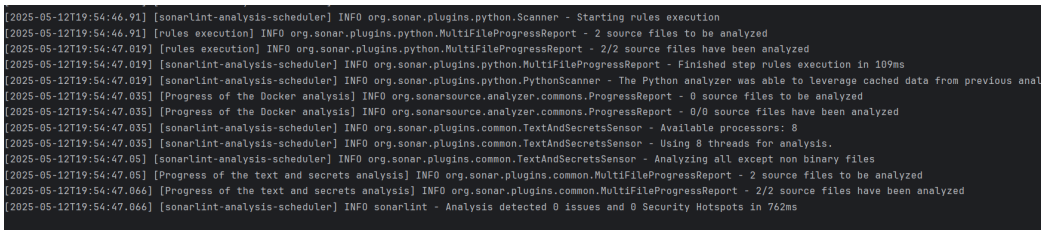


FIGURA B.1. Análisis estático de código.

# C Repositorio

El repositorio con el código fuente y archivos adicionales de utilidad se encuentra disponible en el siguiente link:

https://github.com/maxit1992/MIA_RL2/tree/master/tp1

# Bibliografía

[1] Nick Koumaris. *Python Tetris Game Pygame*. https://github.com/educ8s/Python-Tetris-Game-Pygame. Jun. de 2023. (Visitado 02-05-2025).
[2] Miguel Augusto Azar. *ar2*. https://github.com/aear-uba/ar2. Abr. de 2025. (Visitado 02-05-2025).
[3] Hasselt at al. «Deep Reinforcement Learning with Double Q-learning». En: *AAAI* (2016).
[4] Hester at al. «Deep Q-learning from Demonstrations». En: *AAAI* (2018).