

Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak



Inteligencia Artificial *Obligatorio*

Autores:

Felipe Briosso - 269851
Iara Vieira - 252258
Fiorella Alvira - 234232

Docentes:

Alejo Garat
Matias Molinolo

2025

FLAN: Flight-Level Adjustment Network.....	3
Pre-procesamiento de los datos.....	3
Discretización.....	3
Modelo Q-Learning.....	3
Exploración de hiperparámetros.....	3
¿Qué hiperparámetros hay?.....	3
Test 1.....	3
Test 2.....	4
Test 3.....	4
Test 4.....	5
Test 5.....	6
Modelo final.....	8
Modelo Stochastic Q-learning.....	8
Ventajas del Stochastic Q-learning frente a Q-learning.....	8
Exploración de hiperparámetros y discretizaciones.....	8
Modelo final.....	9
BORED: Board-Oriented Reasoning for Emergent Domination.....	10
Descripción del juego TacTix.....	10
Táctica 1: Minimax con alpha-beta pruning.....	10
Consecuencias de haber aplicado alfa-beta pruning.....	11
Camino de pruebas.....	11
Prueba 1.....	12
Prueba 2.....	12
Prueba 3.....	12
Prueba 4.....	13
Prueba 5.....	14
Prueba 6.....	15
Prueba 7.....	16
Prueba 8.....	17
Prueba 9.....	18
Conclusión.....	18
Táctica 2: Expectimax.....	19
Diferencia clave respecto a Minimax.....	19
Pruebas realizadas.....	19
Prueba 1 — Profundidad 3 — 50 partidas.....	20
Prueba 2 — Profundidad 3 — 100 partidas.....	20
Prueba 3 — Profundidad 4 — 50 partidas.....	20
Prueba 4 — Profundidad 4 — 100 partidas.....	20
Prueba 5 — Profundidad 5 — 50 partidas.....	21
Prueba 6 — Profundidad 5 — 100 partidas.....	21
Observaciones.....	21
Análisis y conclusión.....	22

FLAN: Flight-Level Adjustment Network

Pre-procesamiento de los datos

Discretización

Primero se intentó generar entre -1 y 1 de tamaño 200 cada uno dado que se va de -infinito a +infinito y dado que las acciones van de -1 a 1 nos pareció correcto.

```
altitude_space = np.linspace(-1, 1, 200)
vertical_velocity_space = np.linspace(-1, 1, 200)
target_altitude_space = np.linspace(-1, 1, 200)
runway_distance_space = np.linspace(-1, 1, 200)
```

El problema es que nos encontramos con un límite computacional.

Así que bajamos la cantidad de elementos de cada array y decidimos aumentar el espacio de acción debido a que si daba problema si el array era entre -1 y 1.

```
altitude_space = np.linspace(-2, 2, 100)
vertical_velocity_space = np.linspace(-3, 3, 100)
target_altitude_space = np.linspace(-2, 2, 100)
runway_distance_space = np.linspace(-2, 2, 100)
```

Modelo Q-Learning

Exploración de hiperparámetros

¿Qué hiperparámetros hay?

Los hiperparametros son los siguientes:

```
alpha = 0.1          # tasa de aprendizaje
gamma = 0.99         # factor de descuento
epsilon = 1.0         # probabilidad de exploración
epsilon_decay = 0.995
epsilon_min = 0.01
episodes = 500
```

Para poder encontrar el mejor modelo se realizaron un total de 5 test que se detallan a continuación (el epsilon decay no se utilizó de primera).

Test 1

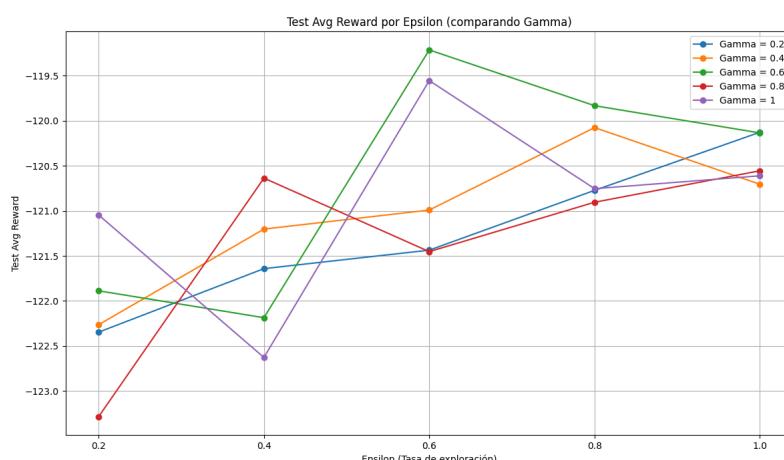
Para este primer test se buscó el mejor valor para cada hiperparametro en base comparación con los epsilon.

Se probaron con distintos epsilon de este modo se buscó ver que tanto mejoraba o empeoraba la IA en base a este parámetro.

Graficando para poder visualizar mejor los resultados. Empezando por gamma:

```
alpha = 0.2          # Tasa de aprendizaje
gammas = [0.2,0.4,0.6,0.8,1] # Factor de descuento
episodes = 500        # Episodios de entrenamiento
test_episodes = 100   # Episodios de testeo

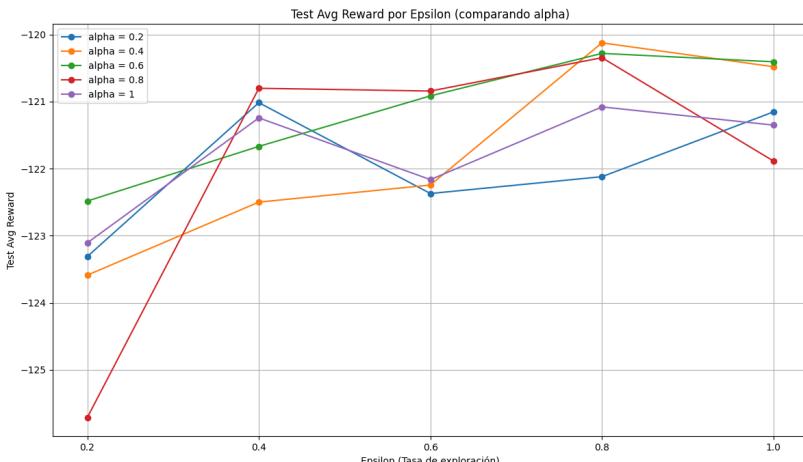
# Lista de valores de epsilon para probar
epsilon = [0.2, 0.4, 0.6, 0.8, 1.0]
```



Siguiendo por Alpha:

```
alphas = [0.2,0.4,0.6,0.8,1] # Tasa de aprendizaje
gamma = 0.6                  # Factor de descuento
episodes = 500                # Episodios de entrenamiento
test_episodes = 100             # Episodios de testeo

# Lista de valores de epsilon para probar
epsilons = [0.2, 0.4, 0.6, 0.8, 1.0]
```



Se llegó a un mejor resultado: 0.6 gamma, 0.6 de epsilon y alpha 0.4 para epsilon 0.8.

Test 2

Tras el primer test se investigó en mayor profundidad y vimos que 500 episodios era muy poco. En este test se buscó ver que tan eficiente era aumentar la cantidad de episodios de training.

```
alpha = 0.4 # Tasa de aprendizaje
gamma = 0.9 # Factor de descuento
list_episodes = [1000,2000,5000,10000,20000] # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.05 # Valor mínimo de epsilon

==== Comparación Final ====
episodes: 1000.00 | Train Avg Reward: -99.30 | Test Avg Reward: -103.31
episodes: 2000.00 | Train Avg Reward: -93.09 | Test Avg Reward: -94.10
episodes: 5000.00 | Train Avg Reward: -98.34 | Test Avg Reward: -87.56
episodes: 10000.00 | Train Avg Reward: -94.58 | Test Avg Reward: -109.26
episodes: 20000.00 | Train Avg Reward: -102.86 | Test Avg Reward: -103.52
```

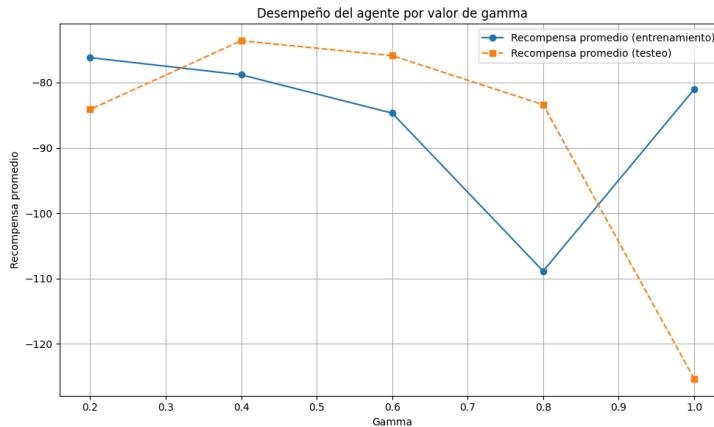
Con esto llegamos a que aumentar tanto los episodios no hace tanta diferencia. Pero 500 claramente era muy poco.

Test 3

Tomando el test 2 se rehizo el test 1 pero ahora con más episodios y con un epsilon decay.

```
alpha = 0.4 # Tasa de aprendizaje
gammas = [1,0.8,0.6,0.4,0.2] # Factor de descuento
episodes = 5000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon
```

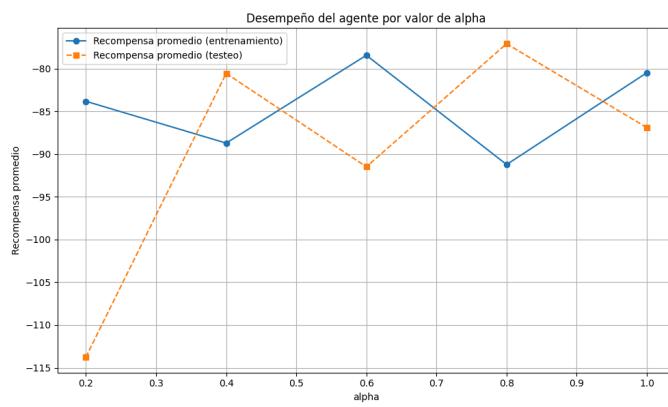


```

alphas = [1,0.8,0.6,0.4,0.2] # Tasa de aprendizaje
gamma = 0.4      # Factor de descuento
episodes = 5000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon

```



```

alpha = 0.8 # Tasa de aprendizaje
gamma = 0.4      # Factor de descuento
episodes = 10000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.9999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon

```

```

== Comparación Final ==
{'alpha': 0.8, 'episodes': 10000, 'train_avg_reward': -98.00420788538183, 'test_avg_reward': -73.48736668398334}

```

```

alpha = 0.4 # Tasa de aprendizaje
gamma = 0.4      # Factor de descuento
episodes = 10000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.9999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon

```

```

== Comparación Final ==
{'alpha': 0.4, 'episodes': 10000, 'train_avg_reward': -87.32144673566633, 'test_avg_reward': -69.33842271825}

```

Test 4

A pesar de que en el test 3 se llegó al mejor resultado hasta el momento dado el funcionamiento de alpha y gamma lo mejor era probarlas en conjunto que por separado.

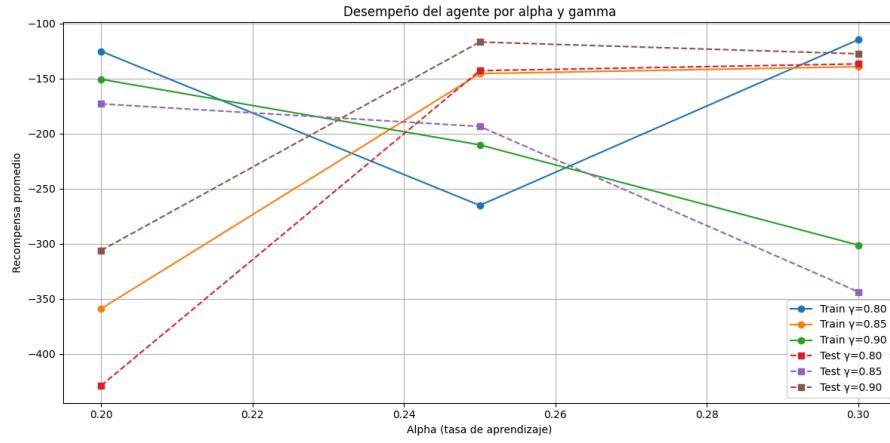
A recomendación de los docentes se busca comenzar con un alpha entre 0.2 y 0.3 y gamma entre 0.8 y 0.9. A partir de cada iteración se busca ir mejorando los valores de alpha y gamma.

```

alphas = [0.2,0.25,0.3] # Tasa de aprendizaje
gammas = [0.8,0.85,0.9] # Factor de descuento
episodes = 5000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon

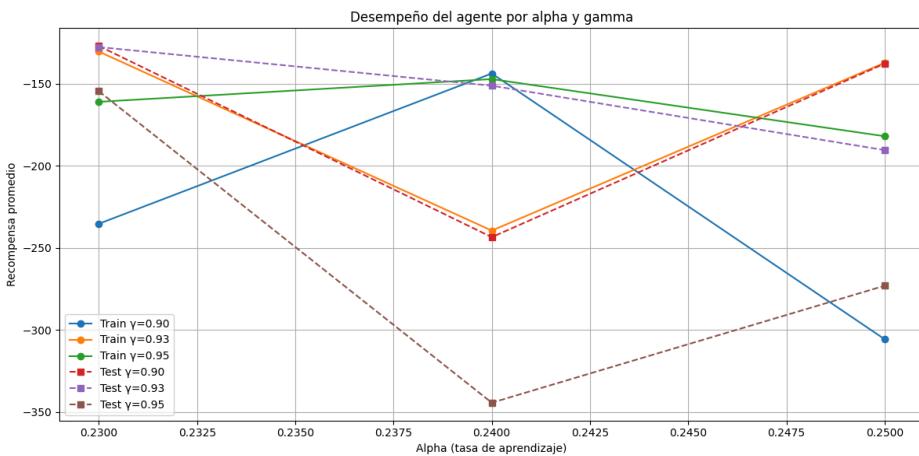
```



```

alphas = [0.230,0.25,0.4] # Tasa de aprendizaje
gammas = [0.85,0.9,0.95] # Factor de descuento
episodes = 5000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

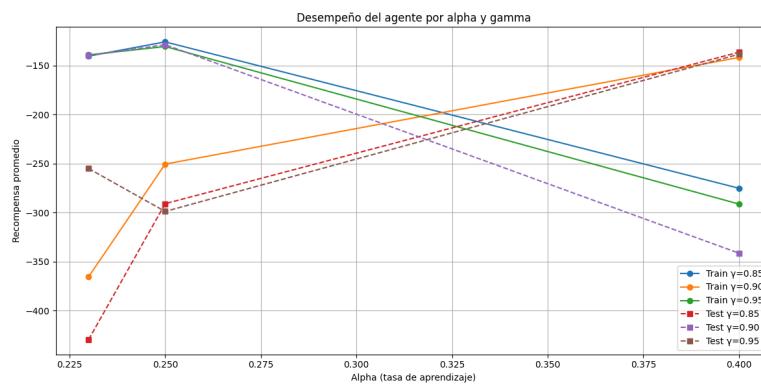
epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon
    
```



```

alphas = [0.23,0.24,0.25] # Tasa de aprendizaje
gammas = [0.9,0.925,0.950] # Factor de descuento
episodes = 5000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon
    
```



Test 5

Para este último test se busca mejorar la discretización probando distintos valores a partir de los siguientes hiperparametros base:

```

alphas = [0.23,0.24,0.25] # Tasa de aprendizaje
gammas = [0.9,0.925,0.95] # Factor de descuento
episodes = 5000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon

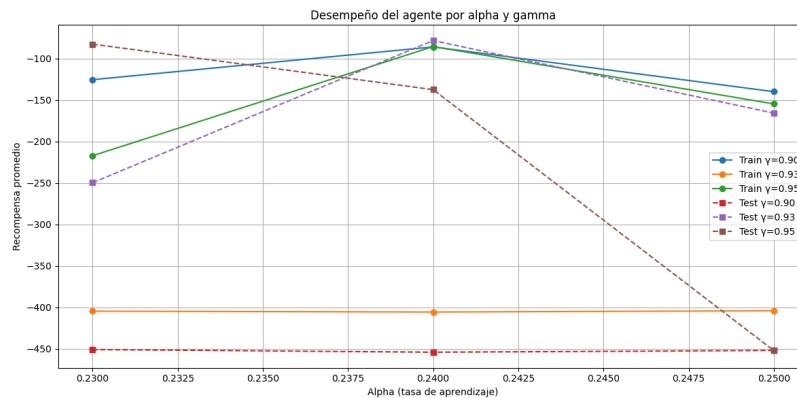
```

I.

```

altitude_space = np.linspace(-1000, 5000, 50)
vertical_velocity_space = np.linspace(-1000, 1000, 50)
target_altitude_space = np.linspace(-1000, 5000, 50)
runway_distance_space = np.linspace(-1000, 5000, 50)
altitude_space

```

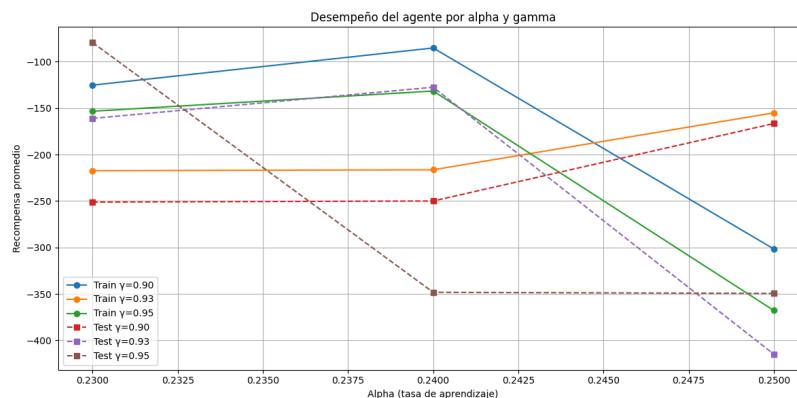


II.

```

altitude_space = np.linspace(-7000, 14000, 50)
vertical_velocity_space = np.linspace(-7000, 14000, 50)
target_altitude_space = np.linspace(-7000, 14000, 50)
runway_distance_space = np.linspace(-7000, 14000, 50)
altitude_space

```

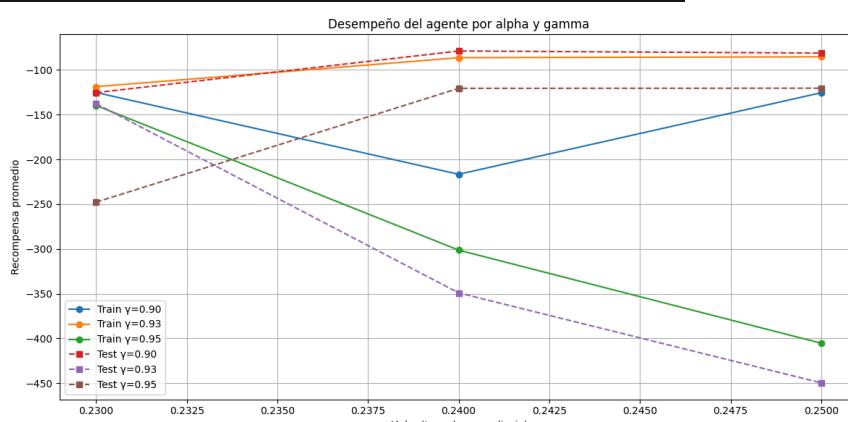


III.

```

altitude_space = np.linspace(-10000, 20000, 50)
vertical_velocity_space = np.linspace(-10000, 20000, 50)
target_altitude_space = np.linspace(-10000, 20000, 50)
runway_distance_space = np.linspace(-10000, 20000, 50)
altitude_space

```



Se realizaron un par intermedias pero no aportan valor a la muestra y se descartaron. Con estos ya podemos ver que el valor de la discretización es otra variable mas y no significa que para los mismos valores de gamma y alpha se encuentren mejoras solamente cambiando la discretización.

Modelo final

Nuestro mejor modelo al final resultó ser uno probado al principio.

Durante el test 3 teniendo la primera discretización probada (valores entre -2 y 2) y los siguientes hiperparametros:

```
alpha = 0.8  
gamma = 0.4  
episodios de training = 10000  
episodios de test= 100  
epsilon = 100  
epsilon decay = 0.9999  
epsilon min = 0.01
```

el modelo final se encuentra en el archivo *continuos_descent_env_final_model.ipynb*

El modelo final no alcanzó el rendimiento esperado. Consideramos que aún hay margen de mejora mediante la experimentación con diferentes técnicas de discretización y ajustes de hiperparámetros.

Modelo Stochastic Q-learning

Ventajas del Stochastic Q-learning frente a Q-learning

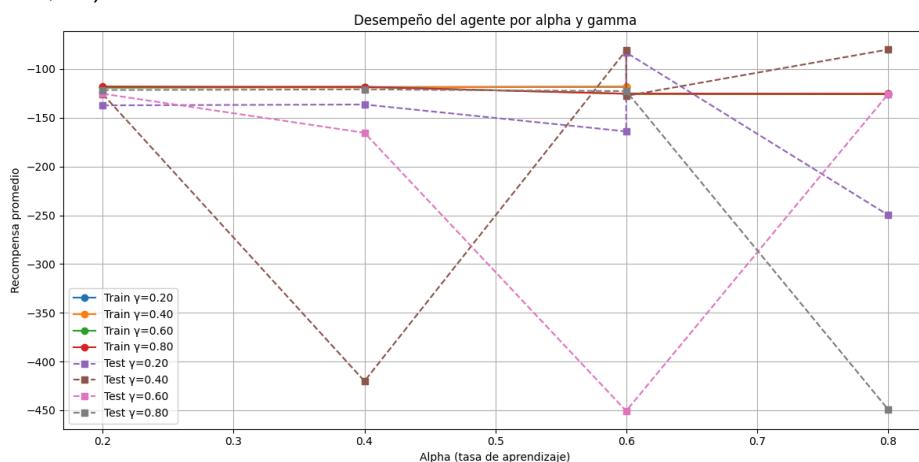
En entornos con espacios de estado infinitos o continuos, el uso de Stochastic Q-Learning ofrece varias ventajas clave. Permite aprender de manera incremental a partir de muestras individuales sin necesidad de explorar o almacenar todo el espacio, lo que lo hace escalable y eficiente. Además promueve una mejor exploración al incorporar aleatoriedad, ayudando al agente a evitar políticas subóptimas.

Exploración de hiperparámetros y discretizaciones

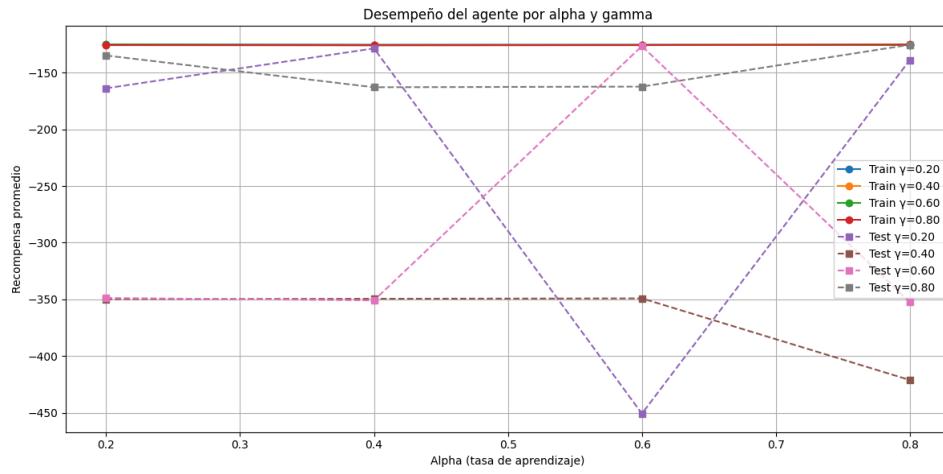
se probaron varios hiperparámetros y discretizaciones distintas para luego poder hacer pruning y llegar al mejor modelo.

```
alphas = [0.2,0.4,0.6,0.8] # Tasa de aprendizaje  
gammas = [0.2,0.4,0.6,0.8] # Factor de descuento  
episodes = 5000 # Episodios de entrenamiento  
test_episodes = 100 # Episodios de testeo  
  
epsilon = 1 # Valor inicial de epsilon  
epsilon_decay = 0.999 # Decaimiento de epsilon  
epsilon_min = 0.01 # Valor mínimo de epsilon
```

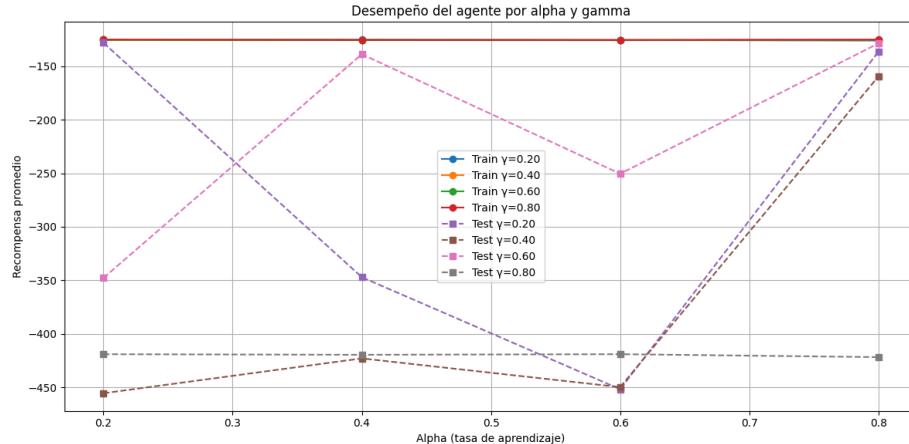
A partir de estos hiperparámetros se probaron las siguientes discretizaciones:
(-10000, 20000, 50)



(-5000, 10000, 50)



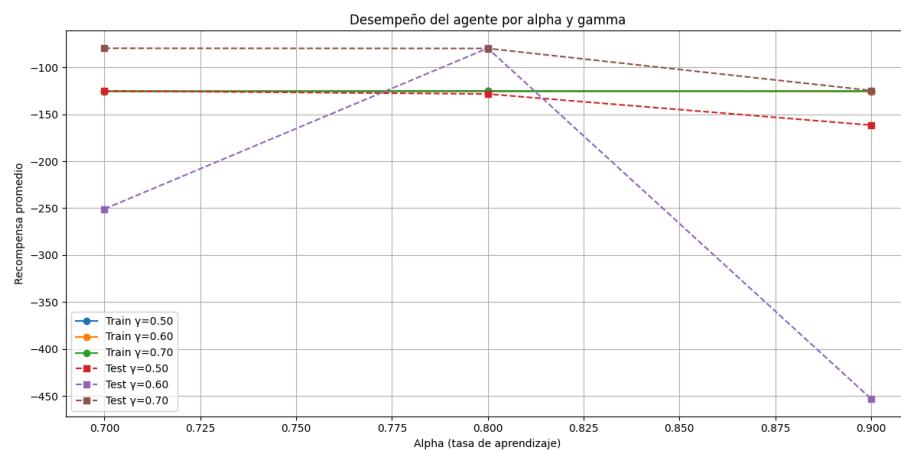
(-30000, 30000, 50)



Vemos que el training no podía mejorar más de aca así que decidimos seguir por este lado. Buscando ahora mejorar los hiperparametros:

```
alphas = [0.7,0.8,0.9] # Tasa de aprendizaje
gammas = [0.5,0.6,0.7] # Factor de descuento
episodes = 5000 # Episodios de entrenamiento
test_episodes = 100 # Episodios de testeo

epsilon = 1 # Valor inicial de epsilon
epsilon_decay = 0.999 # Decaimiento de epsilon
epsilon_min = 0.01 # Valor mínimo de epsilon
```



Modelo final

El modelo final toma la discretización final toma los valores (-30000,30000,50) y los siguientes hiperparametros:

```
alpha = 0.8
gamma = 0.7
episodios de training = 5000
episodios de test= 100
epsilon = 100
epsilon decay = 0.999
epsilon min = 0.01
```

el modelo final se encuentra en el archivo *continuos_descent_env_final_model_Stochastic.ipynb*

Nuevamente no se consiguió un resultado ideal por temas de tiempo no se logró probar más variaciones para buscar un mejor valor.

BORED: Board-Oriented Reasoning for Emergent Domination.

Descripción del juego TacTix

TacTix es un juego estratégico para dos jugadores basado en una cuadrícula de $N \times N$, donde se deben quitar casillas de una grilla hasta dejar al otro jugador sin movimientos.

Reglas del juego:

- Se juega en una grilla cuadrada de tamaño $N \times N$ (por defecto, 6x6).
- Al inicio del juego, todas las casillas están encendidas (representadas por 1).
- Las casillas removidas se representan con 0.
- Los jugadores se turnan para remover una secuencia contigua de fichas encendidas (1) de una sola fila o columna.
- Solo se puede remover una secuencia contigua de casillas encendidas (1) en una sola fila o columna.
- No puede haber casillas apagadas (0) dentro del rango.
- No se puede cruzar filas y columnas en una sola acción.

El desafío del juego está en:

- Reducir el tablero a una posición perdedora para el oponente.
- Controlar el ritmo del juego (quién tendrá que hacer la última jugada).
- Decidir si conviene jugar en una fila o columna.

Representación del estado y acciones:

El estado del juego (obs) se representa como un diccionario con:

```
{  
    "board": np.array de shape (N, N), con valores 1 (encendido) o 0 (apagado),  
    "current_player": 0 o 1  
}
```

Cada acción es una tupla de 4 valores:

```
(row_or_col, start, end, is_row)
```

Donde:

`row_or_col`: índice de fila o columna (de 0 a $N-1$).

`start y end`: índices del rango continuo de fichas a apagar.

`is_row`: 1 si se juega en una fila, 0 si en una columna.

Condiciones de victoria:

- Modalidad normal (por defecto): El juego termina cuando no queda ninguna casilla encendida (`np.count_nonzero(board) == 0`). El jugador que hace la última jugada gana.
- Modalidad Misère (regla inversa): En esta variante, el jugador que hace la última jugada pierde.

Táctica 1: Minimax con alpha-beta pruning

Para poder aplicar la técnica de Minimax con alpha-beta pruning creamos un archivo nuevo donde estará el código del agente: tactix > minimax_agent.py. Este agente hereda de la clase "[Agent.py](#)" que tiene las firmas básicas de un agente.

Tanto Minimax como Expectimax utilizan funciones heurísticas para ayudar al agente a distinguir si su jugada es buena o mala y aprender así a jugar mejor. Por esto agregamos en la clase "Agent" la firma de esta función que ambos implementarán.

```
1  from abc import ABC, abstractmethod  
2  from tactix_env import TacTixEnv  
3  
4  class Agent(ABC):  
5  
6      @abstractmethod  
7      def __init__(self, env:TacTixEnv):  
8          self.env = env  
9  
10     @abstractmethod  
11     def act(self, obs):  
12         return NotImplemented  
13  
14     @abstractmethod  
15     def heuristic_utility(self, board: TacTixEnv):  
16         return NotImplemented
```

Consecuencias de haber aplicado alfa-beta pruning

Alfa-beta pruning es una optimización del algoritmo Minimax que evita evaluar ramas del árbol de juego que no pueden influir en la decisión final.

- Alfa es el mejor valor que el jugador MAX puede garantizar hasta ese punto.
- Beta es el mejor valor que el jugador MIN puede garantizar hasta ese punto.
- Si durante la búsqueda se descubre que una rama no puede mejorar la decisión ya calculada, se “poda” y no se sigue explorando.

El espacio de estados en TacTix crece exponencialmente con el tamaño del tablero y la profundidad del árbol. Con poda, el agente evita “desperdiciar tiempo” en jugadas subóptimas cuando ya encontró algo mejor. Sin poda, evaluar a profundidad 6 puede tardar horas o días, mientras que con poda alfa-beta podés llegar a depth 6 o más en pocos minutos o menos. En nuestras pruebas los tiempos promedios para 200 corridas fueron:

- depth = 3 -> 7 minutos
- depth = 4 -> 25 minutos
- depth = 6 -> 66 minutos

Camino de pruebas

En el agente definimos el algoritmo minimax que usaremos para jugar. Este está hecho de la siguiente manera:

- Condición base: si llegamos a profundidad 0 o no hay más movimientos, devolvemos el resultado de la heurística (heuristic_utility).
- Si estamos maximizando: explora cada acción posible, simula el movimiento, llama recursivamente a minimax pero ahora como el jugador minimizador, actualiza el mejor resultado y aplica poda cuando beta <= alpha.
- Si estamos minimizando: igual que el caso anterior, pero minimizando la evaluación y actualizando beta.
- La función “simulate_move”: hace una copia del tablero y aplica la acción simulando qué pasa si tomás esas casillas sin modifica el entorno real.

```
5   class MinimaxTacTixAgent(Agent):
6
7      32     def minimax(self, board, depth, maximizing_player, alpha, beta):
8          valid_actions = self.get_valid_actions(board)
9          if depth == 0 or not valid_actions:
10              return self.evaluate_board(board), None
11
12              best_action = None
13
14              39      if maximizing_player: # si toca maximizar
15                  max_eval = -np.inf
16                  for action in valid_actions:
17                      new_board = self.simulate_move(board, action)
18                      eval, _ = self.minimax(new_board, depth - 1, False, alpha, beta)
19                      if eval > max_eval:
20                          max_eval = eval
21                          best_action = action
22                      alpha = max(alpha, eval)
23                      if beta <= alpha:
24                          break
25
26                  return max_eval, best_action
27
28
29      else: # si toca minimizar
30          min_eval = np.inf
31          for action in valid_actions:
32              new_board = self.simulate_move(board, action)
33              eval, _ = self.minimax(new_board, depth - 1, True, alpha, beta)
34              if eval < min_eval:
35                  min_eval = eval
36                  best_action = action
37                  beta = min(beta, eval)
38                  if beta <= alpha:
39                      break
40
41          return min_eval, best_action
42
43
44      def simulate_move(self, board, action):
45          # Realiza una copia del tablero y aplica la acción
46          new_board = copy.deepcopy(board)
47          row_or_col, start, end, is_row = action
48          if is_row:
49              new_board[row_or_col, start:end + 1] = 0
50          else:
51              new_board[start:end + 1, row_or_col] = 0
52
53          return new_board
```

Para probar el agente utilizamos el código en el archivo “tactix.ipynb”. Allí modificamos el código para que nuestro agente corra contra el Trainer y podamos evaluar los resultados.

```

# Run a series of games with different difficulties for the TrainerAgent
difficulties = [i / 10 for i in range(11)]
results_summary = []

for d in difficulties:
    agent1 = TrainerAgent(env, difficulty=d)
    agent2 = MinimaxTacTixAgent(env, depth=3)

    results = run_multiple_games(env, agent1, agent2, num_games=200)
    results_summary.append((d, results["agent1_wins"], results["agent2_wins"]))

```

Prueba 1

Creamos un agente con una función heurística básica que cuenta la cantidad de casillas vacías.

Heurística 1: heuristic_casillas_encendidas(board)

Cuenta cuántas casillas 1 quedan.

Cuantas más casillas libres quedan, más posibilidades de jugadas tenemos.

```

5   class MinimaxTacTixAgent(Agent):
74       def heuristic_utility(self, board):
75           # Heurística simple: cantidad de casillas encendidas
76           return np.sum(board)

```

Podemos ver con esta heurística que el nuestro agente no logró ganarle al entrenador:



Prueba 2

Probamos con la misma heurística pero con profundidad 4 y entendimos que con mayor profundidad el modelo mejora.



Prueba 3

Probamos tener 2 heurísticas ahora y combinarlas con una ponderación (inicialmente les damos el mismo peso a ambas, luego se pueden cambiar). A la función anterior le agregamos una función que contabiliza el “control” del agente sobre el juego considerando la cantidad de secuencias y la cantidad de fichas disponibles.

Heurística 2: heuristic_control(board)

Cuenta cuántas secuencias contiguas de 1 quedan en filas y columnas. Penaliza si hay muchas secuencias porque cuanto menos secuencias disponibles para jugar tenga, menos opciones va a tener para jugar. El control del tablero se mide por cómo podés reducir el espacio de jugadas posibles del otro jugador. Las combinamos en una combinación lineal.

```

83     def heuristic_control(board):
84         # Heurística de control: cuenta las secuencias de fichas encendidas
85         remaining = board.sum()
86         sequences = 0
87         size = board.shape[0]
88
89         # Contar secuencias en filas y columnas
90         for is_row in [0, 1]:
91             for idx in range(size):
92                 line = board[idx, :] if is_row else board[:, idx]
93                 in_sequence = False
94                 for val in line:
95                     if val == 1:
96                         if not in_sequence:
97                             sequences += 1
98                         in_sequence = True
99                     else:
100                         in_sequence = False
101
102         # Más fichas y menos secuencias es mejor (más control)
103         return remaining - sequences * 1.5
104
105     class MinimaxTacTixAgent(Agent):
106
107         def heuristic_utility(self, board):
108             # Heurística compuesta: combina varias heurísticas
109             return (1 * heuristic_casillas_encendidas(board) +
110                   1 * heuristic_control(board))
111
112         def heuristic_casillas_encendidas(board):
113             # Heurística simple: cantidad de casillas encendidas
114             return np.sum(board)
115
116         def heuristic_control(board):
117             # Heurística de control: cuenta las secuencias de fichas encendidas
118             remaining = board.sum()
119             sequences = 0
120             size = board.shape[0]
121
122             # Contar secuencias en filas y columnas
123             for is_row in [0, 1]:
124                 for idx in range(size):
125                     line = board[idx, :] if is_row else board[:, idx]
126                     in_sequence = False
127                     for val in line:
128                         if val == 1:
129                             if not in_sequence:
130                                 sequences += 1
131                             in_sequence = True
132                         else:
133                             in_sequence = False
134
135             # Más fichas y menos secuencias es mejor (más control)
136             return remaining - sequences * 1.5

```



Observamos que con esta nueva heurística los resultados mejoraron un poco aunque no significativamente.

Prueba 4

Agregamos múltiples heurísticas para que el agente tenga más criterios de análisis y recompensa/penalización. Agregamos 3 heurísticas más:

Heurística 3: heuristic_num_jugadas_posibles(board, agent)

Cuenta cuántas jugadas legales tiene disponible el jugador. Cuantas más jugadas disponibles tenga, más flexibilidad tiene. Si tiene pocas jugadas entonces el oponente está más cerca de quedar sin opciones.

```

108     def heuristic_num_jugadas_posibles(board, agent):
109         return len(agent.get_valid_actions(board))

```

Heurística 4: heuristic_control_centro(board)

Penaliza las casillas más alejadas del centro. El centro tiende a ofrecer más opciones, mientras que bordes y esquinas se agotan más fácil.

```

111 def heuristic_control_centro(board):
112     size = board.shape[0]
113     center = size // 2
114     value = 0
115     for i in range(size):
116         for j in range(size):
117             if board[i, j] == 1:
118                 # Penaliza cuanto más lejos del centro
119                 dist = abs(i - center) + abs(j - center)
120                 value -= dist
121     return value

```

Heurística 5: heuristic_cantidad_secuencias(board)

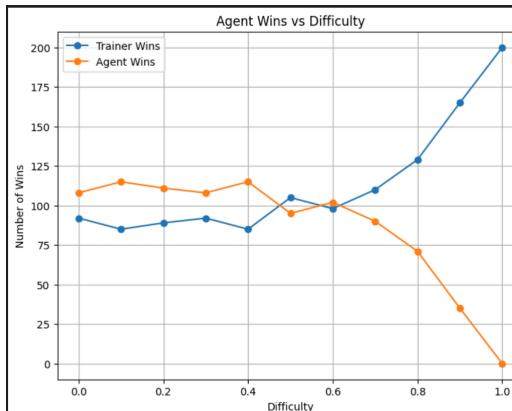
Penaliza la cantidad de secuencias contiguas en filas y columnas (similar a la #2 pero más simple). Cuanto menos secuencias haya, más "bloques grandes" habrán y se tendrá más control del tablero. Las combinamos todas en una gran combinación lineal de 5 heurísticas.

```

123 def heuristic_cantidad_secuencias(board):
124     size = board.shape[0]
125     sequences = 0
126     for is_row in [0, 1]:
127         for idx in range(size):
128             line = board[idx, :] if is_row else board[:, idx]
129             in_seq = False
130             for val in line:
131                 if val == 1:
132                     if not in_seq:
133                         sequences += 1
134                         in_seq = True
135                     else:
136                         in_seq = False
137     return -sequences # penaliza muchas secuencias

5 class MinimaxTacTixAgent(Agent):
74     def heuristic_utility(self, board):
75         # Heurística compuesta: combina varias heurísticas
76         return (1 * heuristic_casillas_encendidas(board) +
77                 1 * heuristic_control(board) +
78                 1 * heuristic_num_jugadas_posibles(board, self) +
79                 1 * heuristic_control_centro(board) +
80                 1 * heuristic_cantidad_secuencias(board))
81

```



Vimos que no había mucha mejora con respecto a la prueba anterior que tenía solo 2 heurísticas. A partir de esto pensamos que debíamos mejorar la fusión heurística al encontrar la combinación lineal más óptima. Además sospechamos que no todas las heurísticas que planteamos son buenas o útiles.

Prueba 5

Continuamos con las mismas heurísticas pero le damos más profundidad al árbol minimax (de profundidad 4 a 6).

```

class MinimaxTacTixAgent(Agent):
    def heuristic_utility(self, board):
        # Heurística compuesta: combina varias heurísticas
        return (1 * heuristic_casillas_encendidas(board) +
                1 * heuristic_control(board) +
                1 * heuristic_num_jugadas_posibles(board, self) +
                1 * heuristic_control_centro(board) +
                1 * heuristic_cantidad_secuencias(board))

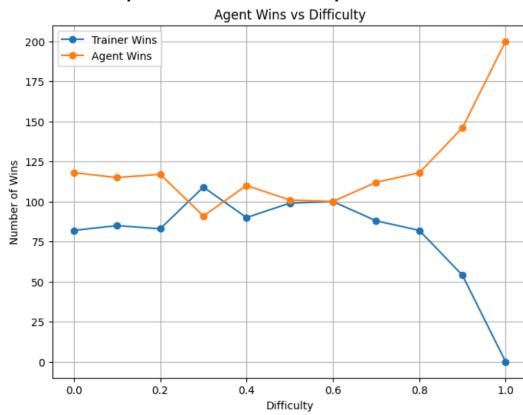
# Run a series of games with different difficulties for the TrainerAgent
difficulties = [i / 10 for i in range(11)]
results_summary = []

for d in difficulties:
    agent1 = TrainerAgent(env, difficulty=d)
    agent2 = MinimaxTacTixAgent(env, depth=6)

    results = run_multiple_games(env, agent1, agent2, num_games=200)
    results_summary.append((d, results["agent1_wins"], results["agent2_wins"]))

```

Con esta decisión logramos tener una gran mejora en la cual en la mayoría de los casos el agente gana. Nuevamente observamos que la profundidad del árbol afecta en gran medida ya que a pesar de que haber usado estas 5 heurísticas no nos había dado mucho resultado comparado con la función de 2 heurísticas, observamos que subiéndole la profundidad el agente ganaba más.



Prueba 6

Decidimos probar uno a uno las heurísticas para ver cómo rendían y ver cuáles son las mejores y en qué medida mejoran la performance del agente. Para eso definimos la profundidad en 3 para que las pruebas no tomen demasiado tiempo y poder comparar varias combinaciones.

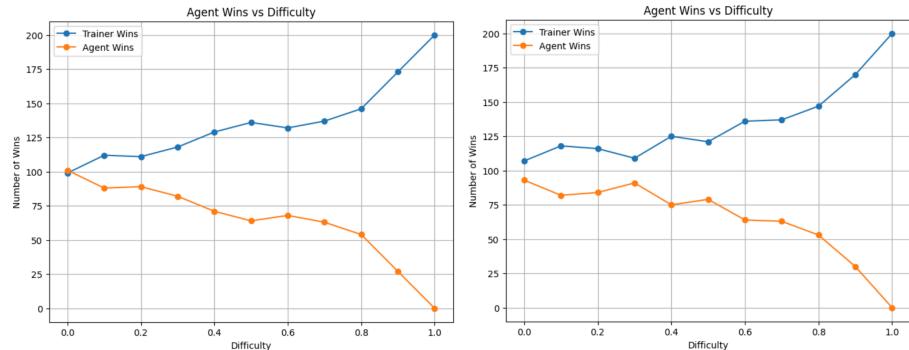
```
# Run a series of games with different difficulties for the TrainerAgent
difficulties = [i / 10 for i in range(11)]
results_summary = []

for d in difficulties:
    agent1 = TrainerAgent(env, difficulty=d)
    agent2 = MinimaxTacTixAgent(env, depth=3)

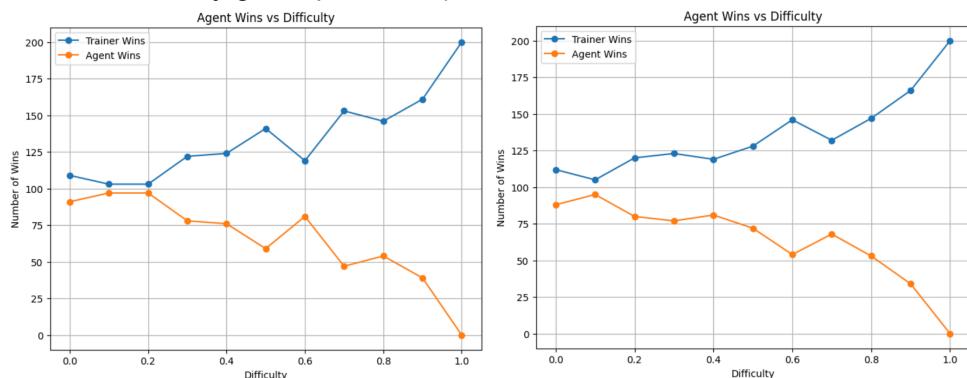
    results = run_multiple_games(env, agent1, agent2, num_games=200)
    results_summary.append((d, results["agent1_wins"], results["agent2_wins"]))
```

Pruebas individuales:

- 1) Heurística de casillas encendidas 2) Heurística de control de la tabla



- 3) Heurística de cantidad de jugadas posibles 4) Heurística de control del centro



- 5) Heurística de cantidad de secuencias del oponente



Observaciones: Parecería ser que las heurísticas 2 y 3 tuvieron un mejor desempeño frente a las otras ya que estuvieron más empatadas con el entrenado que las otras. Por eso podemos probar en darles más peso y ver qué tan performantes resultan.

```
class MinimaxTicTacToeAgent(Agent):
    def heuristic_utility(self, board):
        # Heurística compuesta: combina varias heurísticas
        return (0.5 * heuristic_casillas_encendidas(board) +
               1 * heuristic_control(board) +
               1 * heuristic_num_jugadas_posibles(board, self) +
               0.5 * heuristic_control_centro(board) +
               0.5 * heuristic_cantidad_secuencias(board))
```



Para poder comparar el resultado de la Prueba 4, en donde introducimos las 5 heurísticas sin ponderaciones, probamos este caso ponderado pero con profundidad 4.

Este fue el resultado comparado con el resultado de la Prueba 4:

Prueba 4:



Prueba 6:



Se puede ver que el resultado de la prueba 4 fue mejor porque tiene mayor cantidad de victorias. Por lo que parece ser que nuestra suposición de que las heurísticas 2 y 3 solas parecían buenas, no fue muy acertado.

Prueba 7

Para buscar la mejor combinación lineal de las heurísticas programamos un algoritmo que prueba todas las combinaciones lineales posibles para competir contra el agente en una dificultad determinada con una profundidad determinada y evalúa el porcentaje de éxito del agente en 200 partidas. Con esta prueba logramos hacer un gran salto en el rendimiento del agente ya que encontramos una combinación lineal de heurísticas que potenció enormemente el agente e hizo que ganara la mayoría de veces en todas las dificultades.

Los resultados fueron:

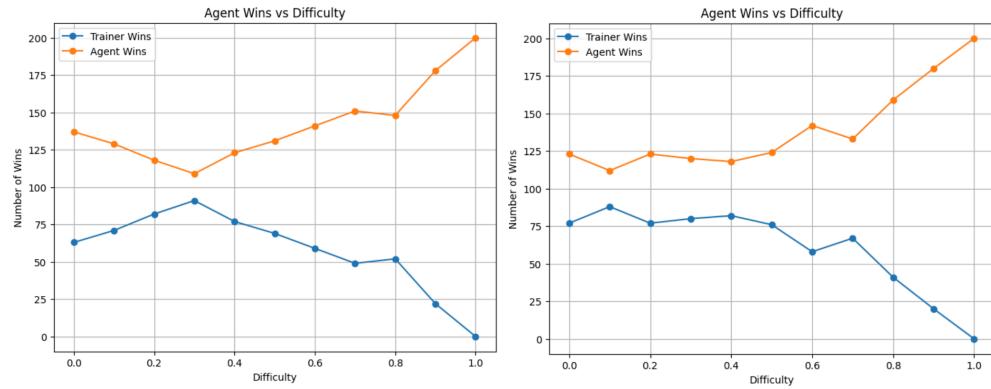
- 1) w1:0.0 w2:0.0 w3:0.5 w4:0.0 w5:0.5 ➤ Win Rate: 84.50%
- 2) w1:0.0 w2:1.0 w3:0.5 w4:1.0 w5:1.0 ➤ Win Rate: 81.00%
- 3) w1:0.0 w2:1.0 w3:1.0 w4:1.0 w5:1.5 ➤ Win Rate: 80.50%

```
w1:1.5 w2:1.5 w3:1.5 w4:1.0 w5:1.5 ► Win Rate: 67.00%
w1:1.5 w2:1.5 w3:1.5 w4:1.0 w5:0.0 ► Win Rate: 29.50%
w1:1.5 w2:1.5 w3:1.5 w4:1.0 w5:0.5 ► Win Rate: 55.00%
w1:1.5 w2:1.5 w3:1.5 w4:1.5 w5:0.5 ► Win Rate: 24.50%
w1:1.5 w2:1.5 w3:1.5 w4:1.5 w5:0.0 ► Win Rate: 26.50%
w1:1.5 w2:1.5 w3:1.5 w4:1.5 w5:1.0 ► Win Rate: 23.50%
w1:1.5 w2:1.5 w3:1.5 w4:1.5 w5:1.5 ► Win Rate: 23.50%

➊ Mejor combinación encontrada:
casillas: 0.0
control: 0.0
jugadas: 0.5
centro: 0.0
secuencias: 0.5
✓ Win rate: 84.50%
```

Para visualizar mejor estas heurísticas en la práctica probamos evaluar el modelo con cada una de las 3 mejores combinaciones lineales en el entorno de 200 juegos y árbol de profundidad 4 comparando con 10 distintas dificultades del agente.

1) $w1:0.0 w2:0.0 w3:0.5 w4:0.0 w5:0.5$ 2) $w1:0.0 w2:1.0 w3:0.5 w4:1.0 w5:1.0$



3) $w1:0.0 w2:1.0 w3:1.0 w4:1.0 w5:1.5$

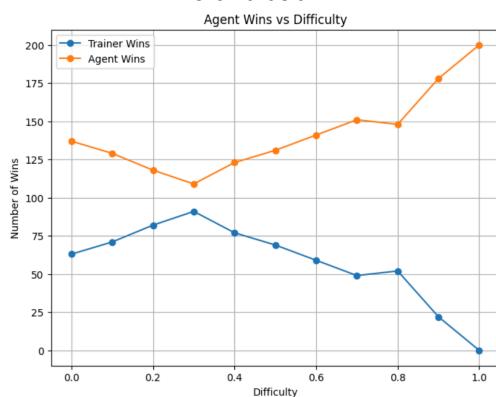


Con las 3 opciones logramos que el agente le gane siempre al entrenador. Sin embargo, yendo más al detalle, las 3ra combinación lineal tiene menor diferencia de ganado que las otras dos. Las dos primeras están muy empatadas en rendimiento pero la primera tiene una leve mejora.

Prueba 8

Para intentar mejorar el agente aún más, fuimos profundizando el árbol. Corrimos ahora con la combinación lineal (0, 0, 0.5, 0, 0.5) y profundidad 6. Lo comparamos con el de profundidad 4 y vimos que el de profundidad 6 performance mejor.

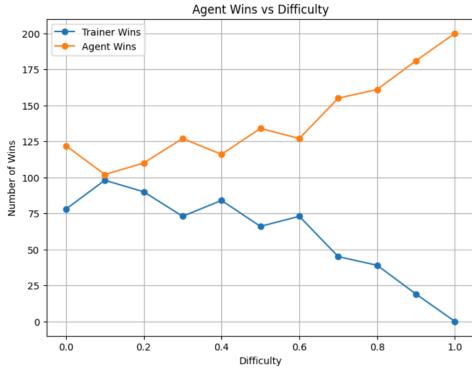
Profundidad 4:



Profundidad 6:



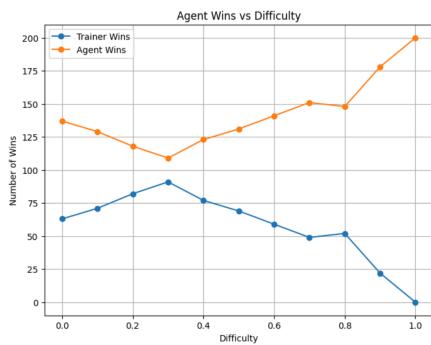
Profundidad 8 performance peor:



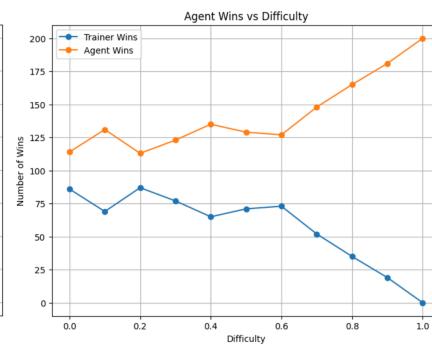
Prueba 9

Ya que la combinación lineal 1 y 2 fueron muy parecidas, decidimos probar la performance del agente con esta segunda función (0, 1, 0.5, 1, 1) también para comparar resultados. Comenzamos comparando la performance de este agente con profundidad 4 con profundidad 6:

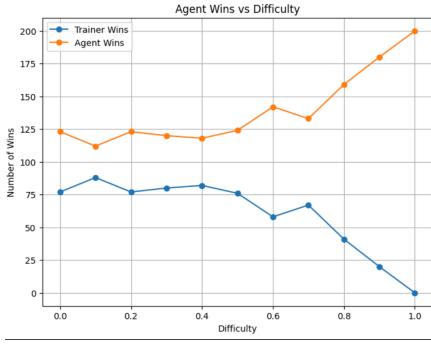
Profundidad 4:



Profundidad 6:



CL 1:



CL 2:

Finalmente probamos con profundidad 8:



Conclusión

Para elegir el mejor modelo de Minimax exploramos diferentes heurísticas, combinaciones lineales y profundidades. Pasamos de tener un agente que no lograba ganarle al entrenador, a elegir el que mejor se desempeña entre tres agentes que sí le ganan. Finalmente decidimos quedarnos con el agente minimax de función heurística (0, 0, 0.5, 0, 0.5) y profundidad 6.

Agente minimax

- Profundidad: 6
- Función heurística:

```

def heuristic_utility(self, board):
    # Heurística compuesta: combina varias heurísticas
    return (0 * heuristic_casillas_encendidas(board) +
           0 * heuristic_control(board) +
           0.5 * heuristic_num_jugadas_posibles(board, self) +
           0 * heuristic_control_centro(board) +
           0.5 * heuristic_cantidad_secuencias(board))

def heuristic_num_jugadas_posibles(board, agent):
    return len(agent.get_valid_actions(board))

def heuristic_cantidad_secuencias(board):
    size = board.shape[0]
    sequences = 0
    for is_row in [0, 1]:
        for idx in range(size):
            line = board[idx, :] if is_row else board[:, idx]
            in_seq = False
            for val in line:
                if val == 1:
                    if not in_seq:
                        sequences += 1
                        in_seq = True
                    else:
                        in_seq = False
    return -sequences # penaliza muchas secuencias

```

- Performance:



Táctica 2: Expectimax

Para la segunda táctica implementamos el algoritmo Expectimax, creando el archivo `expectimax_agent.py`. Este agente hereda de la clase base `Agent.py`, la cual define la firma de la función `heuristic_utility` que Expectimax reutiliza igual que Minimax.

Para mantener la comparación justa con el agente Minimax, se utilizó exactamente la misma heurística definida en `Agent.py` y las mismas funciones de utilidad. Los hiperparámetros explorados fueron profundidad (depth) y cantidad de partidas por prueba.

El agente utiliza las mismas funciones de generación de acciones (`get_valid_actions`), simulación de movimientos (`simulate_move`) y evaluación de tablero (`heuristic_utility`).

Diferencia clave respecto a Minimax:

A diferencia de Minimax, Expectimax NO usa poda alfa-beta. En lugar de asumir que el oponente juega de forma perfecta (nodo MIN), Expectimax calcula la expectativa: promedia todos los posibles movimientos del rival, modelándolo como un jugador aleatorio. Esto hace que el árbol de estados crezca mucho más rápido el tiempo de cómputo aumente exponencialmente y que a mayores profundidades se vuelva impráctico correr grandes cantidades de partidas sin optimizaciones.

Pruebas realizadas

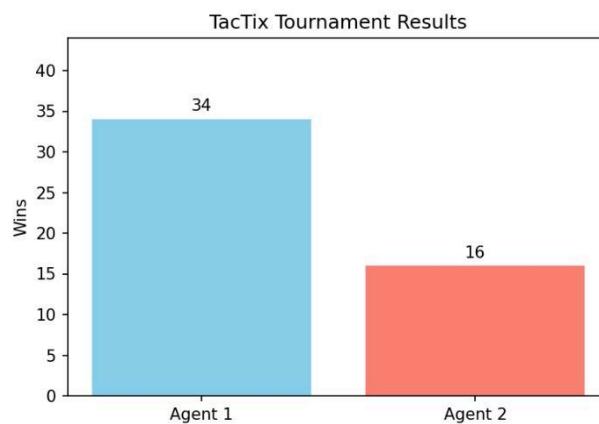
A continuación se muestran las distintas pruebas realizadas para validar el desempeño del agente Expectimax enfrentando al `TrainerAgent`.

Para cada caso se especifica la profundidad, la cantidad de partidas y el resultado obtenido.

Prueba 1 — Profundidad 3 — 50 partidas

- **Resultado:** Expectimax: 34 victorias TrainerAgent: 16 victorias

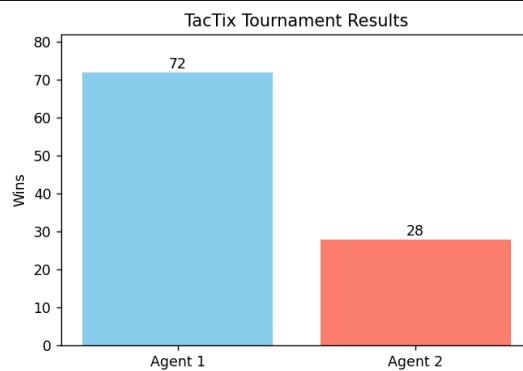
```
> python play.py
Resultados: {'agent1_wins': 34, 'agent2_wins': 16}
```



Prueba 2 — Profundidad 3 — 100 partidas

- **Resultado:** Expectimax: 72 victorias TrainerAgent: 28 victorias

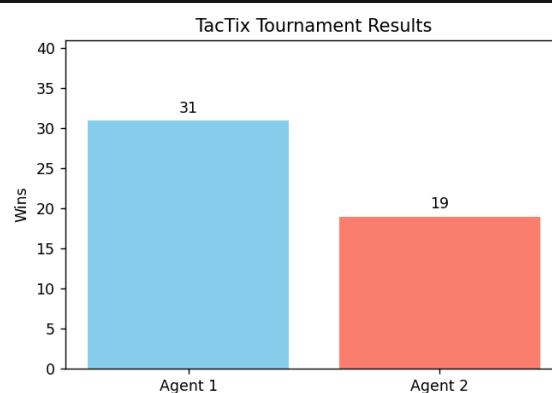
```
PS C:\Users\FiorellaAlvira\Downloads\InteligenciaArtificial (2)\InteligenciaArtificial\tactix> python play.py
Resultados: {'agent1_wins': 72, 'agent2_wins': 28}
```



Prueba 3 — Profundidad 4 — 50 partidas

- **Resultado:** Expectimax: 31 victorias TrainerAgent: 19 victorias

```
PS C:\Users\FiorellaAlvira\Downloads\InteligenciaArtificial (2)\InteligenciaArtificial\tactix> python play.py
Resultados: {'agent1_wins': 31, 'agent2_wins': 19}
```



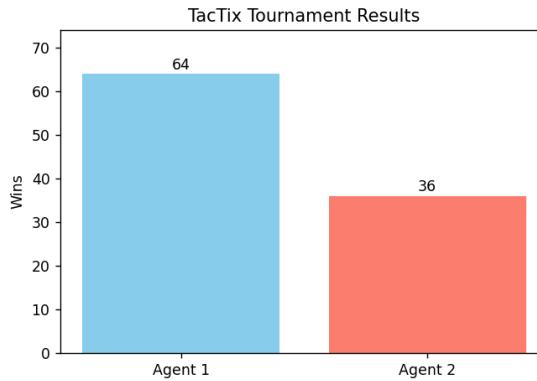
Prueba 4 — Profundidad 4 — 100 partidas

- **Resultado:** Expectimax: 64 victorias TrainerAgent: 36 victorias

```

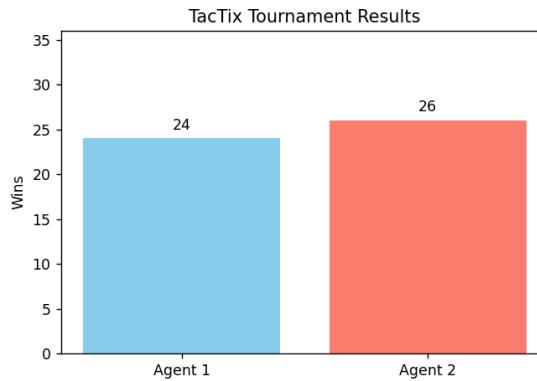
Resultados: {'agent1_wins': 64, 'agent2_wins': 36}
PS C:\Users\FiorellaAlvira\Downloads\InteligenciaArtificial (2)\InteligenciaArtificial\tactix> python play.py

```



Prueba 5 — Profundidad 5 — 50 partidas

- **Resultado:** Expectimax: 24 victorias TrainerAgent: 26 victorias



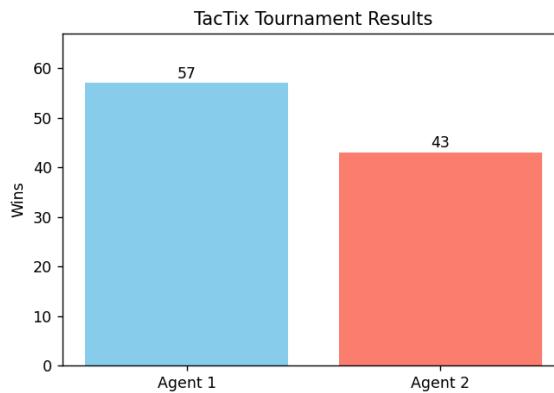
Prueba 6 — Profundidad 5 — 100 partidas

- **Resultado:** Expectimax: 57 victorias TrainerAgent: 43 victorias

```

PS C:\Users\FiorellaAlvira\Downloads\InteligenciaArtificial (2)\InteligenciaArtificial\tactix> python play.py
Resultados: {'agent1_wins': 57, 'agent2_wins': 43}

```



Observaciones

- Se observa que a mayor profundidad, Expectimax tiende a mejorar su rendimiento frente al TrainerAgent, aunque en profundidades altas sin poda el costo computacional se vuelve prohibitivo.
- El caso de Profundidad 5 muestra que, sin poda ni ajuste fino de la heurística, el rendimiento no siempre mejora de forma lineal.
- La prueba de Profundidad 6 fue descartada debido al excesivo tiempo de cómputo, lo que evidencia que Expectimax sin poda se vuelve inviable para árboles muy profundos.

A continuación se muestra la tabla resumen de todas las pruebas realizadas para el agente Expectimax, indicando para cada profundidad el número de partidas, resultados obtenidos frente al TrainerAgent y la duración estimada de cada ejecución. Esto evidencia la relación directa entre profundidad, costo computacional y rendimiento del agente.

Profundidad	Partidas	Expectimax	Trainer	Duración aprox.
3	50	34	16	~2-3 min
3	100	72	28	~5-7 min
4	50	31	19	~15-20 min
4	100	64	36	~30-40 min
5	50	24	26	~12 h
5	100	57	43	~21 h

Análisis y conclusión

Las figuras muestran la evolución de las victorias del agente Expectimax frente al TrainerAgent, diferenciando escenarios con 50 y 100 partidas.

Se observa que al aumentar la profundidad, Expectimax mejora su desempeño, aunque no de forma estrictamente lineal, lo que evidencia la influencia de la heurística y la ausencia de poda. En profundidades altas el costo computacional se vuelve impráctico, reforzando la diferencia con Minimax, que optimiza la búsqueda mediante poda alfa-beta.

Con estos resultados se valida que el agente Expectimax cumple con lo solicitado: utiliza funciones heurísticas, simula movimientos posibles y ajusta su decisión maximizando la expectativa frente a un rival aleatorio.

Se observa que al aumentar tanto la profundidad como la cantidad de partidas, el rendimiento de Expectimax mejora de forma más estable y consistente. Las pruebas con 100 partidas mostraron que Expectimax superó al TrainerAgent en todos los casos, mientras que en las pruebas con 50 partidas se registró una derrota aislada.

Esto evidencia que utilizar un mayor número de partidas permite obtener evaluaciones más robustas y reduce la variabilidad en los resultados, especialmente en algoritmos sin poda como Expectimax, donde el comportamiento puede ser más irregular frente a muestras pequeñas.

Se adjuntan las gráficas, la tabla de resultados y todas las capturas que evidencian el rendimiento probado.

