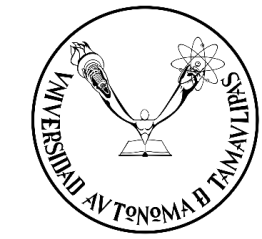
Universidad Autónoma de Tamaulipas
Facultad de Ingeniería Tampico



ASIGNATURA
Automatización y Robótica

9no. Semestre – Grupo “J”
2025 -3

TRABAJO
Proyecto Integrador

Docente: Dr. García Ruiz Alejandro H.

Integrantes:
Aldama Trinidad Alfonso Rene
Clemente Villegas José Adán
Cristóbal Francisco Jesús Marcelino
Domínguez Reyes Pavel Noel
Posadas Pérez Isaac Sayeg
Sánchez Morales Sergio Israel
Yáñez Reyna Leonardo

Índice

Dino Neuroevolutivo

Repositorio(s)

	Repositorio
Proyecto Integrador	https://github.com/JCriss06/PI_AyR_2025.git

Descripción

El presente proyecto integrador consiste en el desarrollo e implementación de un sistema de inteligencia artificial basado en neuroevolución, diseñado para aprender y dominar de manera autónoma la mecánica del popular videojuego "Dino" de Google Chrome. La simulación del entorno gráfico y las físicas del juego se ha desarrollado utilizando el lenguaje de programación Python y la biblioteca Pygame, creando un ambiente controlado donde agentes virtuales deben sobrevivir el mayor tiempo posible esquivando obstáculos generados proceduralmente. El núcleo del sistema reside en la integración de dos ramas fundamentales de la inteligencia artificial: las redes neuronales artificiales y los algoritmos genéticos.

Para dotar de "inteligencia" a cada agente, se ha implementado un Perceptrón Multicapa (MLP) utilizando la arquitectura de TensorFlow y Keras. Este modelo actúa como el cerebro del dinosaurio y está configurado con una topología específica que consta de una capa de entrada de seis neuronas, una capa oculta de seis neuronas con función de activación ReLU y una capa de salida de una única neurona con activación sigmoide. El sistema toma decisiones en tiempo real procesando seis variables críticas del entorno: la altura actual del personaje, la distancia al obstáculo inmediato, la anchura y altura de dicho obstáculo, la velocidad vertical del personaje y la velocidad de desplazamiento del mundo, la cual se incrementa progresivamente para aumentar la dificultad.

El proceso de aprendizaje no utiliza métodos de supervisión tradicionales como la retropropagación, sino que emplea un enfoque evolutivo. Una población inicial de agentes con pesos sinápticos aleatorios interactúa con el entorno; aquellos que demuestran un mejor desempeño, medido por su tiempo de supervivencia, son seleccionados para transmitir su información genética a la siguiente generación. Mediante operaciones de cruce de pesos y mutaciones aleatorias controladas, el sistema refina iterativamente los parámetros de la red neuronal. La decisión final de ejecutar un salto se toma cuando la salida de la red neuronal supera un umbral predefinido de 0.6, permitiendo así que el agente evolucione desde un comportamiento errático hasta una ejecución precisa y adaptativa ante la variabilidad de los obstáculos.

Introducción

En el ámbito contemporáneo de las ciencias de la computación, la inteligencia artificial ha trascendido la automatización de tareas estáticas para adentrarse en la resolución de problemas dinámicos y complejos. Dentro de este espectro, los videojuegos se han establecido como entornos de prueba ideales para evaluar la capacidad de adaptación de agentes autónomos, dado que ofrecen escenarios controlados, pero con variables cambiantes que requieren toma de decisiones en tiempo real. Este proyecto se sitúa en la intersección del aprendizaje automático y la computación evolutiva, abordando el desafío de entrenar un agente capaz de interactuar exitosamente con un entorno sin intervención humana directa ni conocimiento previo de las reglas del sistema.

La metodología seleccionada para abordar este reto es la neuroevolución, una técnica que combina la estructura de procesamiento de información de las redes neuronales artificiales con la estrategia de optimización de los algoritmos genéticos. A diferencia del aprendizaje supervisado, que requiere grandes conjuntos de datos etiquetados, o del aprendizaje por refuerzo tradicional, que suele demandar extensos recursos computacionales para ajustar políticas mediante gradientes, la neuroevolución imita los procesos biológicos de selección natural. Este enfoque permite optimizar la topología y los pesos de una red neuronal evaluando únicamente el desempeño final del agente en su entorno, lo que resulta en una estrategia eficiente para problemas de control continuo y navegación reactiva.

El desarrollo de este sistema implica la orquestación de múltiples componentes tecnológicos. Por un lado, se requiere una simulación fidedigna de la física del juego que proporcione retroalimentación inmediata; por otro, una arquitectura neuronal capaz de interpretar variables sensoriales como distancias y velocidades para emitir una respuesta binaria de acción. A través de este proyecto integrador, se busca no solo demostrar la viabilidad técnica de la neuroevolución aplicada a entornos lúdicos, sino también analizar cómo parámetros específicos, tales como la tasa de mutación y la selección de variables de entrada, influyen directamente en la convergencia del aprendizaje y en la robustez del comportamiento emergente del agente artificial.

Desarrollo

El desarrollo del juego del dinosaurio neuroevolutivo se llevó a cabo en dos etapas, comenzando por la construcción del videojuego, replicando la dinámica original del "Dino" de Google Chrome mediante Python y Pygame. La segunda etapa correspondió a la implementación del sistema de neuroevolución, integrando un modelo de red neuronal tipo MLP y un algoritmo genético para permitir que los agentes aprendieran a jugar de manera autónoma.

Constantes globales

Constantes globales para facilitar ajustes. Incluye dimensiones de pantalla, forma del personaje, forma de los obstáculos, configuración del algoritmo genético (tasa de mutación, tamaño de población) y estructura de la red neuronal.

settings.py

```
# Dimensiones Pantalla
ANCHO_PANTALLA = 800
ALTO_PANTALLA = 400
SUELO_Y = 350
FPS = 60

# Forma dinosaurio
ANCHO_DINO = 40
ALTO_DINO = 50
COLOR_DINO = (152, 251, 152)

# Forma obstáculo
ANCHO_OBS = (20, 70) # (mínimo, máximo)
ALTO_OBS = (20, 60) # (mínimo, máximo)

POBLACION_TAMANO = 80
MODO_VISUAL = False

# Configuración Genética
PROBABILIDAD_MUTACION = 0.10
VARIACION_MUTACION = 0.5
MAX_GENERACIONES = 2

# Configuración Red Neuronal
INPUT_NODES = 6
HIDDEN_NODES = 6
OUTPUT_NODES = 1
UMBRAL_SALTO = 0.6

MODEL_NAME = 'dino_model'
```

Construcción videojuego

El desarrollo se realizó utilizando Python y la biblioteca Pygame, replicando la mecánica básica del Dino de Google Chrome. Se implementaron elementos como el personaje principal, los obstáculos, las físicas de salto, la detección de colisiones y la lógica del entorno.

Se comenzó por la definición de las entidades que serán las protagonistas en el videojuego, a continuación, se describe su funcionamiento:

La clase `Dinosaurio` representa el personaje del videojuego, definiendo sus características ý controlando su comportamiento durante la partida. El método `update()` gestiona la física del movimiento de forma vertical, aplicando gravedad y detectando cuando el personaje vuelve a tocar el suelo para poder reiniciar el salto. Además, aumenta el score conforme al tiempo que el personaje sigue vivo.

El método `saltar()` modifica la velocidad vertical para iniciar un salto siempre y cuando el dinosaurio no se encuentre saltando.

Nota. Esta entidad será utilizada posteriormente para la versión controlada por la red neuronal.

dinosaurio.py

```
class Dinosaurio:
    def __init__(self, alto=ALTO_DINO, ancho=ANCHO_DINO,
color=COLOR_DINO):
        self.x = 50
        self.y = SUELO_Y
        self.alto = alto
        self.ancho = ancho
        self.color = color
        self.vel_y = 0
        self.saltando = False
        self.vivo = True
        self.score = 0
        self.rect = pg.Rect(self.x, self.y - self.alto, self.ancho,
self.alto)

    def update(self):
        if not self.vivo: return

        # Física
        self.vel_y += 0.8
        self.y += self.vel_y
        if self.y >= SUELO_Y:
            self.y = SUELO_Y
            self.vel_y = 0
            self.saltando = False
        self.rect.y = self.y - self.alto
```

```
            self.score += 1

        def saltar(self):
            if not   self.saltando:
                self.vel_y = -15
                self.saltando = True

        def dibujar(self, screen):
            if self.vivo:
                pg.draw.rect(screen, self.color, self.rect)
```

La clase `Obstaculo` representa los elementos que el dinosaurio debe de evitar durante la partida, estos son generados con dimensiones de manera aleatoria. El método `update()` desplaza el obstáculo horizontalmente hacia la izquierda según la velocidad del juego, simulando el movimiento del escenario.

obstaculo.py

```
class Obstaculo:
    def __init__(self, ancho=ANCHO_OBS, alto=ALTO_OBS, x=0):
        self.ancho = rand.randint(ancho[0], ancho[1]) # tupla (mínimo,
máximo)
        self.alto = rand.randint(alto[0], alto[1]) # tupla (mínimo,
máximo)
        self.rect = pg.Rect(x, SUELO_Y - self.alto, self.ancho,
self.alto)

    def update(self, velocidad):
        self.rect.x -= velocidad

    def dibujar(self, screen):
        pg.draw.rect(screen, (0, 0, 0), self.rect)
```

Una vez definidas las entidades protagonistas del videojuego, se implementó la lógica general del juego utilizando Pygame.

Al iniciar el juego, se configura la ventana y se crea una instancia del dinosaurio junto una lista inicial de los obstáculos. Dentro del loop del juego se gestiona el evento del teclado, puesto que se utiliza la tecla SPACE para que el dinosaurio pueda saltar.

Posteriormente se actualiza la lógica del entorno. Esto incluye:

- Los obstáculos aparecen aleatoriamente a diferentes distancias.
- El movimiento del obstáculo se actualiza conforme la velocidad del mundo.
- Dibujo del fondo, del dinosaurio y de los obstáculos.
- Detección de cuando el dinosaurio choca con un obstáculo.

El puntaje aumenta conforme el jugador permanece con vida. Este código es la versión jugable de forma manual del Dino.

main_manual.py

```
def main():
    # Inicializar Pygame (Forzamos visualización aquí)
    pygame.init()
    pantalla = pygame.display.set_mode((ANCHO_PANTALLA, ALTO_PANTALLA))
    pygame.display.set_caption("Dino Chrome Manual")
    clock = pygame.time.Clock()
    font = pygame.font.Font(None, 30)

    dino = Dinosaurio()
    obstaculos = [Obstaculo(x=ANCHO_PANTALLA + 200)]
    game_speed = 10
    score = 0

    corriendo = True
    while corriendo:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                corriendo = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE:
                    dino.saltar()

        # Dibujar Fondo
        pantalla.fill((255, 255, 255))
        pygame.draw.line(pantalla, (0, 0, 0), (0, SUELO_Y),
(ANCHO_PANTALLA, SUELO_Y))

        # --- LÓGICA DEL JUEGO---
        # Generar obstáculos
        if len(obstaculos) == 0 or obstaculos[-1].rect.x < ANCHO_PANTALLA
- 400:
            if rand.randint(0, 100) < 5:
                obstaculos.append(Obstaculo(x= ANCHO_PANTALLA +
rand.randint(50, 300)))

        # Actualizar obstáculos
        for obs in obstaculos:
            obs.update(game_speed)
            obs.dibujar(pantalla)
            if obs.rect.right < 0:
                obstaculos.remove(obs)

        if dino.vivo:
```

```
                dino.update( )
                dino.dibujar(pantalla)
                score += 1

                # Detección de Colisión
                for obs in obstaculos:
                    if dino.rect.colliderect(obs.rect):
                        dino.vivo = False
                        print(f"--> El dino chocó. Score final: {score}")
            else:
                print("--- Reiniciando partida ---")
                dino.vivo = True
                dino.rect.y = SUELO_Y - dino.alto
                dino.y = SUELO_Y
                dino.vel_y = 0
                dino.saltando = False

                obstaculos = [Obstaculo(x=ANCHO_PANTALLA + 200)]
                game_speed = 10
                score = 0

            game_speed += 0.005
            if game_speed > 30: game_speed = 30

            txt = font.render(f"Score: {score} | Vel: {game_speed:.1f}",
True, (0, 0, 0))
            pantalla.blit(txt, (10, 10))

            pygame.display.flip()
            clock.tick(60)

    pygame.quit()
    sys.exit()


if __name__ == "__main__":
    main()
```

Implementación de neuroevolución

En la implementación de neuroevoloción se llevó a cabo tanto el entrenamiento del modelo, la creación del modelo y las pruebas del modelo para evaluar el desempeño en tiempo real del modelo.

Para comenzar se hizo una extensión del personaje principal, se creó la clase `DinosaurioEvo` que extiende de la clase `Dinosaurio` utilizada en la versión manual del videojuego. Esta

nueva entidad incorpora un modelo que funciona como su "cerebro", permitiéndole analizar el entorno y decidir cuándo saltar.

La clase `DinosaurioEvo` sobrescribe el método `update()` para incluir la función `pensar()`, que esta se encarga de recopilar las seis variables del entorno y procesarlas mediante la red neuronal. Si la salida del modelo supera un umbral predefinido, el agente ejecuta un salto.

dinosaurio_evo.py

```
class DinosaurioEvo(Dinosaurio):
    def __init__(self, modelo=None):
        super().__init__()
        self.cerebro = modelo if modelo is not None else crear_modelo()

    def update(self, obstaculos=None, vel_mundo=None):
        super().update()
        self.pensar(obstaculos, vel_mundo)

    def pensar(self, obstaculos, vel_mundo):
        obs_siguiente = None
        for obs in obstaculos:
            if obs.rect.right > self.rect.left:
                obs_siguiente = obs
                break

        if obs_siguiente:
            # Las 6 variables para la Red Neuronal
            inputs = [
                (SUELO_Y - self.y) / 100.0,  # x1
                (obs_siguiente.rect.x - self.rect.right) / 800.0,  # x2
                obs_siguiente.rect.width / 100.0,  # x3
                obs_siguiente.rect.height / 100.0,  # x4
                self.vel_y / 20.0,  # x5
                vel_mundo / 20.0  # x6
            ]

            # Usar Keras para decidir
            decision = predecir_accion(self.cerebro, inputs)

            if decision >= UMBRAL_SALTO and not self.saltando:
                self.saltar()
```

El "cerebro" del agente está conformado por un perceptrón multicapa implementado con TensorFlow y Keras. El modelo consta de:

- Capa de entrada: 6 variables del entorno.
- Capa oculta: número configurable de neuronas con activación ReLU.
- Capa de salida: una única neurona sigmoide cuyo valor indica la probabilidad de realizar un salto.

El modelo se construye mediante la función `crear_modelo()` y se ejecuta utilizando `predecir_accion()`, que convierte las entradas en tensores y obtiene una predicción eficiente sin retropropagación. Debido a que la neuroevolución no requiere entrenamiento tradicional, el modelo no se optimiza con gradientes sino exclusivamente mediante la modificación evolutiva de sus pesos.

Asimismo, se incluyen funciones auxiliares para guardar y cargar modelos (`guardar_modelo_keras`, `cargar_modelo_keras`) con el fin de salvar el mejor desempeño alcanzado.

modelo.py

```
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

def crear_modelo():
    """Crea una Red Neuronal MLP usando Keras."""
    model = Sequential([
        # Capa Oculta (Dense) con activación ReLU
        Dense(HIDDEN_NODES, input_shape=(INPUT_NODES,),
activation='relu'),

        # Capa de Salida (Dense) con activación Sigmoide (0 a 1)
        Dense(OUTPUT_NODES, activation='sigmoid')
    ])

    model.compile(loss='mse', optimizer='adam')
    return model


def predecir_accion(model, inputs):
    """Realiza la inferencia (predicción) de forma eficiente."""
    input_tensor = tf.convert_to_tensor([inputs], dtype=tf.float32)

    prediccion = model(input_tensor, training=False)

    # Devolver el valor numérico (ej. 0.85)
    return prediccion.numpy()[0][0]


def guardar_modelo_keras(model, nombre_base="dino_modelo"):
    """Guarda el modelo en los formatos .keras y .weights.h5"""
    try:
        # Guardar modelo completo
        model.save(f"{nombre_base}.keras")
        model.save_weights(f"{nombre_base}.weights.h5")
        print(f"--> Guardado: {nombre_base}.keras y
{nombre_base}.weights.h5")
    except Exception as e:
        print(f"Error guardando modelo: {e}")


def cargar_modelo_keras(ruta_modelo):
```

```
    """Carga un modelo .keras existente"""
    return load_model(ruta_modelo)
```

El entrenamiento del agente se realizo mediante el uso de un algoritmo genético. En cada generación se evalúa una población completa de dinosaurios autónomos, cada uno con pesos iniciales aleatorios. El score obtenido por cada uno es utilizado como fitness.

El proceso evolutivo incluye:

- Elitismo: Se clona el dino que mejor desempeño demostró.
- Selección: Selección de los mejores dinos con el fin de convertirlos en candidatos para la reproducción.
- Crossover: Mezcla los pesos de dos padres para crear nuevos modelos.
- Mutación: Agrega variabilidad para poder explorar mas el espacio de soluciones.

Con esto permite a la red adaptar sus pesos generación tras generación, creando así dinos cada vez más capaces.

genetica.py

```
def evolucionar_poblacion(dinos_muertos):
    """Algoritmo Genético usando pesos de modelos Keras."""
    print("--- Evolucionando (TensorFlow) ---")

    # Ordenar por fitness
    dinos_muertos.sort(key=lambda x: x.score, reverse=True)
    mejor_dino = dinos_muertos[0]
    print(f"Mejor Score: {mejor_dino.score}")

    nueva_poblacion = []

    # 1. ELITISMO: Clonar al mejor
    mejor_modelo = crear_modelo()
    mejor_modelo.set_weights(mejor_dino.cerebro.get_weights())
    nueva_poblacion.append(DinosaurioEvo(modelo=mejor_modelo))

    # 2. REPRODUCCIÓN
    while len(nueva_poblacion) < POBLACION_TAMANO:
        padre1 = random.choice(dinos_muertos[:8])  # Top 8
        padre2 = random.choice(dinos_muertos[:8])

        # Cruzar cerebros
        pesos_hijo = cruzar_pesos(padre1.cerebro.get_weights(),
                                  padre2.cerebro.get_weights())

        # Crear dino nuevo y asignarle los pesos mutados
        modelo_hijo = crear_modelo()
        modelo_hijo.set_weights(pesos_hijo)
        nueva_poblacion.append(DinosaurioEvo(modelo=modelo_hijo))
```

```
    return nueva_poblacion, mejor_modelo


def cruzar_pesos(pesos_p1, pesos_p2):
    """
    Recibe listas de matrices de numpy (que vienen de get_weights).
    Devuelve una nueva lista con los pesos mezclados y mutados.
    """
    nuevos_pesos = []

    for w1, w2 in zip(pesos_p1, pesos_p2):
        # w1 y w2 son matrices de pesos o vectores de bias

        # 1. Cruce (Crossover Uniforme)
        mask = np.random.rand(*w1.shape) > 0.5
        hijo_w = np.where(mask, w1, w2)

        # 2. Mutación
        if random.random() < PROBABILIDAD_MUTACION:
            noise = np.random.normal(0, VARIACION_MUTACION, w1.shape)
            hijo_w = hijo_w + noise

        nuevos_pesos.append(hijo_w)

    return nuevos_pesos
```

Finalmente, se desarrollo el entorno de entrenamiento que este ejecuta repetidamente el juego en modo automático. En cada iteración:

- Todos los dinos de la población juegan.
- Los obstáculos se van generando, incrementando la dificultad al aumentar la velocidad del mundo.
- Cada dinosaurio toma decisiones mediante su MLP y continúa mientras no choque con algún obstáculo.
- Cuando toda la población muere, se aplica el algoritmo genético y se crea una nueva generación.
- El proceso continúa hasta alcanzar el número máximo de generaciones o hasta que se muestre un desempeño satisfactorio.

El sistema permite ejecutarse tanto con visualización como en acelerado sin gráficos, lo que reduce el tiempo de entrenamiento.

main_train.py

```
def main():
    path = './model/'
    model_name = path + MODEL_NAME
    if MODO_VISUAL:
        pygame.init()
```

```
        pantalla = pygame.display.set_mode((ANCHO_PANTALLA,
ALTO_PANTALLA))
        pygame.display.set_caption("Dino IA - Entrenamiento")
        clock = pygame.time.Clock()
        font = pygame.font.Font(None, 30)
    else:
        print("--- INICIANDO EN MODO 2DO PLANO (SIN VENTANA) ---")
        print("Esto entrenará mucho más rápido y consumirá menos
recursos.")

    # Población inicial
    dinos = [DinosaurioEvo() for _ in range(POBLACION_TAMANO)]
    dinos_muertos = []

    obstaculos = [Obstaculo(x=ANCHO_PANTALLA + 200)]
    game_speed = 10
    generacion = 1
    mejor_modelo_global = None

    ejecutando = True
    while ejecutando:
        if MODO_VISUAL:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    ejecutando = False
                    if mejor_modelo_global:
                        guardar_modelo_keras(mejor_modelo_global,
model_name)

            pantalla.fill((255, 255, 255))
            pygame.draw.line(pantalla, (0, 0, 0), (0, SUELO_Y),
(ANCHO_PANTALLA, SUELO_Y))

        # --- Lógica del Juego ---
        if len(obstaculos) == 0 or obstaculos[-1].rect.x < ANCHO_PANTALLA
- 400:
            if rand.randint(0, 100) < 5:
                obstaculos.append(Obstaculo(x=ANCHO_PANTALLA +
rand.randint(50, 300)))

        # Mover Obstáculos
        for obs in obstaculos:
            obs.update(game_speed)
            if MODO_VISUAL: obs.dibujar(pantalla)
            if obs.rect.right < 0:
                obstaculos.remove(obs)

        # Mover Dinos
        vivos_actuales = 0
        for dino in dinos:
            if dino.vivo:
                vivos_actuales += 1
                dino.update(obstaculos, game_speed)
                if MODO_VISUAL: dino.dibujar(pantalla)
```

```
                for obs in obstaculos:
                    if dino.rect.colliderect(obs.rect):
                        dino.vivo = False
                        dinos_muertos.append(dino)

        game_speed += 0.005
        if game_speed > 30: game_speed = 30

        # --- NUEVA GENERACIÓN ---
        if vivos_actuales == 0:
            if generacion >= MAX_GENERACIONES:
                print(f"--- META ALCANZADA: {generacion} GENS ---")
                if mejor_modelo_global:
guardar_modelo_keras(mejor_modelo_global, model_name)
                ejecutando = False
                break

            dinos, mejor_modelo_gen =
evolucionar_poblacion(dinos_muertos)
            mejor_modelo_global = mejor_modelo_gen

            print(f"Gen: {generacion} Completada. Mejor Score:
{int(dinos_muertos[0].score)}")

            dinos_muertos = []
            obstaculos = [Obstaculo(x=ANCHO_PANTALLA + 200)]
            game_speed = 10
            generacion += 1

        if MODO_VISUAL:
            txt = font.render(f"Gen: {generacion} | Vivos:
{vivos_actuales} | Vel: {game_speed:.1f}", True, (0, 0, 0))
            pantalla.blit(txt, (10, 10))
            pygame.display.flip()
            clock.tick(FPS)

    if MODO_VISUAL: pygame.quit()
    sys.exit()


if __name__ == "__main__":
    main()
```

Una vez finalizada la etapa de neuroevolución y de haber obtenido el modelo entrenado, se procede a la fase de probarlo, con el fin de ver el comportamiento del dino inteligente dentro del videojuego. En este punto el juego ya no es controlado por el usuario, sino por la red previamente entrenada.

Al igual que la versión de entrenamiento, se hace uso del `DinosaurioEvo`, que es una variante que incluye un "cerebro" capaz de interpretar el entrono y decidir si debe saltar. Además, se recrea el entorno del juego para que comience la fase de prueba.

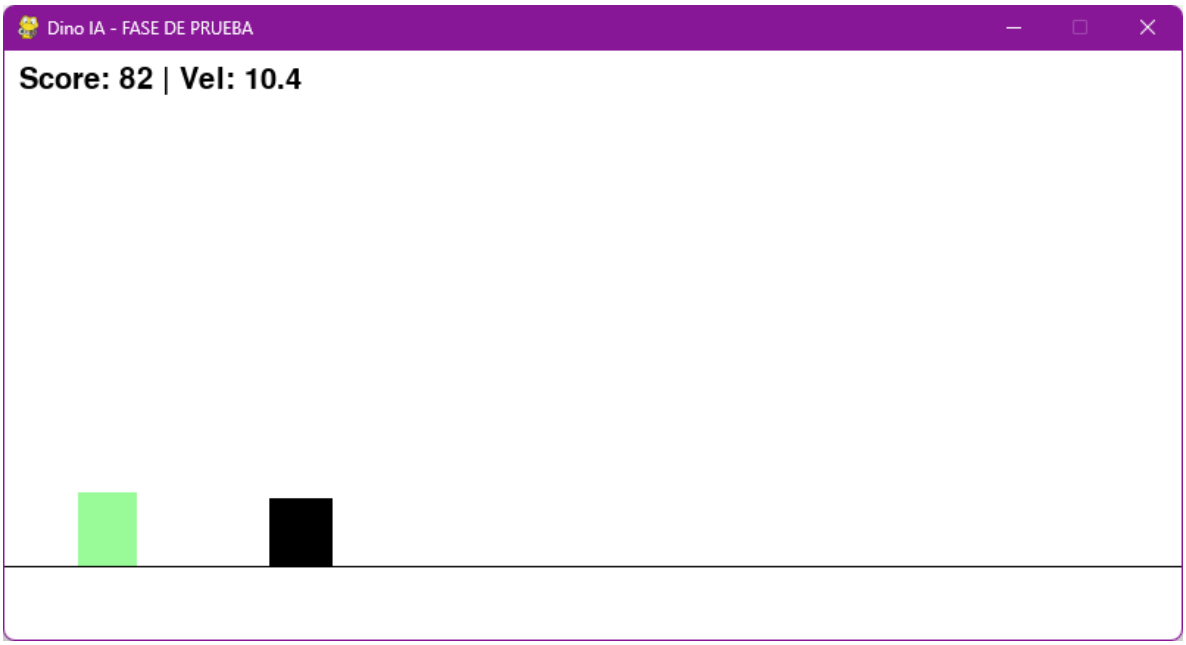
El dino ahora toma decisiones de forma completamente autónoma, demostrando su capacidad para identificar obstáculos y ejecutar saltos en el momento adecuado. Esta etapa funciono como validación final del proceso, permitiendo comprobar su desempeño bajo condiciones reales y dinámicas del juego.

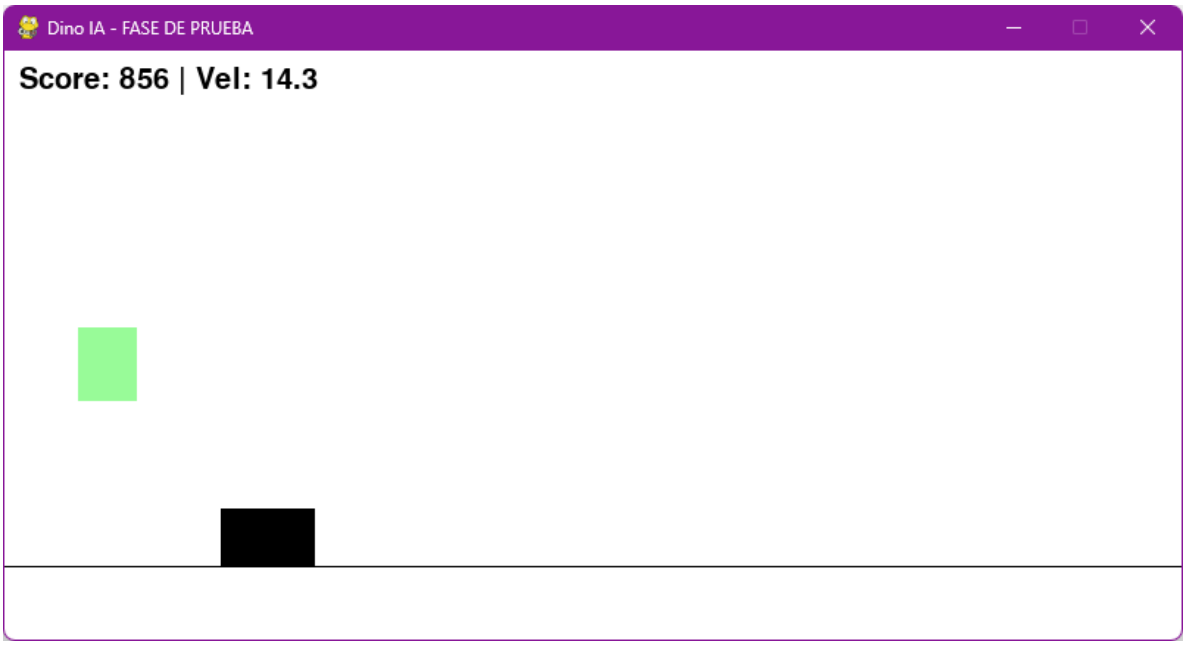
main_ia.py

```
def main():
    pygame.init()
    pantalla = pygame.display.set_mode((ANCHO_PANTALLA, ALTO_PANTALLA))
    pygame.display.set_caption("Dino IA - FASE DE PRUEBA")
    clock = pygame.time.Clock()
    font = pygame.font.Font(None, 30)

    # --- 1. CARGAR EL MODELO ---
    path = './src/training/model/'
    model_name = path + f"{MODEL_NAME}.keras"
    print(f"Cargando cerebro desde: {model_name}...")

    try:
        modelo_cargado = load_model(model_name)
        print("¡ÉXITO! Modelo cargado.")
    except Exception as e:
        print(f"\n[ERROR CRÍTICO] No se pudo cargar '{model_name}'.")
        print("Asegúrate de que el entrenamiento (main.py) terminó y creó
el archivo.")
        print(f"Detalle técnico: {e}")
        try:
            print("Intentando cargar backup_dino.keras...")
            modelo_cargado = load_model("backup_dino.keras")
            print("¡Backup cargado!")
        except:
            sys.exit()

    # --- 2. PREPARAR EL JUEGO ---
    dino = DinosaurioEvo(modelo=modelo_cargado)

    obstaculos = [Obstaculo(x=ANCHO_PANTALLA + 200)]
    game_speed = 10
    score = 0

    corriendo = True
    while corriendo:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                corriendo = False

        # Dibujar Fondo
        pantalla.fill((255, 255, 255))
        pygame.draw.line(pantalla, (0, 0, 0), (0, SUELO_Y),
(ANCHO_PANTALLA, SUELO_Y))

        # --- LÓGICA DEL JUEGO ---
        # Generar obstáculos
```

```
        if len(obstaculos) == 0 or obstaculos[-1].rect.x < ANCHO_PANTALLA
- 400:
            if rand.randint(0, 100) < 5:
                obstaculos.append(Obstaculo(x=ANCHO_PANTALLA +
rand.randint(50, 300)))

        # Actualizar obstáculos
        for obs in obstaculos:
            obs.update(game_speed)
            obs.dibujar(pantalla)
            if obs.rect.right < 0:
                obstaculos.remove(obs)

        # Actualizar Dino (predicción)
        if dino.vivo:
            dino.update(obstaculos, game_speed)
            dino.dibujar(pantalla)
            score += 1

            # Detección de Colisión
            for obs in obstaculos:
                if dino.rect.colliderect(obs.rect):
                    dino.vivo = False
                    print(f"--> El dino chocó. Score final: {score}")

        else:
            print("--- Reiniciando partida de prueba ---")
            dino.vivo = True
            dino.rect.y = SUELO_Y - dino.alto
            dino.y = SUELO_Y
            dino.vel_y = 0
            dino.saltando = False

            obstaculos = [Obstaculo(x=ANCHO_PANTALLA + 200)]
            game_speed = 10
            score = 0

        game_speed += 0.005
        if game_speed > 30: game_speed = 30

        txt = font.render(f"Score: {score} | Vel: {game_speed:.1f}",
True, (0, 0, 0))
        pantalla.blit(txt, (10, 10))

        pygame.display.flip()
        clock.tick(60)

    pygame.quit()
    sys.exit()


if __name__ == "__main__":
    main()
```

Ejecución


Dino IA - FASE DE PRUEBA
Score: 82 | Vel: 10.4


Dino IA - FASE DE PRUEBA
Score: 856 | Vel: 14.3

Conclusión

La realización de este Proyecto Integrador ha permitido demostrar cómo la programación clásica de videojuegos puede converger exitosamente con conceptos avanzados de Inteligencia Artificial para crear sistemas autónomos capaces de aprender y adaptarse. A lo largo del desarrollo, no solo se logró replicar la mecánica del famoso juego del dinosaurio utilizando librerías gráficas como Pygame, sino que se transformó este entorno en un laboratorio de pruebas para algoritmos de aprendizaje automático. El núcleo del proyecto dejó de ser simplemente "jugar" para convertirse en "simular", donde el objetivo principal fue sustituir la intervención humana por una toma de decisiones automatizada y eficiente.

Uno de los pilares fundamentales de este trabajo fue la implementación de una Red Neuronal Artificial como el "cerebro" del agente. A diferencia de la programación tradicional, donde se dictan reglas rígidas (por ejemplo: "si el obstáculo está cerca, salta"), este sistema se diseñó para percibir su entorno a través de sensores simulados —como la distancia a los obstáculos o la velocidad del juego— y procesar esa información para tomar una decisión en tiempo real. Esto valida la capacidad de las redes neuronales para interpretar situaciones dinámicas y variables sin necesidad de que el programador prevea cada escenario posible.

Sin embargo, el reto del proyecto residió en la integración de la Computación Evolutiva. Al carecer de un conjunto de datos preexistente para "entrenar" al dinosaurio de manera convencional, se enfocó en la selección natural. El módulo genético desarrollado permitió que el sistema partiera de individuos sin conocimientos previos, que tomaban decisiones aleatorias, y progresivamente refinara su comportamiento generación tras generación. Mediante procesos de selección, cruce y mutación, logramos que el código "descubriera" por sí mismo las estrategias óptimas de supervivencia. Esto demuestra empíricamente que las máquinas pueden aprender a resolver problemas complejos simplemente interactuando con su entorno y recibiendo retroalimentación basada en su desempeño.