



ALGORITMOS GENÉTICOS **APRENDIZAJE DEL PACMAN**

Julieta Brizuela - Thiago Laleggia - Mateo Santos - Florencia Soria



INTRODUCCION



¿POR QUÉ ELEGIMOS PAC-MAN Y ALGORITMOS GENÉTICOS?

Motivo de elegir Pac-Man:

- Juego icónico, sencillo y desafiante.
- Oportunidad para optimizar la estrategia de Pac-Man para conseguir la mayor cantidad de puntos

¿Por qué algoritmos genéticos?

- Optimización adaptativa del comportamiento de Pac-Man.
- Los algoritmos genéticos permiten que Pac-Man aprenda a evadir de manera más efectiva.



CONTEXTUALIZACIÓN DEL ALGORITMO GENÉTICO

¿Qué es un algoritmo genético?

- Técnica que simula la evolución natural para encontrar soluciones óptimas.
- Procesos de selección, cruzamiento y mutación para crear estrategias efectivas.

Aplicación al juego:

- Pac-Man evoluciona su estrategia de evasión frente a los fantasmas.
- Ciclo de evolución: Selección → Cruzamiento → Mutación → Optimización.



DESAFÍOS DEL JUEGO Y DEL ALGORITMO

Desafíos en el juego:

- Los fantasmas tienen patrones predecibles, lo que hace que Pac-Man deba aprender a anticipar y escapar de manera más inteligente.
- Dinámica de evasión en tiempo real, ajustándose a la ubicación y el movimiento de los fantasmas.

Desafíos del algoritmo:

- Complejidad computacional: la evolución de las estrategias de evasión requiere muchos cálculos.
- Selección de parámetros para la evolución de Pac-Man: tasa de mutación, diversidad de estrategias y tiempos de adaptación.

GENOTIPO

El genotipo en un algoritmo genético representa la estructura interna que codifica la solución de un problema. En este caso, el genotipo es una representación de los pesos de una red neuronal, que se inicializan aleatoriamente en la población. Cada individuo en la población tiene su propio conjunto de pesos, que corresponden a las conexiones entre las capas de la red neuronal.

Cada conjunto de pesos representa el "genotipo" de un individuo, que será modificado a través de las generaciones mediante selección, cruce y mutación.

```
layer_1 = np.zeros((12, 32))
layer_2 = np.zeros((32, 16))
layer_3 = np.zeros((16, 8))
layer_4 = np.zeros((8, 4))

for i in range(population_size):
    weights[i] = [np.random.randn(*layer_1.shape) * np.sqrt(1 / layer_1.shape[0]),
                  np.random.randn(*layer_2.shape) * np.sqrt(1 / layer_2.shape[0]),
                  np.random.randn(*layer_3.shape) * np.sqrt(1 / layer_3.shape[0]),
                  np.random.randn(*layer_4.shape) * np.sqrt(1 / layer_4.shape[0])]
```




FENOTIPO

El fenotipo es la manifestación externa de la solución que resulta de aplicar el genotipo. En este contexto, el fenotipo se refiere a un agente de Pacman, cuyas decisiones de movimiento están influenciadas por la red neuronal entrenada. La red neuronal recibe las distancias de los fantasmas y la situación actual del juego como entradas, y en base a estos datos, genera la dirección en la que el personaje debe moverse.

El agente Pacman, como fenotipo, está compuesto por atributos como:

- Posición (x, y)
- Velocidad (velX, velY)
- Historial de movimiento
- Animaciones de los movimientos (izquierda, derecha, arriba, abajo)

La evolución del genotipo mejora las decisiones del fenotipo, ajustando los movimientos de Pacman para optimizar su rendimiento en el juego.



PROBLEMAS Y SOLUCIONES

PROBLEMAS Y SOLUCIONES

Durante el desarrollo del agente de Pacman utilizando algoritmos genéticos, nos enfrentamos a varios desafíos que afectaron el rendimiento del agente. Estos problemas estuvieron relacionados con el comportamiento del agente, la selección de padres en el algoritmo y la forma en que evaluábamos su rendimiento.

1

**MOVIMIENTO DEL
PACMAN**

2

**ESTANCAMIENTO EN
BUCLE LOCAL**

3

**SELECCIÓN DE PADRES
INEFICIENTE**

4

**FUNCIÓN DE APTITUD
INADECUADA**

5

MUTACIÓN

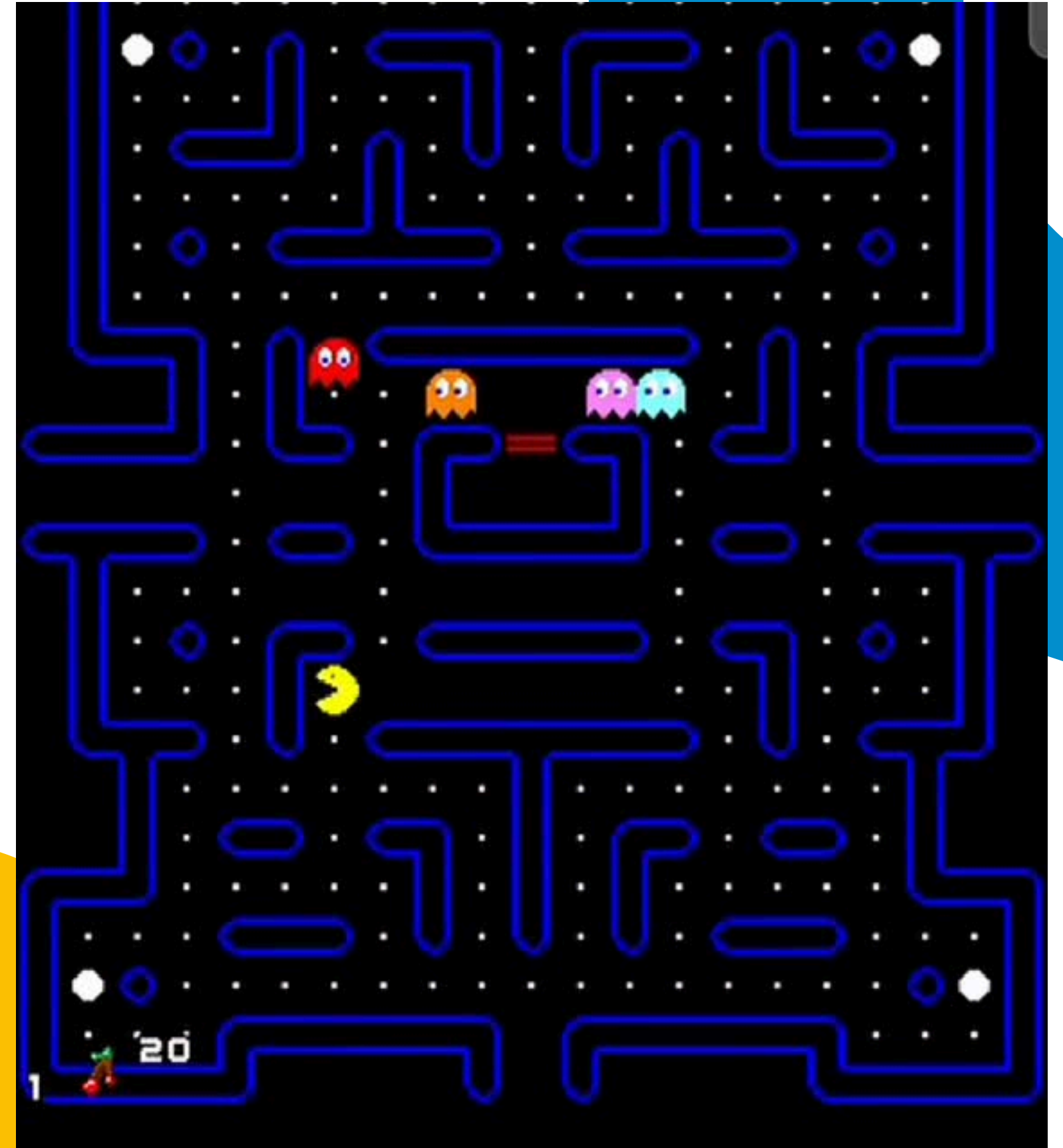
MOVIMIENTO DEL PACMAN

PROBLEMA

Durante las primeras etapas, notamos que el Pacman a veces se quedaba atrapado en una esquina, sin realizar ninguna acción. Observando con mas detenimiento, encontramos que en ocasiones la red neuronal elegía un movimiento en dirección a una pared.

SOLUCION

Agregamos un evento que detecte que el pacman está detenido para que la red neuronal intente elegir un nuevo movimiento.



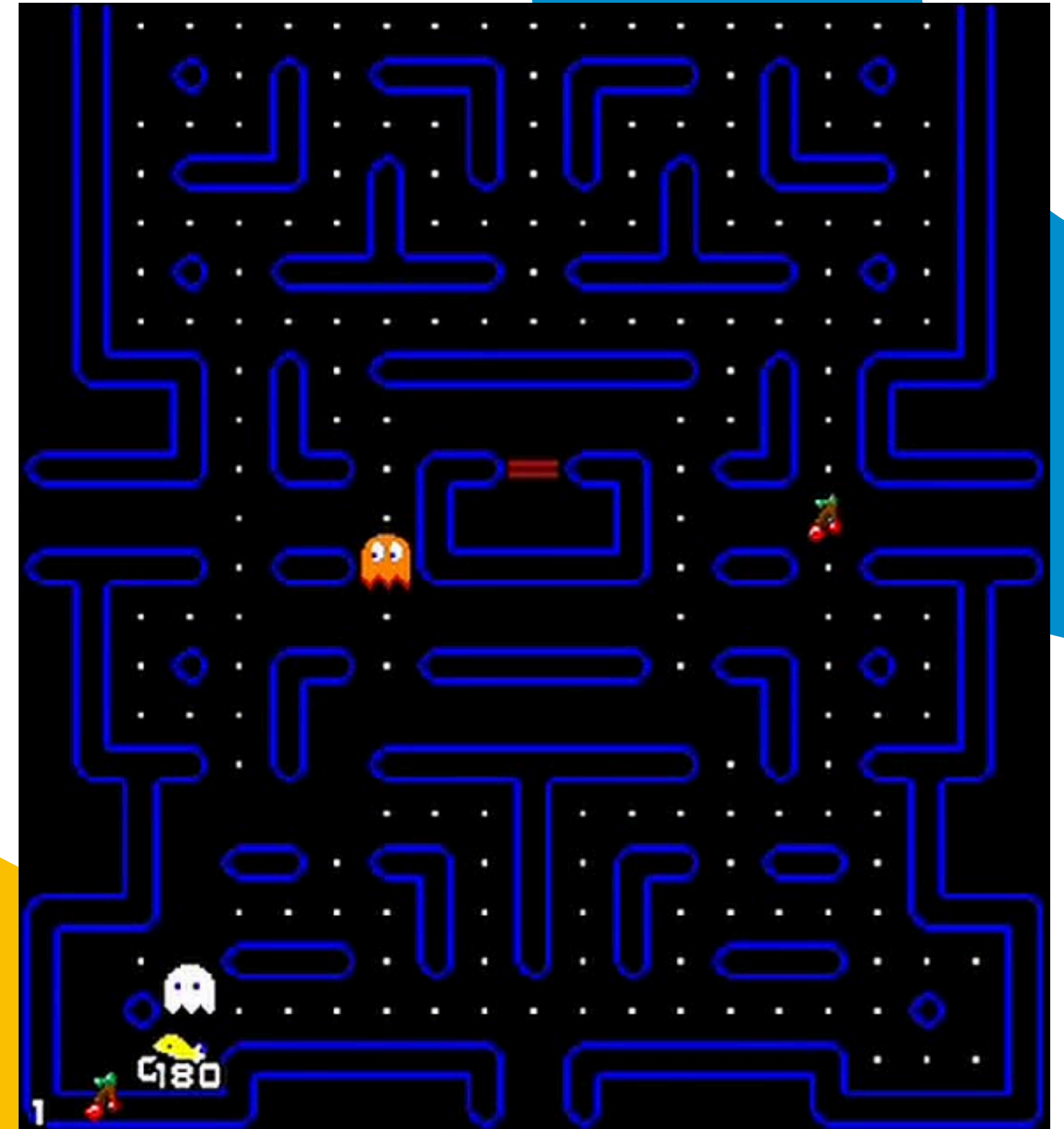
ESTANCAMIENTO EN BUCLE LOCAL

PROBLEMA

Durante la evolución, observamos que el Pacman a veces se quedaba atrapado en una esquina, moviéndose en los mismos 3 casilleros. Este comportamiento afectaba negativamente su rendimiento, ya que, al no salir, no podía recolectar más puntos ni evitar a los fantasmas.

SOLUCION

Ajustamos la selección de padres y la forma en que el algoritmo generaba las estrategias, priorizando aquellas que ayudaban a Pacman a mantenerse en movimiento. Esto evitó que se quedara atrapado en esquinas y le permitió mejorar su rendimiento en las siguientes generaciones.



SELECCIÓN DE PADRES INEFICIENTE

PROBLEMA

Inicialmente, adoptamos un enfoque en el que se seleccionaban los padres de manera equitativa, es decir, un 50% de un padre y un 50% del otro. Sin embargo, este enfoque no siempre daba buenos resultados, ya que en algunas ocasiones se heredaban las peores características de ambos padres, lo que resultaba en una descendencia menos eficiente.

SOLUCION

Optamos por una selección de padres más adaptativa, donde los padres más exitosos (según el puntaje) tenían más probabilidades de ser seleccionados, y las características de ambos se ponderaban en lugar de seleccionar la mitad de cada uno. Esto mejoró el rendimiento de la descendencia y ayudó a evitar que las estrategias ineficientes se propagaran.

FUNCIÓN DE APTITUD INADECUADA

PROBLEMA

En un principio, la función de aptitud consideraba tanto el puntaje como el tiempo de supervivencia. Sin embargo, este enfoque causaba que Pacman, al estar evitando a los fantasmas, pudiera quedarse en una esquina sin moverse, lo que aumentaba artificialmente su tiempo de supervivencia, sin que realmente mejorara su rendimiento.

SOLUCION

Decidimos simplificar la función de aptitud para que solo se basara en el puntaje obtenido durante la partida. Esto incentivó a Pacman a moverse y recolectar puntos activamente, en lugar de simplemente sobrevivir, lo que resultó en una evolución más efectiva de su comportamiento.

MUTACIÓN

PROBLEMA

Conforme perfeccionamos el funcionamiento de la red neuronal, y las primeras generaciones fueron mas exitosas, notamos que las generaciones siguientes no mejoraban. Por el contrario, vimos que el fitness máximo era cada vez más bajo.

SOLUCION

Implementamos varias estrategias en la mutación, que garantizaran que la evolución de los individuos mejore.

Estas técnicas fueron:

- Ya no mutamos al 100% de los nuevos individuos
- Mutación Suave con Desviación Estándar: No se asignan valores totalmente nuevos
- Mutación Proporcional: cada peso del genotipo tiene una ligera mutación



DESARROLLO FINAL



**DEPURACIÓN
EFICIENTE Y CLARA**

RED NEURONAL ARQUI

```
#np.random.uniform(layer_2)
layer_3 = np.zeros((26, 13)) #((capa entrada, capa salida))
#np.random.uniform(layer_3)
layer_4 = np.zeros((13, 4))
#np.random.uniform(layer_4)
...
layer_1 = np.zeros((12, 32)) # Primera capa con 14 nodos de entrada la
layer_2 = np.zeros((32, 16)) # Segunda capa con 20 nodos layer_2: [[0.
layer_3 = np.zeros((16, 8)) # Tercera capa con 14 nodos layer_3: [[0.
layer_4 = np.zeros((8, 4)) # Cuarta capa con 4 nodos de salida lay

for i in range(population_size):
    weights[i] = [np.random.randn(*layer_1.shape) * np.sqrt(1 / layer_1.

weights = {dict: 3} {0: [[[-0.40602992  0.62519467 -0.01326395  0.18492943  0.17568592  0.22250956, -0.20701232 -0.28745212  0.28950197...
0 = {list: 4} [[[-0.40602992  0.62519467 -0.01326395  0.18492943  0.17568592  0.22250956, -0.20701232 -0.28745212  0.28950197...
> 0 = {ndarray: (12, 32)} [[[-0.40602992  0.62519467 -0.01326395  0.18492943  0.17568592  0.22250956, -0.20701232 -0.2...View as A
> 1 = {ndarray: (32, 16)} [[[-1.99077428e-01  6.01672236e-02 -6.87332562e-02 -1.00974122e-01,  1.42187750e-01  3.45948...View as A
> 2 = {ndarray: (16, 8)} [[[-0.37205711 -0.13831369  0.41101523 -0.08330546  0.36789916 -0.16588255,  0.09276768  0.2118...View as A
> 3 = {ndarray: (8, 4)} [[ 0.26120705 -0.27452504  0.34460181 -0.08608604], [ 0.69930724 -0.39963143  0.05420848 -0.16...View as A
10 __len__ = (int) 4
def soft
sum_
# pr
```

Weights almacena los pesos de cada uno de nuestros individuos en sus diferentes capas, resultando en un vector (c/ individuo) de vectores (c/capa de c/individuo)

Estos **pesos** son aquellos que forman los **cimientos de nuestra red** es decir la estructura del cerebro

LAS MATEMÁTICAS DECIDEN NUESTRO MOVIMIENTO

```
def neural_net(sample, input_ga): #no se usa 2 usages      input_ga

    temp2 = relu(np.matmul(*args: input_ga.T, weights[sample][0]))
    temp3 = relu(np.matmul(*args: temp2, weights[sample][1]))
    temp4 = relu(np.matmul(*args: temp3, weights[sample][2]))    temp4:
    out = softmax(np.matmul(*args: temp4, weights[sample][3]))

    # print(f"i am out {out}")
    ind = np.argmax(out)    #printea el valor mas alto de

r
```

La red recibe parametros: pesos e input_ga. En base a esto nos dira cual es nuestra mejor siguiente jugada. Pero qué es input_ga ??

Un **vector** que nos **describe nuestro entorno** en un momento dado

EXPLOREMOS EL GAINPUT

```
def GAIInput(ghostDistance, playerinput): 3 usages      ghostDistance:      playerinput:
# Normalizar las distancias a los fantasmas
walls = GetWallInput() walls:

ghostDistance = [float(i) / 30 for i in ghostDistance]
orientacion = direction_to_one_hot(playerinput.current_direction) orientacion:
|

# Crear el vector de entrada, incentivando exploración
inputValues = walls + ghostDistance + orientacion inputValues:

inputArray = np.asarray(inputValues) inputArray:

return inputArray

> {ndarray: (12,)} [0. 0. 1. 1. 0.36666667 0.8, 0.53333333 0.13333333 1. 0. 0. 0. ]..
```

Es el **traductor** personal de la red neuronal. Es quien coordina y recolecta toda la información del entorno en un simple **vector normalizado**.

Utiliza la **distancia** de los fantasmas, la **direccion** en la que esta viendo el pacman y en que lados hay **pared** (en torno a nuestro jugador)

TODOS NOS EQUIVOCAMOS. AYUDEMOSLA!!!



```
class pacman(): 2 usages
    def update_movement(self): 1 usage

        # Calcula la posición resultante con el movimiento sugerido
        new_x = self.x + temp_velX
        new_y = self.y + temp_velY

        # Verifica si el nuevo movimiento intercepta un muro
        if not thisLevel.CheckIfHitWall(possiblePlayerX_possiblePlayerY: (new_x, new_y), row_col: (
            # Si no intercepta un muro, aplica las velocidades
```

Durante el entrenamiento la ia sugeria movimientos invalidos, impidiendole poder seguir intentando jugar.

Así que decidimos ayudarla, sugiriendole en tiempo de juego que hay **mejores opciones** y que lo intente nuevamente.

Aún no es tarde los fantasmas no te atrapan!!



FITNESS FUNCTION

El **fitness** refleja la calidad de la solución considerando el puntaje, con algunas modificaciones comparándolo con la puntuación normal en un juego de pacman.

El puntaje obtenido en el juego se suma directamente al fitness.
Sin embargo, como nuestro objetivo es que la IA complete el nivel sin importar el puntaje real, modificamos la forma en la que se obtiene el puntaje.

Eliminamos la obtención de puntos al consumir fantasmas y frutas, mientras que modificamos el valor del consumible que le da poder a pacman para que valga lo mismo que cualquier otro punto consumible.

Además, para mantener los valores del fitness entre 0 y 1 dividimos el fitness por la cantidad máxima de puntos posible en el juego después de las modificaciones.

FITNESS FUNCTION

62

1 usage

63

```
def cal_pop_fitness(score, time, ghosDistance):
```

64

65

```
    return score/1960
```

66

67

68

69

70

71

72

73

74



CROSSOVER

La función crossover combina los "genes" (pesos de la red neuronal) de dos padres para generar nuevos hijos:

- La función crossover selecciona dos padres aleatorios de la población, asegurándose de que no sean iguales.
- Para cada capa de la red neuronal, se promedian los pesos de los dos padres.
- Los pesos cruzados de cada capa se someten a una pequeña mutación para introducir variabilidad.
- Se repite este proceso para todas las capas de la red neuronal, formando así la red de cada hijo.
- Los nuevos hijos se almacenan en una lista y reemplazan los pesos de la generación anterior, creando una nueva población para la siguiente iteración del algoritmo evolutivo.

CROSSOVER

```
def crossover(mating_pool):
    global weights
    global population_size
    new_children_weights = dict()
    for child in range(population_size):
        child_net = []
        p1 = random.choice(mating_pool)
        p2 = random.choice(mating_pool)
        while np.array_equal(p1, p2):
            p2 = random.choice(mating_pool)
        for layer in range(4):
            p1Matrix = p1[layer]
            p2Matrix = p2[layer]

            crossed = ((p1Matrix + p2Matrix) / 2)
            crossed = mutation(crossed, percent_mutation=0.01)

            child_net.append(crossed)

        new_children_weights[child] = child_net
    weights = new_children_weights
```




MUTACION

La mutación se aplica con una probabilidad del 90% al individuo. Es decir, en un 10% de los casos, el individuo no se muta, preservando a los mejores.

En lugar de cambiar el valor de cada peso a un valor completamente nuevo, aplicamos una mutación suave, donde utilizamos una desviación estándar, que depende de las dimensiones de la cantidad de nodos y capas de la red neuronal.

La mutación se aplica con una probabilidad definida por `percent_mutation`, y aplicamos una multiplicación proporcional en cada nodo para realizar ajustes suaves.

Finalmente, utilizamos una matriz de factores aleatorios entre 0.95 y 1.05, para que ajuste cada peso ligeramente.

MUTACION

```
def mutation(offspring_crossover, percent_mutation):
    num_rows, num_cols = offspring_crossover.shape
    variance = 2 / (num_rows + num_cols)
    std_dev = np.sqrt(variance)

    if np.random.rand() > 0.1:
        for i in range(num_rows):
            for j in range(num_cols):
                if np.random.rand() < percent_mutation:
                    offspring_crossover[i, j] += np.random.normal(0, std_dev)

    multiplier = np.random.uniform(0.95, 1.05, size=(num_rows, num_cols))
    offspring_crossover *= multiplier

    return offspring_crossover
```



**EXPLOTEMOS LOS
SERVIDORES DE
GOOGLE**



CONCLUSION

LECCIONES APRENDIDAS

A través de los desafíos enfrentados, entendimos mejor cómo optimizar el algoritmo genético y mejorar la adaptación del pacman, identificando áreas clave para futuros ajustes y mejoras.

IMPORTANCIA DE LA FUNCIÓN DE APTITUD

Una función de aptitud bien diseñada es clave para el rendimiento del agente, ya que influye directamente en sus decisiones y comportamiento.

AJUSTE DE PARÁMETROS GENÉTICOS

La tasa de mutación, el tamaño de la población y la selección de padres requieren ajustes para obtener mejores resultados en la evolución del agente.

DESAFÍOS DEL APRENDIZAJE AUTÓNOMO

El agente no siempre aprende de manera eficiente, especialmente cuando enfrenta situaciones complejas como quedarse atrapado en esquinas o no evitar los fantasmas de manera efectiva.

MEJORAS FUTURAS

Optimización de parametros

Utilizar otras estrategias de selección de padres, como selección por torneo o ruleta, para mejorar la diversidad genética y acelerar el proceso de evolución.

Pruebas con Diferentes Mapas



Probar diferentes mapas del juego, para evaluar cómo el pacman se adapta a nuevas configuraciones y mejorar su rendimiento en contextos variados.

Optimización de Redes Neuronales

Refinar la integración de redes neuronales para mejorar aún más la toma de decisiones del pacman, permitiendo que aprenda de manera más eficiente y adaptativa en tiempo real.

Incorporar Nuevas Estrategias

Implementar algoritmos adicionales para optimizar la evasión de fantasmas y mejorar el recorrido en situaciones más complejas del juego.

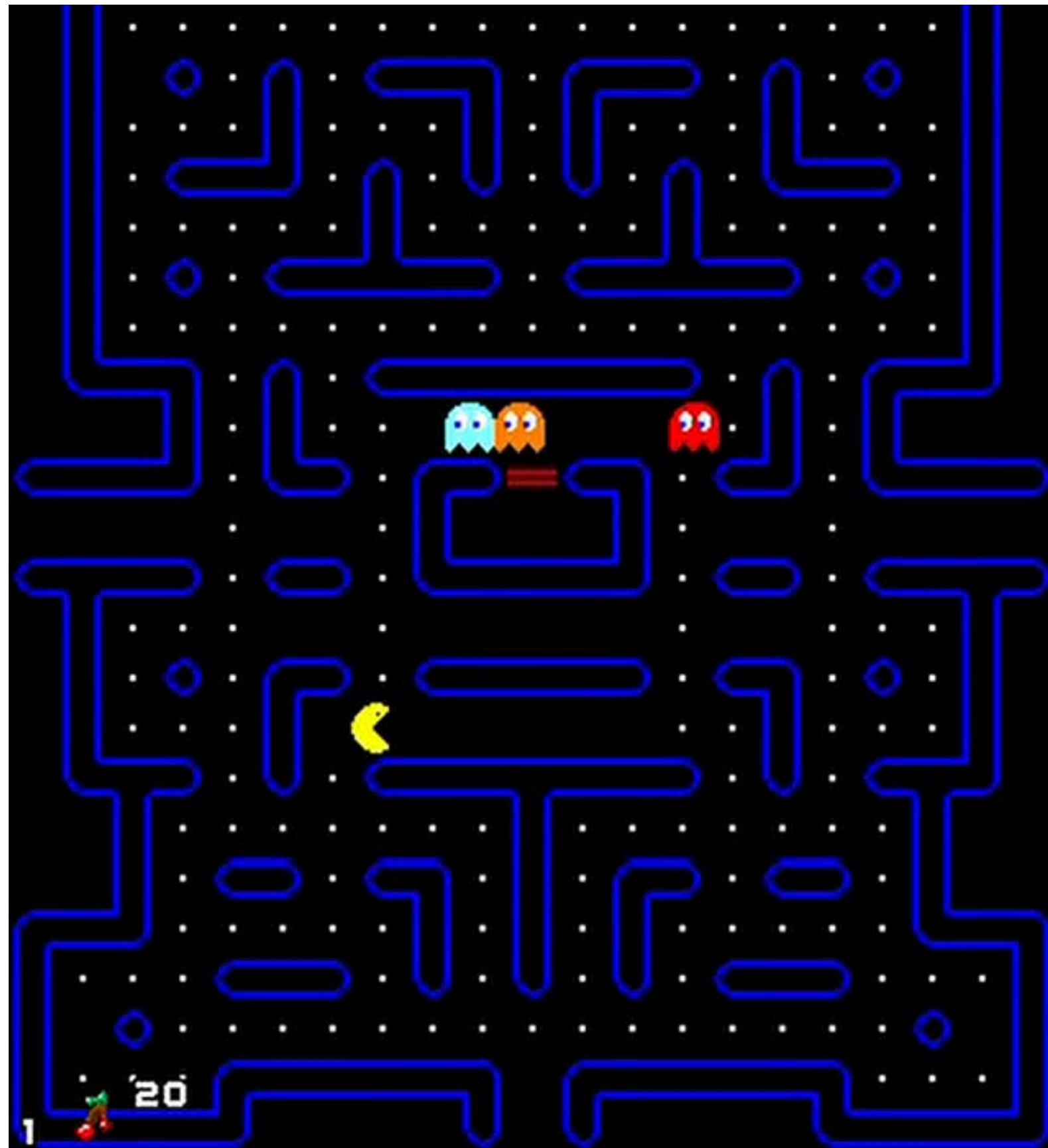


Logramos aplicar un algoritmo genético, permitiendo que el pacman mejorara su rendimiento generación tras generación. A pesar de enfrentar algunos desafíos, como el hecho de que el pacman a veces no lograba escapar de los fantasmas o quedaba atrapado en esquinas, encontramos soluciones para optimizar su comportamiento. Sin embargo, aún hay varias áreas de mejora, como la toma de decisiones más complejas, y los métodos de selección de los padres podríamos elegir uno más óptimo.



DEMOSTRACION

GENERACIÓN 2



GENERACIÓN 3

