PH227 - PROJECT REPORT

HOW TO TRAIN YOUR DINOSAUR

Using Supervised Learning and Genetic Learning to play a self-made recreation of the "El Clasico" Chrome Dino Game

Team Billie Jean

1. Chinmay Kale, 23B1849
2. Gokularamanan R S, 23B1854
3. Arya Joshi, 23B1853
4. Arash Dev Ahlawat, 23B1817

Our code repository: https://github.com/ChiniKale/ChromeDinoGame

Introduction:

This project involves the re-creation of the classic Chrome Dinosaur game using python and making AI play the game. We have used **Supervised Learning** as well as **Genetic Learning** in parts to train the model.

The problem at hand:

We have to make the ML model learn to play the dino game by jumping and ducking over/ under obstacles - birds (more like *velociraptors :-)*) and cacti, in a **continuously and randomly generated obstacle course, with varying speed levels** (the speed of the dino and the course increases as we progress further into the game).

Our Approach to the problem:

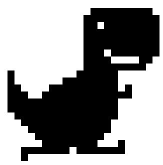
- We first built a standalone game on python using the help of pygame so that we could visually and manually play the game in order to collect training data. For this part, we coded 3 initial files, named in the repo as basic.py, obstacle.py, and dinosaur.py. It is important to note that the game was created ***entirely by us***, with minimal usage of GPT, and even a couple of the sprites were hand-drawn
    - In order to collaborate, we set up a GitHub repo, and used git commands to stay updated.
    - These files described the basic dinosaur game, along with an extra easter egg (batsymbol.py) which projects a bat symbol onto the background.
    - The dinosaur can either duck, or perform a big or small jump. The dinosaur has a running animation.
    - The objects spawn randomly, they can be either cacti, or (animated flapping) birds which fly in the sky, and can be ducked under.
- Then, we collected the training data by manually playing the game using the *Training_Runs.py* file that saved each run in the form of a csv file, of the following format:

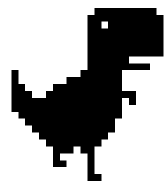
```
Time,Obstacle Distance,Obstacle Height,Obstacle Size,Velocity,Dino Jumping,Dino Ducking,Action
```

- Here the columns are as follows:
  - Time: Records the time at which an action takes place
  - Obstacle Distance: Records the distance between the dinosaur and the oncoming obstacle.
  - Obstacle Height: Records the height of the obstacle.
  - Obstacle Size: Records the size of the obstacle.
  - Velocity: Records the velocity of the dinosaur.
  - Dino Jumping: Denotes if in the current state the dino is jumping or not, as a binary 0 or 1.

  - Dino Ducking: Denotes if in the current state the dino is ducking or not.


(Duck)


(Jump)


(We also have run)

- Next, we concatenated all the csv files into one file and fed this as "training data" to the train_simple_model.py, which is a very basic ANN model that trains over the given data and saves the best model in a .pth file (say, *best_model.pth*). This concludes the Supervised Learning part.
- Once this is done, we pass this *best_model.pth* file to the RL model, Genetic_Model.py that uses Genetic algorithms such as - mutation, cross-over

Components of the code:

1. Game components and classes:
   a. dinosaur.py

```python
import pygame
dinocolour = 255,255,255
DINOHEIGHT = 45
DINOWIDTH = 20
FPS = 60
class Dinosaur:
    def __init__(self, surfaceHeight):
        self.x = 60
        self.y = 0
        self.height = 60  # New height of the dinosaur
        self.width = 40
        self.yvelocity = 0
        self.is_collided = False
        self.is_ducking = False

        size = (self.width, self.height)

        self.running_frames = [
            pygame.transform.scale(pygame.image.load(r"dinorun0000.png"), size),
            pygame.transform.scale(pygame.image.load(r"dinorun0001.png"), size),
        ]
        self.jumping_frames = [
            pygame.transform.scale(pygame.image.load(r"dinoJump0000.png"), size),
        ]
        self.collision_frames = [
            pygame.transform.scale(pygame.image.load(r"dinoDead0000.png"), size),
        ]
        self.ducking_frames = [
            pygame.transform.scale(pygame.image.load(r"dinoduck0000.png"), (1.4*self.width, self.height // 2)),
            pygame.transform.scale(pygame.image.load(r"dinoduck0001.png"), (1.4*self.width, self.height // 2)),
        ]
        self.current_frame = 0
        self.animation_time = 0.1  # Time per frame in seconds
        self.time_accumulator = 0  # Tracks elapsed time to switch frames
        self.height = DINOHEIGHT
        self.width = DINOWIDTH
        self.surfaceHeight = surfaceHeight
        self.is_jumping = False  # Indicates if the dinosaur is jumping
```

This section of code gives the *__init__* function of the dinosaur, along with the basic dimensions, such as width. The <frames> arrays that are seen here, are to do with animation, as the sprite rapidly switches back and forth between the 2 png files.

```
def update_collision_animation(self, deltaTime):
    # Update animation frame if collision occurs
    if self.is_collided:
        self.time_accumulator += deltaTime
        if self.time_accumulator > self.animation_time:
            self.current_frame += 1
            self.time_accumulator = 0
        if self.current_frame >= len(self.collision_frames):
            return True  # Animation is complete
    return False
def bigjump(self): #When adding classes into function, the first parameter must be the parameter
    if(self.y == 0): #Only allow jumping if the dinosaur is on the ground to prevent mid air jumps.
        self.yvelocity = 300
        self.is_jumping = True
def smoljump(self): #When adding classes into function, the first parameter must be the parameter
    if(self.y == 0): #Only allow jumping if the dinosaur is on the ground to prevent mid air jumps.
        self.yvelocity = 200
        self.is_jumping = True
def duck(self, is_ducking):
    self.is_ducking = is_ducking
    if self.is_ducking:
        self.height = 30  # Reduce height for ducking posture
    else:
        self.height = 60  # Reset height when not ducking
```

This section of code gives us the collision detection animation(sprite changes upon collision), and the action functions that tell us what the dinosaur does when it jumps/ducks. It sets certain boolean values to True/False, which will be used later.

b. [Easter egg] batsymbol.py

```python
def update(self, deltaTime): #Updates the y position of the dinosaur each second
    if not self.is_ducking:
        self.yvelocity += -750*deltaTime
    else:
        self.yvelocity += -2500*deltaTime #Gravity
    self.y += self.yvelocity * deltaTime
    if self.y < 0: #if the dinosaur sinks into the ground, make velocity and y = 0
        self.y = 0
        self.yvelocity = 0
        self.is_jumping = False
    self.time_accumulator += deltaTime
    if self.time_accumulator > self.animation_time:
        self.current_frame = (self.current_frame + 1) % len(self.running_frames)
        self.time_accumulator = 0


def draw(self, display):
    if self.is_jumping or self.y > 0:
        current_image = self.jumping_frames[self.current_frame % len(self.jumping_frames)]
    if self.is_ducking:
        current_image = self.ducking_frames[self.current_frame % len(self.ducking_frames)]
    elif self.is_collided:
        current_image = self.collision_frames[self.current_frame % len(self.collision_frames)]
    else:
        current_image = self.running_frames[self.current_frame % len(self.running_frames)]

    display.blit(current_image, (self.x, self.surfaceHeight - self.y - self.height))
```

These 2 functions are the update functions, which work as the functions that control the animation/gravity, and the draw function, which actually draws the sprite.

```python
import pygame
import random

colour = 0, 0, 255

class Obstacle:
    def __init__(self, x, size, GroundHeight, is_high=True):
        self.x = x
        self.size = size
        self.GroundHeight = GroundHeight
        self.is_high = is_high
        self.current_frame = 0
        self.animation_time = 0.5  # Time per frame in seconds
        self.time_accumulator = 0
        if is_high:
            self.y = GroundHeight - size -30
            self.frames = [pygame.transform.scale(pygame.image.load(r"birb.png"), (size, size)), pygame.transform.scale(pygame.image.load(r"birb2.png"), (size, size))]  #
        else:
            self.y = GroundHeight - size
            self.frames = [pygame.transform.scale(pygame.image.load(r"cactus.png"), (size, size)), pygame.transform.scale(pygame.image.load(r"cactus.png"), (size, size))]

    def draw(self, gameDisplay):
        current_image = self.frames[self.current_frame % len(self.frames)]
        gameDisplay.blit(current_image, (self.x, self.y))

    def update(self, deltaTime, velocity):
        self.x -= velocity * deltaTime
        self.time_accumulator += deltaTime
        if self.time_accumulator > self.animation_time:
            self.current_frame = (self.current_frame + 1) % len(self.frames)
            self.time_accumulator = 0

    def checkOver(self):
        return self.x < 0
```

This is the obstacle class, which gives us the 2 types of obstacles (cactus and bird). According to a bit of code in basic.py ahead, the bool is_high is set, which defines whether the object's y coordinate will be on the ground, or elevated. Accordingly, the animation of the objects is different. The checkOver function will be explained later.

c. batsymbol.py

```
import pygame
import random

class Batsymb:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.active = False
        self.image = pygame.image.load("Bat symbol.png")
        self.image = pygame.transform.scale(self.image, (260, 260))  # Scale cloud size
        self.timer = random.randint(5, 10)

    def update(self, deltaTime, velocity=100):
        if self.active:
            self.x -= velocity * deltaTime  # Move the symbol
            if self.x < -self.image.get_width():  # If it moves off-screen
                self.active = False  # Deactivate
                self.timer = random.randint(5, 10)  # Reset the timer

        else:  # Decrement timer when inactive
            self.timer -= deltaTime
            if self.timer <= 0:  # Reactivate after the timer runs out
                self.x = 800 + random.randint(0, 300)  # Reset position
                self.y = 115  # Randomize vertical position
                self.active = True

    def draw(self, gameDisplay):
        if self.active:
            gameDisplay.blit(self.image, (self.x, self.y))
```

This is the easter egg batsymbol.py file, which spawns a bat symbol floating along in the background, at some time intervals.

d. basic.py

This is the file that was coded before the modelling was added to it. Hence, we added various quality of life improvements, which were later removed.

```
import pygame
from dinosaur import Dinosaur #import the class Dinosaur from the file 'dinosaur'
from obstacle import Obstacle
from batsymbol import Batsymb

pygame.init() #this 'starts up' pygame
clock = pygame.time.Clock()
from pygame import mixer

# Starting the mixer
mixer.init()

# Loading the song
mixer.music.load("bgm.mp3")

Bat =  Batsymb(0, 115)
# Setting the volume
mixer.music.set_volume(0.7)

# Start playing the song
mixer.music.play(loops = -1)
game_timer = 0
size = width,height = 640, 480#creates tuple called size with width 400 and height 230
gameDisplay= pygame.display.set_mode(size) #creates screen
xPos = 0
yPos = 0
black = 0,0,0
GROUND_HEIGHT = height-100
collision = False  # Track collision state
collision_animation_complete = False


dinosaur = Dinosaur(GROUND_HEIGHT)

lastFrame = pygame.time.get_ticks() #get ticks returns current time in milliseconds
```

This is the basic initialisation files, which start up pygame, and a background music track that plays while the dino is alive. Other initialisations include getting the last frame, and setting collision bool to be False.

```
import random
MINGAP = 200
MAXGAP = 600
MAXSIZE = 40
MINSIZE = 20
obstacles = []
lastObstacle = width
text_font = pygame.font.SysFont("Helvetica", 30)
colour = 255
#direction = 0.1
obstaclesize = 20

ground_image = pygame.image.load(r"ground.png")  # Load the ground texture
ground_width = ground_image.get_width()        # Get the width of the texture
ground_scroll = 0

def draw_text(text, font, text_col, x, y):
    img = font.render(text, True, text_col)
    gameDisplay.blit(img, (x, y))

def reset_game_and_exit_gameover():
    global game_over
    reset_game()
    game_over = False
def button(text, x, y, width, height, inactive_color, active_color, action=None):
    mouse = pygame.mouse.get_pos()
    click = pygame.mouse.get_pressed()
    color = active_color if x + width > mouse[0] > x and y + height > mouse[1] > y else inactive_color
    pygame.draw.rect(gameDisplay, color, (x, y, width, height))

    btn_font = pygame.font.SysFont("Helvetica", 20)
    text_surf = btn_font.render(text, True, (0, 0, 0))
    text_rect = text_surf.get_rect(center=(x + width // 2, y + height // 2))
    gameDisplay.blit(text_surf, text_rect)

    if color == active_color and click[0] == 1 and action:
        action()
def reset_game():
    global game_timer, lastFrame, dinosaur, obstacles, lastObstacle, ground_scroll
    game_timer = 0  # Reset game timer
    lastFrame = pygame.time.get_ticks()
    dinosaur = Dinosaur(GROUND_HEIGHT)
    obstacles = []
    lastObstacle = width
    ground_scroll = 0  # Reset ground scrolling position
    mixer.music.load("bgm.mp3")
    mixer.music.play(loops=-1)
```

These are the helper functions, and object conditions. Variables like MINGAP, MAXGAP help define the distance between consecutive obstacles. The helper functions reset the game and define a button that allows us to reset the game.

```
white = 255,255,255
while True:  # Game loop
    t = pygame.time.get_ticks()
    deltaTime = (t - lastFrame) / 1000.0
    lastFrame = t

    # Increment game timer using delta time
    game_timer += deltaTime
    VELOCITY = 300 + 0.01 * game_timer * 1000  # Adjust velocity based on game timer

    for event in pygame.event.get():
        keys = pygame.key.get_pressed()
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_SPACE:
                dinosaur.bigjump()
        """ if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                dinosaur.smoljump() """
        if keys[pygame.K_DOWN]:
            dinosaur.duck(True)
        else:
            dinosaur.duck(False)

    gameDisplay.fill((colour, colour, colour))
    ground_scroll -= VELOCITY * deltaTime
    if ground_scroll <= -ground_width:
        ground_scroll += ground_width

    gameDisplay.blit(ground_image, (ground_scroll, 300))
    gameDisplay.blit(ground_image, (ground_scroll + ground_width, 300))

    # Draw Score
    draw_text(f"Score: {int(game_timer*10)}", text_font, (0, 255, 0), 100, 50)
    if int(game_timer*10) % 100 == 0 and int(game_timer*10) != 0:
        mixer.music.pause()
        achievement = mixer.Sound("100.mp3")
        achievement.play()
        mixer.music.unpause()
    dinosaur.update(deltaTime)
    dinosaur.draw(gameDisplay)
    Bat.update(deltaTime)
    Bat.draw(gameDisplay)
```

This section of code initialises the game loop, and takes inputs from the user, which then define actions, bigjump, smoljump, and duck. The code changes the bools as per its state. The game loop continuously updates the screen every frame, including rendering the ground, displaying the score, and playing an achievement sound every time the score crosses 100.

```python
if len(obstacles) == 0 or obstacles[-1].x < width - MINGAP:
    ahaha = random.random()
    is_high = ahaha > 0.7
    obstacle_size = random.randint(MINSIZE, MAXSIZE) if not is_high else 30
    obstacles.append(Obstacle(lastObstacle, obstacle_size, GROUND_HEIGHT, is_high))
    lastObstacle += MINGAP + (MAXGAP - MINGAP) * random.random() + 0.01 * game_timer * 1000

# Check for collisions and update obstacles
for obs in obstacles:
    obs.update(deltaTime, VELOCITY)
    obs.draw(gameDisplay)

    dino_rect = pygame.Rect(
        dinosaur.x, dinosaur.surfaceHeight - dinosaur.y - dinosaur.height, dinosaur.width, dinosaur.height
    )
    obs_rect = pygame.Rect(obs.x, obs.y, obs.size, obs.size)

    if dino_rect.colliderect(obs_rect):
        mixer.music.load("gameover.mp3")
        mixer.music.set_volume(0.7)
        mixer.music.play()

        game_over = True
        while game_over:
            gameDisplay.fill(white)
            draw_text(f"Game Over. Score: {int(game_timer*10)}", text_font, (255, 0, 0), width // 2 - 100, height // 2)
            button("Restart", width // 2 - 100, height // 2 + 50, 100, 40, (100, 200, 100), (50, 150, 50), lambda: reset_game_and_exit_gameover())

            pygame.display.update()

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    quit()

lastObstacle -= VELOCITY * deltaTime
pygame.display.update()
```

This final bit of code sets the height of an object by a random variable. It then checks whether the dinosaur has collided with any object, by the colliderect function. If collided, it sets game_over to true, and renders the button and plays the game over music (Pac-Man!)

2. Model Components:
    a. Training_Runs.py

Most of the code is same as the one in basic.py, except for the part where we record the log data:

```
Time,Obstacle Distance,Obstacle Height,Obstacle Size,Velocity,Dino Jumping,Dino Ducking,Action
```

And, the code snippet for this part is as follows:

```python
def get_game_state(dinosaur, obstacles, velocity):
    next_obstacles = [obs for obs in obstacles if obs.x > dinosaur.x]
    if len(next_obstacles) > 0:
        next_obstacle = next_obstacles[0]
        # second_obstacle = next_obstacles[1] if len(next_obstacles) > 1 else None

        # Additional state features
        state = [
            next_obstacle.x - dinosaur.x,  # Distance to next obstacle
            next_obstacle.y,  # Vertical distance
            next_obstacle.size,           # Size of next obstacle
            # second_obstacle.x - dinosaur.x if second_obstacle else WIDTH,  # Distance to second obstacle
            velocity,                     # Current game velocity
            1 if dinosaur.is_jumping else 0,  # Is the dinosaur jumping?
            1 if dinosaur.is_ducking else 0,  # Is the dinosaur ducking?
        ]
    else:
        # Default state when no obstacles are nearby
        state = [WIDTH, 0, 0, velocity, 0, 0]

    return np.array(state, dtype=np.float32)
      aryajoshi0110, last week • Added the mechanism to record train data.
training_data = []  # List to store game states and actions
```

```python
# Log game state and action
game_state = get_game_state(dinosaur, obstacles, VELOCITY)
if action != 2 or t%100 == 0:  # Log data only when action is not 'do nothing'
    training_data.append([t] + game_state.tolist() + [action])
```

```python
finally:
    timestamp = time.strftime("%Y%m%d-%H%M%S")  # Format: YYYYMMDD-HHMMSS
    filename = f"Train_Data/{timestamp}.csv"  # Example: dino_training_data_20241118-143500.csv

    # Create and write to the file
    with open(filename, "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["Time","Obstacle Distance","Obstacle Height", "Obstacle Size", "Velocity", "Dino Jumping", "Dino Ducking", "A
        writer.writerows(training_data)
    print("Training data saved!")
```

(The above three snippets are not continuous, we have just highlighted the extra parts from basic.py) These snippets are self explanatory.

b. train_simple_model.py

This code is for training the neural network to learn to play the game using reinforcement learning. Here's a breakdown of the key components and steps:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class DinoModel(nn.Module):
    def __init__(self):
        super(DinoModel, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(6, 128),  # Increased size for initial layer
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 3)  # Output: [jump, duck, do nothing]
        )

    def forward(self, x):
        return self.fc(x)
```

It first checks whether a CUDA-capable GPU is available and sets the device accordingly for training .We build a NN model called DinoModel using PyTorch. It has 4 fully connected layers with ReLU activation functions. It takes in 6 features (things like Object distance, Game velocity,Object height, Object size, If Dinosaur jumping and If dinosaur Ducking.) and outputs 3 values representing actions: jump, duck, or remaining idle.

```python
# Directory containing the training data files
train_data_folder = "D:/IITB/Second Year/Sem 3/PH227/ChromeDinoGame/Train_Data"

# List to hold dataframes
dataframes = []

# Iterate over all files in the training data folder
for filename in os.listdir(train_data_folder):
    if filename.endswith('.csv'):
        file_path = os.path.join(train_data_folder, filename)
        df = pd.read_csv(file_path, header=0)  # Read the CSV file, assuming the first row is the header
        dataframes.append(df)

# Concatenate all dataframes into one
df = pd.concat(dataframes, ignore_index=True)

# Drop any non-numeric columns (or handle them appropriately)
df = df.apply(pd.to_numeric, errors='coerce')  # Converts everything to numeric, NaN for errors

# Replace NaNs (optional, depending on your data)
df.fillna(0, inplace=True)

X = df.iloc[:, 1:7].values  # Features (assuming columns 1 to 5 are features)
y = df.iloc[:, 7].values  # Labels (assuming the action column is column 6)

# Convert to PyTorch tensors
X = torch.tensor(X, dtype=torch.float32).to(device)
y = torch.tensor(y, dtype=torch.long).to(device)

# Split the dataset into training and testing
X_train, X_test = X[:int(0.8 * len(X))], X[int(0.2 * len(X)):]
y_train, y_test = y[:int(0.8 * len(y))], y[int(0.2 * len(y)):]

# train_dataset = TensorDataset(X_train, y_train)
# train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

# Initialize the model, loss function, and optimizer
model = DinoModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Then we load training data from the CSV files where we stored feature data and correspondingly their action. The input features (X) are extracted from columns 1-6, and the target labels (y) are taken from column 7, which are the actions the model predicts. The data is split into training and testing sets and converted into PyTorch tensors. The model is then trained using Adam optimizer and CrossEntropyLoss as the loss function, which are suitable for classification tasks.

```
# Training loop
for epoch in range(10000):
    model.train()  # Ensure model is in training mode
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    print(f'Epoch {epoch} Loss: {loss.item()}')

from torch.utils.data import DataLoader, TensorDataset


# Wrap the testing data into a DataLoader
test_dataset = TensorDataset(X_test, y_test)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

# Evaluate the model
model.eval()  # Set the model to evaluation mode
correct = 0
total = 0

with torch.no_grad():
    for X_batch, y_batch in test_loader:  # Loop through mini-batches
        outputs = model(X_batch)  # Forward pass
        _, predicted = torch.max(outputs, 1)  # Get the predicted class
        total += y_batch.size(0)  # Accumulate total samples
        correct += (predicted == y_batch).sum().item()  # Compare predictions to true labels

accuracy = 100 * correct / total
print(f'Accuracy: {accuracy}%')


# Save the trained model
torch.save(model.state_dict(), 'model.pth')
```

A loop then runs for 10,000 epochs, where in each epoch, the model is trained on the training data, and the optimizer updates the model's parameters based on the loss. The model is then evaluated on a test dataset. The accuracy of the model is checked after by comparing the predicted actions to the actual actions in the test set.After training and evaluation, the model's state is saved to a file (model.pth) for later use.

Genetic_Model.py (The most important part!!!)

```
# Enhanced State Representation
def get_game_state(dinosaur, obstacles, velocity):
    next_obstacles = [obs for obs in obstacles if obs.x+45 > dinosaur.x]
    if len(next_obstacles) > 0:
        next_obstacle = next_obstacles[0]
        # second_obstacle = next_obstacles[1] if len(next_obstacles) > 1 else None

        # Additional state features
        state = [
            next_obstacle.x - dinosaur.x,  # Distance to next obstacle
            next_obstacle.y,  # Vertical distance
            next_obstacle.size,            # Size of next obstacle
            # second_obstacle.x - dinosaur.x if second_obstacle else WIDTH,  # Distance to second obstacle
            velocity,                      # Current game velocity
            1 if dinosaur.is_jumping else 0,  # Is the dinosaur jumping?
            1 if dinosaur.is_ducking else 0,  # Is the dinosaur ducking?
        ]
    else:
        # Default state when no obstacles are nearby
        state = [WIDTH, 0, 0, velocity, 0, 0]

    return np.array(state, dtype=np.float32)
```

- Get_game_state: This is the most important function, as it records the current state of the game which is then fed to the neural network model to predict the next action of the dinosaur.

```
def crossover(parent1, parent2):
    """Cross two neural networks to produce a child network."""
    child = DinoModel()
    with torch.no_grad():
        for child_param, param1, param2 in zip(child.parameters(), parent1.parameters(), parent2.parameters()):
            # Randomly select weights from each parent
            mask = torch.rand_like(param1) > 0.5
            child_param.copy_(param1 * mask + param2 * ~mask)
    return child

def mutate(agent, mutation_rate=0.1):
    """Apply random mutations to the agent's parameters."""
    with torch.no_grad():
        for param in agent.parameters():
            mutation_mask = torch.rand_like(param) < mutation_rate
            param.add_(mutation_mask * torch.randn_like(param) * 0.01)  # Small random mutations
    return agent
```

This section of code puts the “genetic” in genetic algorithms. It defines a crossing over of 2 members from the same generation, taking combined characteristics of both "children". Also, the mutate function is defined, which randomly mutates, which helps in

```
def run_generation(population, num_dinos=10):
    """Run one generation of the game."""
    game_display = pygame.display.set_mode((WIDTH, HEIGHT))
    dinosaurs = [Dinosaur(GROUND_HEIGHT) for _ in range(num_dinos)]
    agents = [agent.to(device) for agent in population]
    bat = Batsymb(0, 115)
    scores = [0] * num_dinos
    obstacles = []
    lastObstacle = WIDTH
    ground_scroll = 0
    game_timer, velocity = 0, 300
    running = [True] * num_dinos
    last_state = np.array([WIDTH, 0, 0, velocity, 0, 0], dtype=np.float32)
```

the process of evolution.

This function runs one generation of the function where it runs the a specified number of dino population in parallel, each having slight differences/mutations in their parameters, leading them to make different choices at certain events. The final scores for each of these dinos is recorded.

```
def evolve_population(population, scores, num_parents=2, mutation_rate=0.1):
    """Evolve the population based on fitness scores."""
    # Sort by scores (fitness)
    sorted_indices = np.argsort(scores)[::-1]
    top_agents = [population[i] for i in sorted_indices[:num_parents]]

    # Create next generation
    next_population = []
    while len(next_population) < len(population):
        parent1, parent2 = random.sample(top_agents, 2)
        child = crossover(parent1, parent2)
        child = mutate(child, mutation_rate)
        next_population.append(child)

    return next_population
```

This is the function that chooses the best child from each preceding generation to feed forward to the next. It calls the previously defined crossover and mutate functions, in order to create the new generation's population. Basically, it chooses the population which had the maximum score in the previous generation and uses it to create them next generation of the population.

Results

```
pygame 2.6.1 (SDL 2.28.4, Python 3.10.11)
Hello from the pygame community. https://www.pygame.org/contribute.html
Using device: cpu
Starting evolution with 25 agents for 10 generations
Selection: tournament, Crossover: uniform, Mutation: adaptive
------------------------------------------------------------
Generation 1/10
  Max Score: 18.0
  Avg Score: 18.0
  Diversity: 0.0000
  Score Range: 18.0 - 18.0
------------------------------------------------------------
Generation 2/10
  Max Score: 37395.0
  Avg Score: 3246.3
  Diversity: 3.1436
  Score Range: 18.0 - 37395.0
------------------------------------------------------------
Generation 3/10
  Max Score: 13790.0
  Avg Score: 4854.3
  Diversity: 4.7844
  Score Range: 83.0 - 13790.0
------------------------------------------------------------
Generation 4/10
  Max Score: 19532.0
  Avg Score: 4491.1
  Diversity: 4.8563
  Score Range: 48.0 - 19532.0
------------------------------------------------------------
Generation 5/10
  Max Score: 17.0
  Avg Score: 17.0
  Diversity: 4.7868
  Score Range: 17.0 - 17.0
------------------------------------------------------------
```

```
Generation 6/10
  Max Score: 54783.0
  Avg Score: 7006.0
  Diversity: 4.7969
  Score Range: 18.0 - 54783.0
------------------------------------------------------------
Generation 7/10
  Max Score: 53834.0
  Avg Score: 5665.0
  Diversity: 4.5088
  Score Range: 95.0 - 53834.0
------------------------------------------------------------
Generation 8/10
  Max Score: 68412.0
  Avg Score: 10833.4
  Diversity: 4.2385
  Score Range: 72.0 - 68412.0
------------------------------------------------------------
Generation 9/10
  Max Score: 71612.0
  Avg Score: 9937.9
  Diversity: 3.5874
  Score Range: 91.0 - 31612.0
------------------------------------------------------------
Generation 10/10
  Max Score: 71503.0
  Avg Score: 9444.3
  Diversity: 2.6162
  Score Range: 36.0 - 1503.0
------------------------------------------------------------

Evolution Complete!
Best score achieved: 68412.0
Average improvement: 1426.3
Final diversity: 2.6162
Model saved as 'improved_genetic_model.pth'
```

The verbose output of the training session

- Within 6 generations, the algorithm achieves convergence to the optimal agent, aka, optimal model weights of the 5-layer ANN
- Consistently scoring > 50,000

The Model in Action

Links to the final model in action:

1. https://youtu.be/NRz7V8_aCEU (nearly perfect run)
2. https://youtu.be/-wfitVrluaw (perfect run!!)

Issues we faced

We faced quite a few issues with integration of the neural network into our file. We were also confused as to which model to exactly use; whether to use an image-recognition CNN, or to use reinforcement learning, or Deep-Q Learning. Eventually we settled with a combination of Supervised Learning and Genetic Algorithms. Genetic algorithms were something very new to us. We got to know about it after deeply researching about methods. Even after settling on the model, we observed that the model was constantly spamming the jump button, which led it to clear the first few obstacles, but there was very little consistency. Getting it to stop constantly jumping required a lot of tweaking and altering of the score function.

Learnings

While coding this project, we learned pygame and all its libraries from scratch, in order to make the base game. We researched various machine learning algorithms in our explorations to find the best model, from DQN to RL and more. We learned how to do effective prompt engineering (🙃), and also how to use Git.

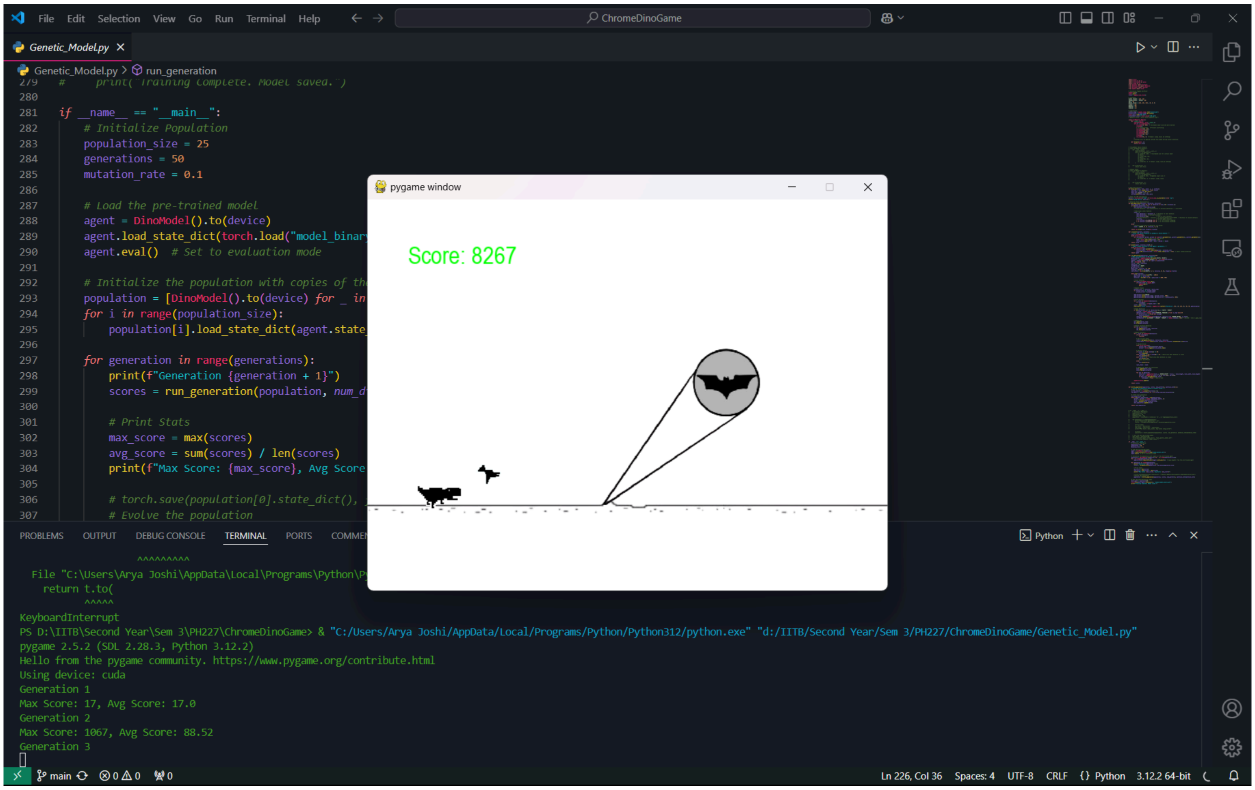
Conclusion

The model is very effective even at higher speeds, consistently ducking and weaving through objects with ease. The combination of supervised learning along with the genetic neural network proves to be extremely effective, and even performs well under variable conditions.

Bibliography

1. https://www.geeksforgeeks.org/genetic-algorithms/

***


pygame window
Score: 8267
Generation 1
Max Score: 17, Avg Score: 17.0
Generation 2
Max Score: 1067, Avg Score: 88.52
Generation 3