DinoAI

> Hi! In this article, I want to share my experience building an AI Dino bot, and I'll guide you through the steps to create it together.

I am just a beginner in this field, so I will probably make a few mistakes. If you have any suggestions, please feel free to share them. I simply want to share my experience with you. <3 This is my first article, so please don't judge me too harshly!

Plan

First step will be thinking about my program architecture. How it should work?What to do..? and more similar questions. In final result i want to only press some buttons and my program gonna compute it all.

> **Short note!**
>
> I don't want to take any information from Google except my screen recorder.

I plan to utilize YOLO in this project.
Step by step how it suppose to be:
download pretrained YOLO model ->
train it on my data ->
Make box predictions ->
measure coordination of dino and nearest object ->
measure distance and class of nearest object ->
If the distance is too short, make the dino jump.

Let's dive in

1 Data

First, we need to simply download the YOLO model. If you don't have `ultralytics` installed, you can install it by opening the command prompt and typing:
`pip install ultralytics #or conda it depends which env u use.It's really not hard step`
Let's create a folder for our first Python file or Jupyter notebook (I prefer using a notebook). I've named the folder 'dino-model'. In this folder, create a file named 'model-creating.ipynb' or 'model-creating.py'.
To start with 'model-creating', we need to import YOLO from the `ultralytics` library:
`from ultralytics import YOLO`
and initialize our yolo model.
Next, we initialize our YOLO model. I plan to use YOLOv8 models, but which one should we pick? Let's check the YOLO documentation to decide.

Performance

Detection (COCO) | Detection (Open Images V7) | Segmentation (COCO) | Classification (ImageNet) | Pose (COCO) | OBB (DOTAv1)

See Detection Docs for usage examples with these models trained on COCO, which include 80 pre-trained classes.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

n - nano
s-small

m-medium
l-large
x-extra large

- **Size** - The size of the input model in pixels.

- **mAPval** - Mean Average Precision; the higher, the better!
- **Speed (CPU)** - The speed of the model when running on a Central Processing Unit (CPU).
- **Speed (A100 TensorRT)** - The speed of the model when running on a high-performance graphics card, like the NVIDIA A100 with TensorRT.
- **Params** - The number of parameters in the model, measured in millions.
- **FLOPs** - The number of Floating Point Operations (FLOPs), which indicates the computational complexity of the model.

We will have around 10 labels, which are not difficult to recognize. Also, we need to focus on speed. Based on this, the best choices for us are **YOLOv8n** or **YOLOv8s**. I chose **YOLOv8n**. Let's download it in our code.

```
from ultralytics import YOLO
model = YOLO('yolov8n.pt')
```

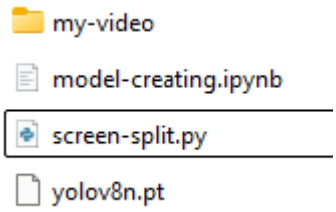
We have the model now, but we lack the training data. Let's create it! I will generate my own data by recording a video of myself playing the Chrome Dino game. For this, we have a wide variety of programs to choose from, and I chose OBS.

I highly recommend storing all your data in one directory. Create a folder named 'my-video' inside the 'dino-model' directory and place the video there.

After creating the video, we need to split it into frames. We can either write our own program to process it or try to find a similar existing program. Here, we will create our custom program.

1.1 Screen-split program

How about creating a new Python file for building our split program? I named it *'screen-split.py'*. Now, our main folder looks like this:



For future use, it's better to create a folder for saving the frames. Let's name it 'screens-from-video'.

In `screen-split.py`, we need to import three libraries for our splitting program:
`cv2` for reading the video,
`os` for creating the full path for each file,
and `uuid` for generating unique IDs.

```
import cv2
import os
import uuid
```

Moving forward, we need to create variables with the names of our paths.
`video_folder = r'my-video/video.mp4'`
`save_folder = r'screens-from-video'`
We create short names because everything is in one folder.If we have files elsewhere, it is better to use the full path.

`cap = cv2.VideoCapture(video_folder)` - VideoCapture return a VideoCapture object, which has various methods to read frames from the video
`fps = cap.get(cv2.CAP_PROP_FPS)` - is a method call that queries the FPS
`frame_count = 0` - for counting frames, we will use this to collect only some frames. We will look at this in more detail later.

`while True:`
`ret, frame = cap.read()` - reads a frame from a video source. `ret` is `True` if the frame is read (or unpacked) successfully, while `frame` is a NumPy array containing the pixel data of the frame.
`if not ret:` - checks if a frame was not read successfully.If it's not read

it's mean the video has ended.

`break`

`if frame_count % fps == 0:` - saves a frame every second by checking if the current frame count is divisible by the video's frames per second (fps).If our fps == 30,it means we save a frame every 30 frames, effectively saving one frame per second.

`save_path = os.path.join(save_folder, str(uuid.uuid4()) + '.jpg')` - I will explain it a bit later

`print(frame_count/30, 'saved successfully')` - print which frame was saved

`cv2.imwrite(save_path, frame)` - and save our file to the path which we create in save_path

frame_count += 1 - counting our frames

`print('Finish')`

`cap.release()` - releases system resources, prevents memory leaks, and ensures proper closure of files or devices.

So,what's going on here

`save_path = os.path.join(save_folder, str(uuid.uuid4()) + '.jpg')`

First, **uuid** is a library used to create unique names, and we use it to ensure name uniqueness—this is logical. We use JPG instead of PNG because JPG has only 3 channels while PNG(additional alpha channel) has 4, which makes PNG files larger in size. Additionally, we store the path in a variable named **save_folder**.

I found a short function on Google that can delete all items in a directory. It can be useful if we want to use this program multiple times.

```
def delete_files_in_directory(directory_path):
try:
files = os.listdir(directory_path)
for file in files:
file_path = os.path.join(directory_path, file)
if os.path.isfile(file_path):
os.remove(file_path)
print("All files deleted successfully.")
```

```
except OSError:
print("Error occurred while deleting files.")
```

And my final code looks like:

```
import cv2
import os
import uuid

def delete_files_in_directory(directory_path):
try:
files = os.listdir(directory_path)
for file in files:
file_path = os.path.join(directory_path, file)
if os.path.isfile(file_path):
os.remove(file_path)
print("All files deleted successfully.")
except OSError:
print("Error occurred while deleting files.")

video_folder = r'my-video/video.mp4'
save_folder = r'screens-from-video'
delete_files_in_directory(save_folder)
cap = cv2.VideoCapture(video_folder)
fps = cap.get(cv2.CAP_PROP_FPS)
frame_count = 0

while True:
ret, frame = cap.read()
if not ret:
break
if frame_count % fps == 0:
save_path = os.path.join(save_folder, str(uuid.uuid4()) +
'.jpg')
print(frame_count/30, 'saved successfully')
cv2.imwrite(save_path, frame)
```

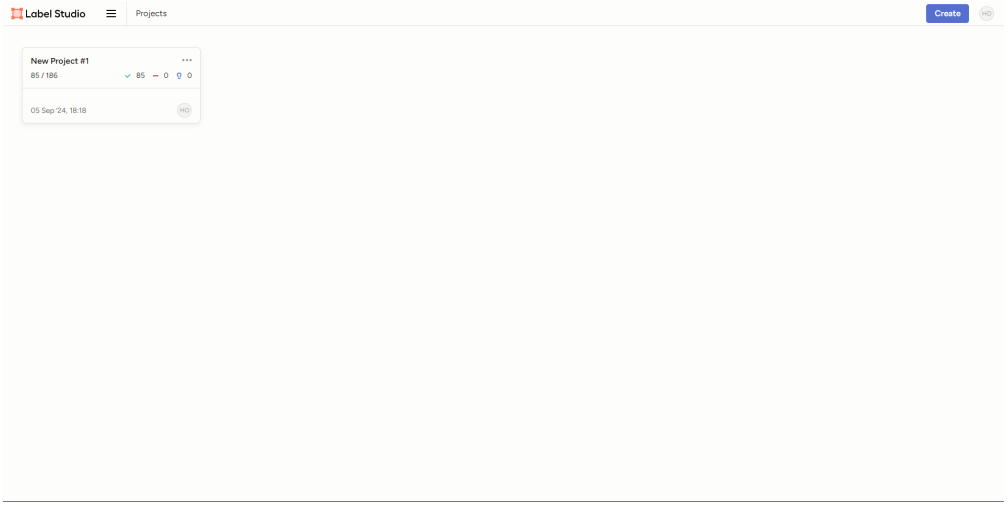
```
frame_count += 1
print('Finish')
cap.release()
```

Nothing complicated; from now on, we have frames that we need to label.

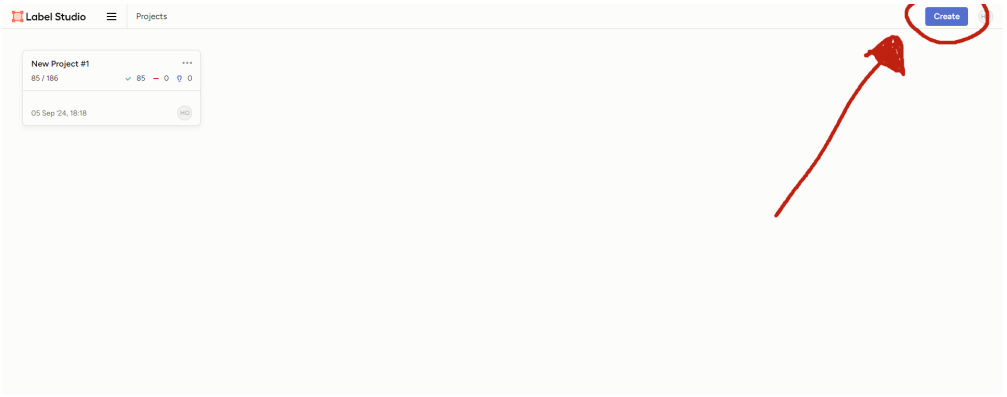
1.2 Labeling

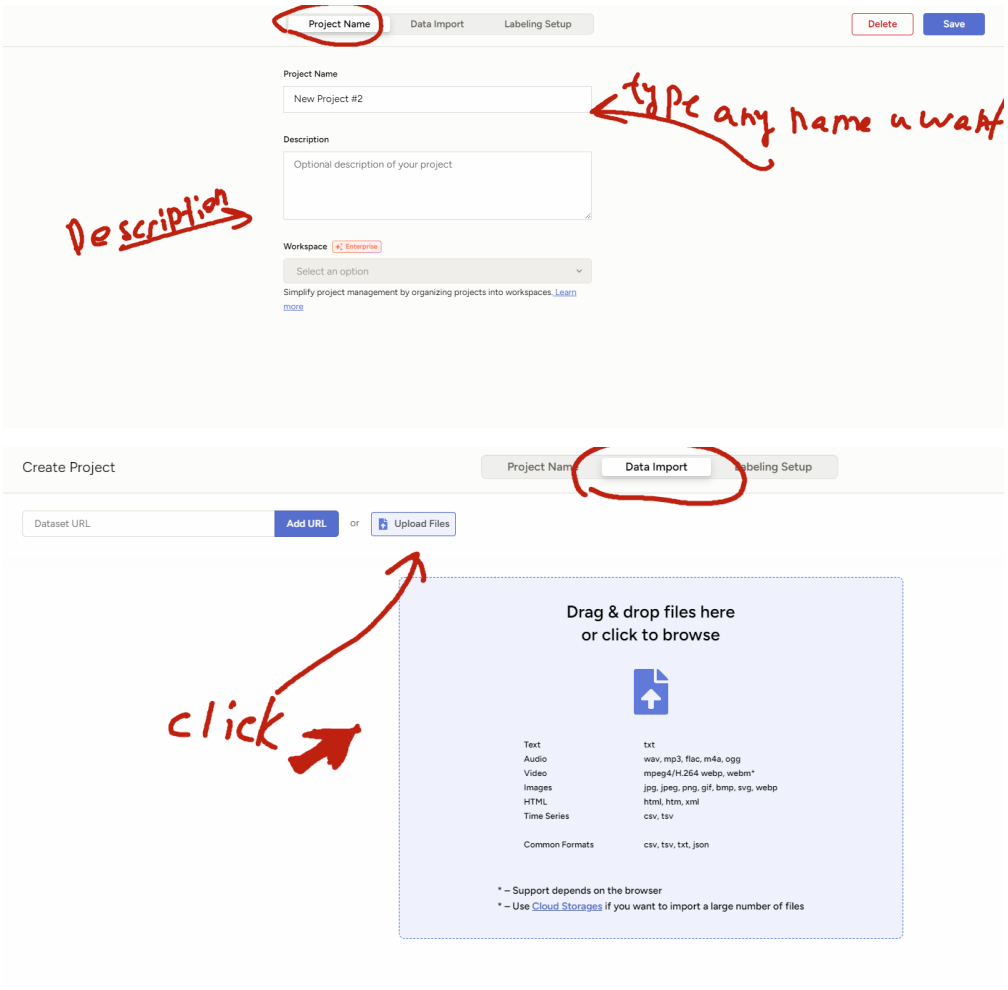
We have several ways to label our photos. I would like to use a program called Label Studio. It is simple to download and run. Open your command prompt and type: `pip install label-studio`, or use the equivalent command for your environment.

To run it, type: `label-studio`. After a short wait, it will open Google Chrome(or other web) with Label Studio ready to use.

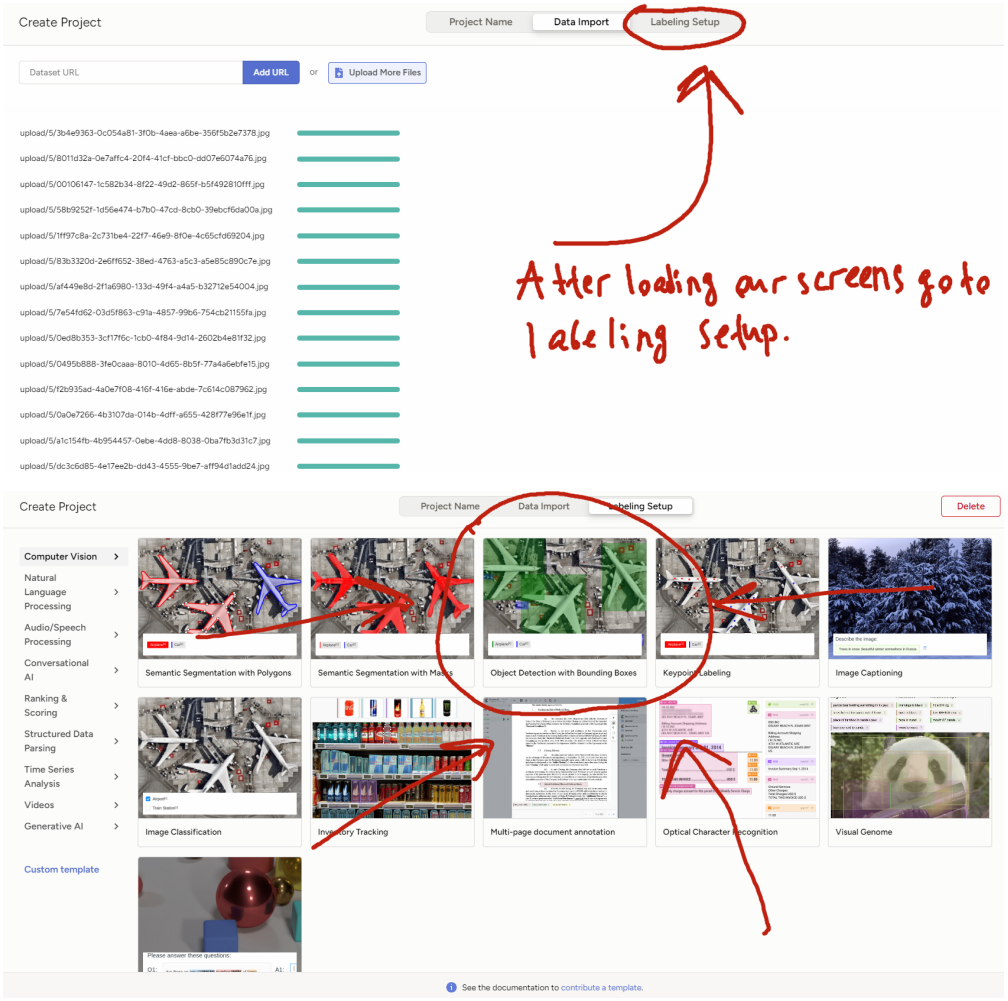


and for creating project folow my step:

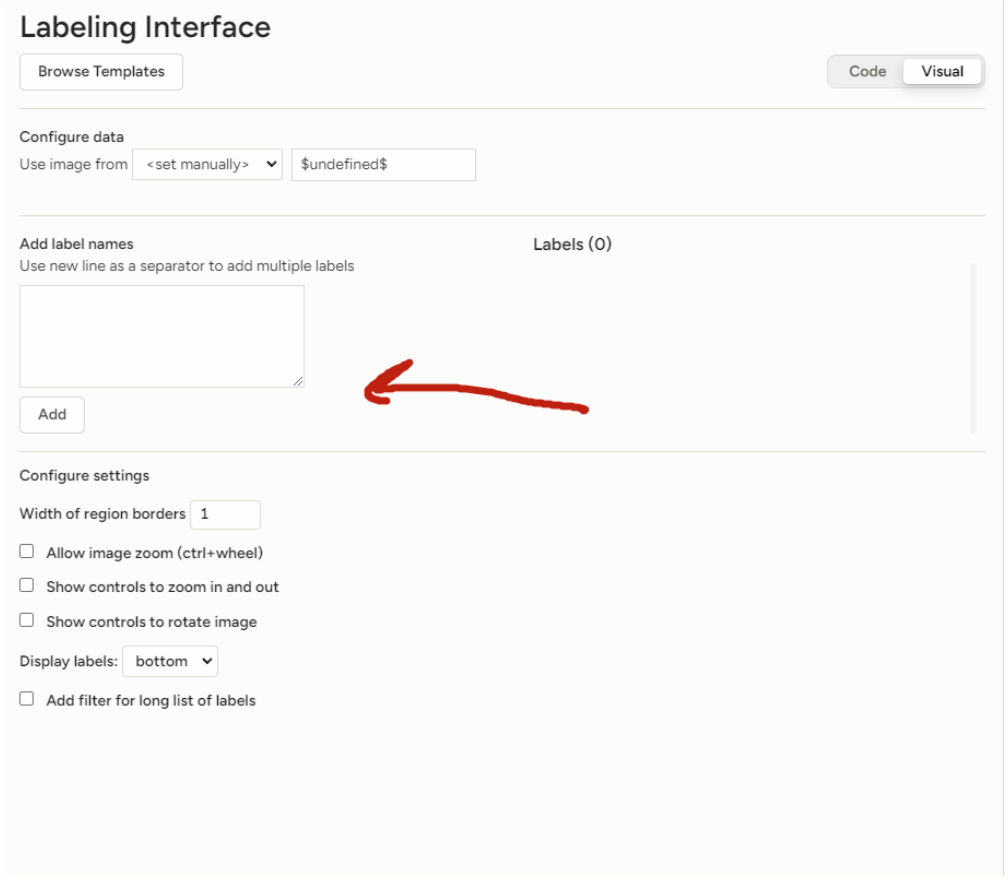




Next, locate the directory where you've saved your frames. If you haven't created any frames yet, you can use mine, which I will upload to GitHub. Everything I create here will be available on GitHub.

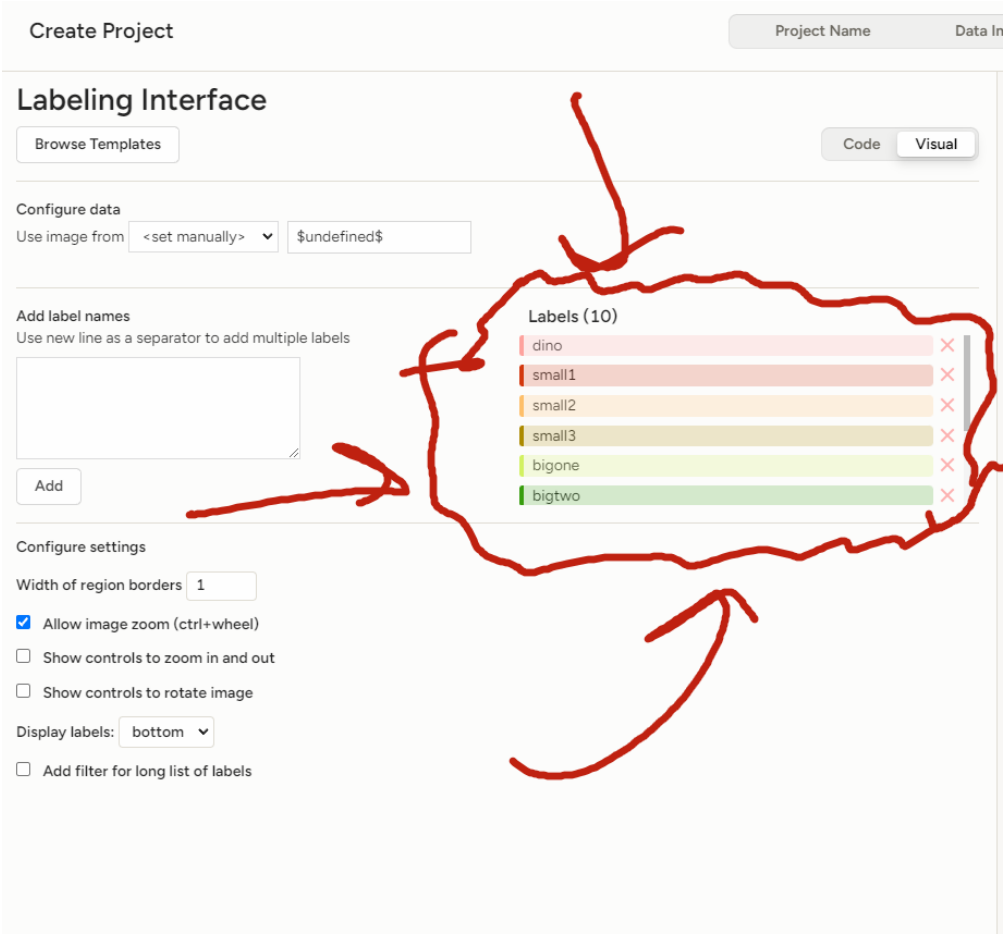


In this context, we have plenty of available annotation categories, but for our simple task, the best choice is object detection with bounding boxes.

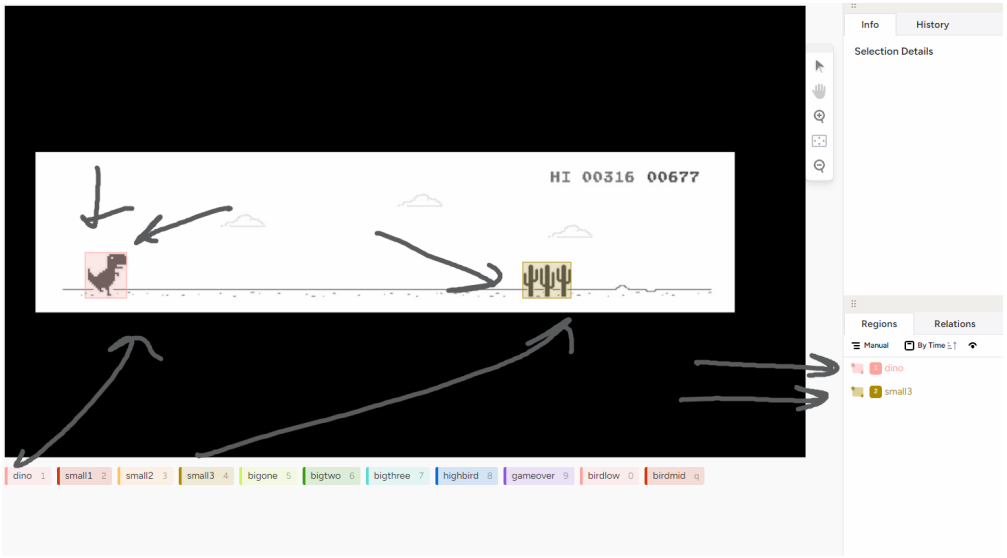


We are going to add the names of objects here. I don't want to include all

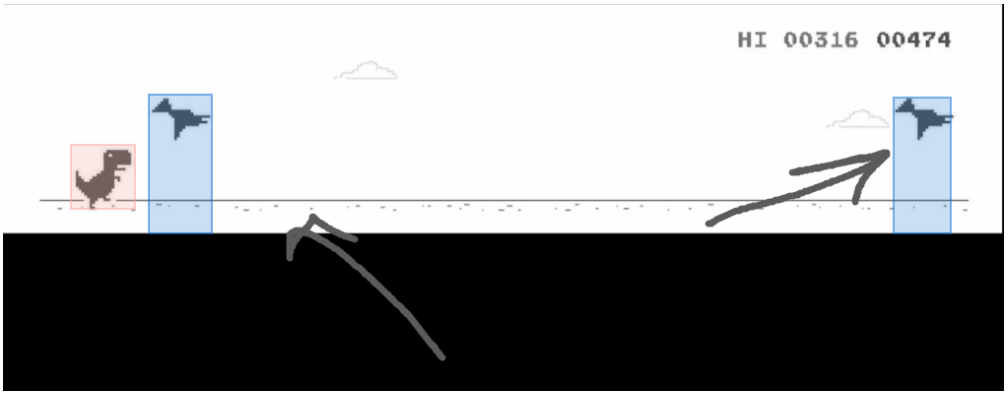
types of birds—only two—because we can ignore those that are in a high position.



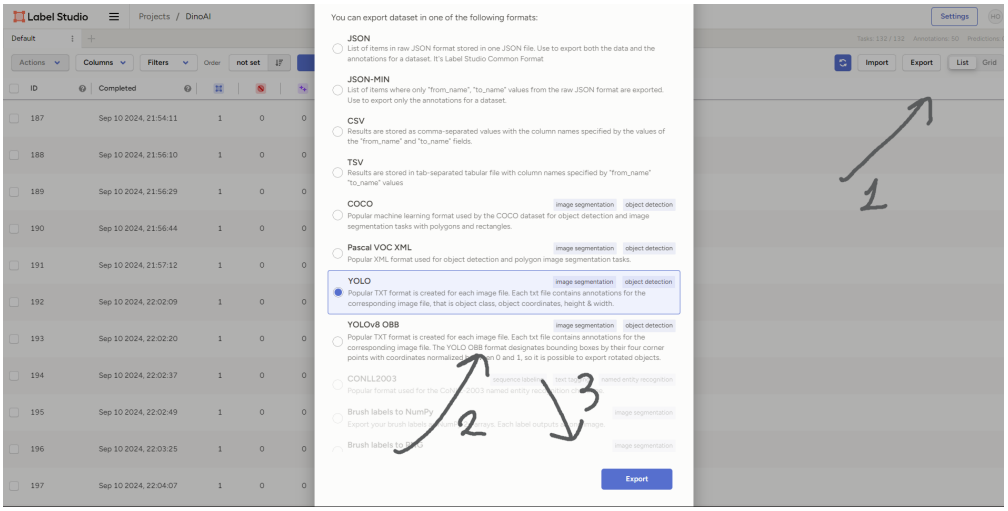
I added 10 labels here with the following names: dino, small1 (for small cactus), small2, small3, bigone, bigtwo, bigthree, birdhigh, birdlow, birdmid, and game over (we will need this later to detect the end of the game). Press 'Save' and start labeling by clicking on the row. It's not difficult, and I think 50 images should be enough for our small model.



Every object should be within a bounding box, except for birds. For birds, we need to define their height levels, as shown in the pictures below.

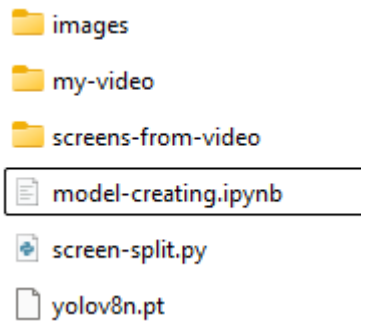


Okay,fine,i make exactly 50 labels,later we will see is it enough or we need more.

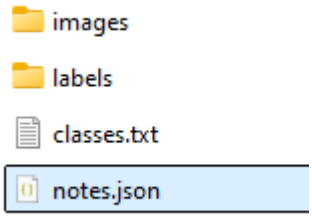


At the end of the labeling process, we need to export all labels in YOLO format. Copy all the files from the zip folder and place them in our working directory. I named this folder 'images'.

Our working directory now looks like this:



and inside the images we have:



1.3 Training

To start training the model, we need to create a YAML file that will serve as our dataset configuration for training. You'll understand this better later. In our main folder, create a file named 'dataset.yaml'. You can fill it out using the documentation (https://docs.ultralytics.com/yolov5/tutorials/train_custom_data/#21-create-datasetyaml)
or follow my short explanation below:

```
path: E:\py\ml\yolov8\works\dino-model - that's my full path
to working directory where we initalize images folder
train: images - that's name of our folder
val: images - same
nc: 11 - how many labels we have
names: [ - all labels which insist
"bigone",
"bigthree",
"bigtwo",
"birdlow",
"birdmid",
```

```
"dino",
"gameover",
"highbird",
"small1",
"small2",
"small3"
]
```

We are now almost ready to train our model. I highly recommend training the model on a GPU (Nvidia only) if it's available. You just need to check the CUDA version of your GPU, visit the PyTorch website at https://pytorch.org/get-started/locally/, and install the PyTorch version that matches your CUDA version. If you are unsure whether your GPU is ready, you can check it in your code editor with the command:

`import torch`
`torch.cuda.is_available()`
Yeap,we prepare all what we need and now we are ready.
Let's open our model-creating file an continuo writing.

```
what we have now
from ultralytics import YOLO
import torch
torch.cuda.is_available()
model = YOLO('yolov8n.pt')
```

and for start training we need write
`model.train(data =`
`'dataset.yaml',epochs=200,imgsz=640,device=0)`
model - our current moel,
data - dataset which we creare recently
epochs - how much cycles we want to train it
imgsize - size in pixels of our model.640 - default

Model	size (pixels)
YOLOv8n	640
YOLOv8s	640
YOLOv8m	640
YOLOv8l	640
YOLOv8x	640

and device where 0 is gpu and 1 cpu
and we can start!this process isn't fast:(
and yea,finish training

```
from ultralytics import YOLO
import torch
```

[2] ✓ 5.0s

```
torch.cuda.is_available()
```

[3] ✓ 0.0s

```
True
```

```
model = YOLO('yolov8n.pt')
```

[4] ✓ 0.0s

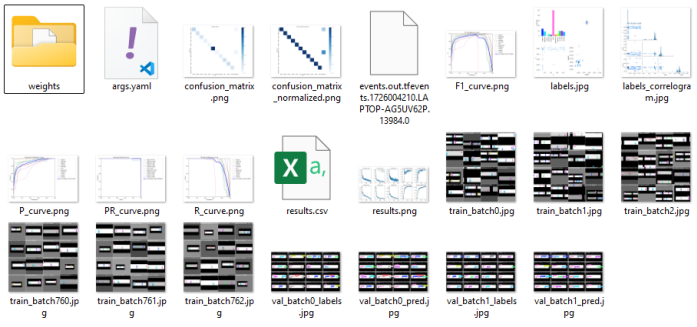
```
model.train(data = 'dataset.yaml',epochs=200,imgsz=640,device=0)
```

[5] ✓ 5m 41.1s

Outputs are collapsed ...

There are more than 500 outputs, show more (open the raw output data in a text editor) ...

A folder named 'runs' has now been created in our main directory. If we open it, we will find some files containing information about our training runs.



There are plenty of graphs and other visualizations here. We can check our confusion matrix to see the results, or we can open `results.csv` for more detailed information.

A1 epoch, train/box_loss, train/cls_loss, train/dfl_loss, metrics/precision(B), metrics/recall(B),

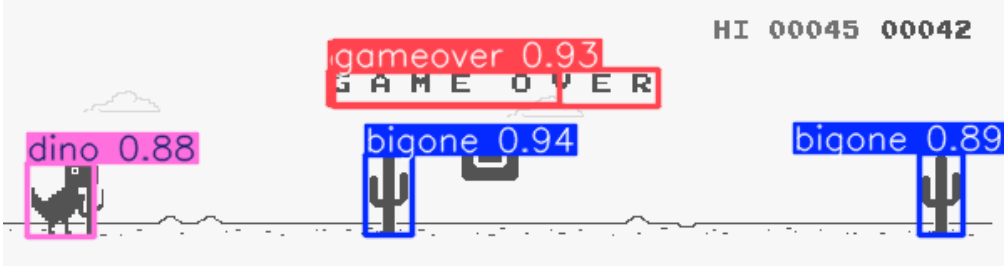
	A	B	C	D	E	F	G	H	I	J	K	L	M	N
37		186,	0.68958,	0.68222,		0.86392,		0.92775,	0.99638,		0.99415,	0.83061,		0.65105,
38		187,	0.69864,	0.68018,		0.84708,		0.92638,	0.99682,		0.99415,	0.84465,		0.64543,
39		188,	0.77067,	0.67707,		0.84752,		0.92543,	0.99721,		0.99415,	0.85228,		0.63705,
90		189,	0.71786,	0.66844,		0.84334,		0.92454,	0.99764,		0.99415,	0.84871,		0.63423,
91		190,	0.66017,	0.68485,		0.83422,		0.92886,	0.99753,		0.99373,	0.8458,		0.63155,
92		191,	0.71311,	0.81327,		0.84834,		0.93507,	0.99717,		0.99373,	0.85159,		0.62867,
93		192,	0.69172,	0.74986,		0.87818,		0.93957,	0.99686,		0.9933,	0.85662,		0.62767,
94		193,	0.6977,	0.74581,		0.85125,		0.94391,	0.9963,		0.9933,	0.85335,		0.62741,
95		194,	0.72417,	0.75919,		0.86009,		0.94701,	0.99579,		0.9933,	0.85558,		0.62553,
96		195,	0.7178,	0.7362,		0.82858,		0.94774,	0.99556,		0.9933,	0.85872,		0.62589,
97		196,	0.65881,	0.77537,		0.82428,		0.94782,	0.99528,		0.99373,	0.85895,		0.62644,
98		197,	0.62958,	0.73229,		0.83171,		0.94935,	0.99497,		0.99373,	0.8539,		0.62714,
99		198,	0.66785,	0.74214,		0.83658,		0.95045,	0.99474,		0.99373,	0.85309,		0.62799,
00		199,	0.71561,	0.75027,		0.84797,		0.95137,	0.99451,		0.99373,	0.84627,		0.62831,
01		200,	0.67248,	0.75408,		0.84584,		0.95233,	0.99424,		0.99373,	0.85757,		0.62818,

Columns 5 and 6 show the results of the tests. Values like 0.95 and 0.99 look perfect for our needs, so let's save our model with the following command: `model.save('modeldino.pt')` or use any other name you prefer. We will test our model using a random screenshot of the Dino game window. You can take a screenshot yourself or use mine.

To test our model, we first need to load it using `model = YOLO('modeldino.pt')` `We also need to specify the path where our image is located:` img = r"E:\Zrzut ekranu 2024-09-11 143217.png"`
To make predictions, print:"

```
results = model(img) - make predictions
result = results[0] - unpack our predictions,because of yolo
model can make predictions for several images in one
time,and so if we make only one we need unpack only one
result.show() - show it
```

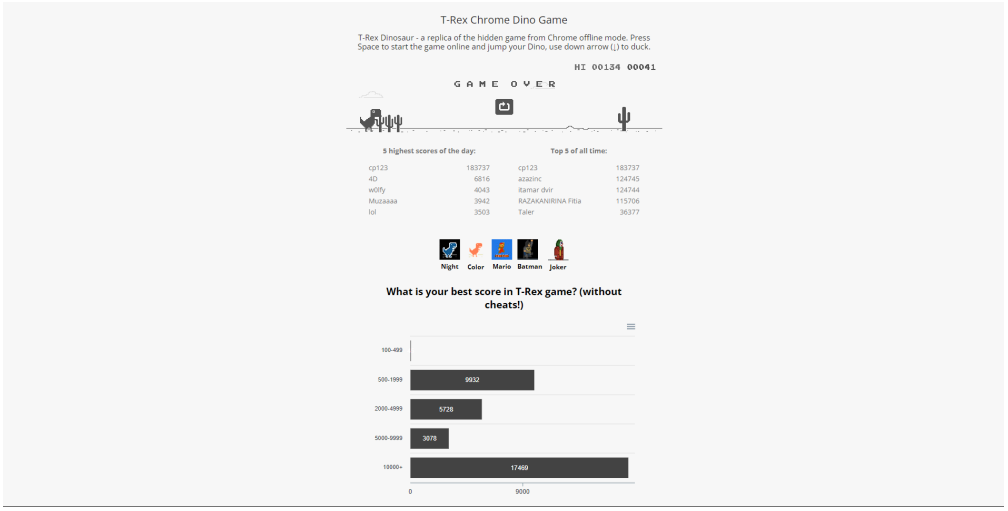
I got this predictions



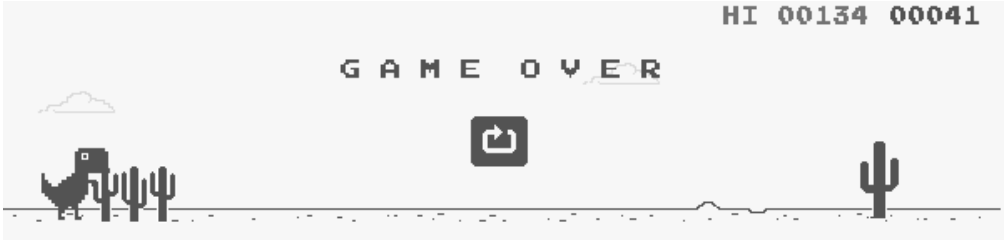
Not perfecrly,but looks fine we don't need to predict collapsed objects

2. Make simple AI

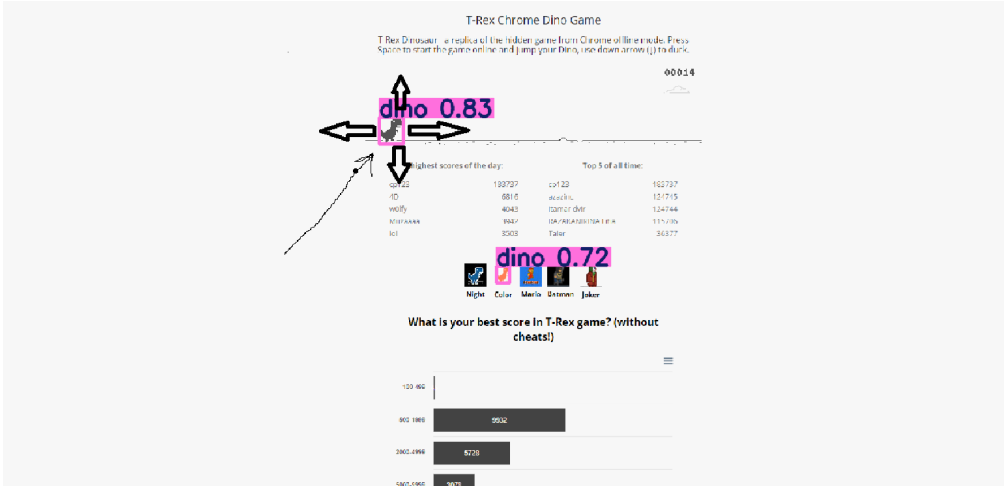
In case I've already trained my model on small-sized pictures, I need a part of the project that will help us scale from a large window to a smaller one. For example, from this:



to this



My idea is to locate the dino in the large window and extend its borders to define the coordinates for a smaller window.



We take the highest prediction label of the dino's coordinates and extend them from the x and y axes. This will become clearer later.

Let's create a new file named 'ai.py' in our working directory and start writing our code.

```
import cv2
from ultralytics import YOLO
import numpy as np
from mss import mss
from pynput.keyboard import Key, Controller
import threading
import time
```

Import some libraries and start with basic functions. The first function will return two types of dictionaries: the first dictionary will have unique values as keys, and the second will have unique object numbers as keys. We will use the first dictionary to detect the dino and select the screen area, and the second one to measure the distance between the dino and the nearest objects.

```
def get_dict(result, for_distance=False):
    my_dict = dict()
    if not for_distance: - if this dict for dino detection
        for n in range(len(result.boxes.cls)):
            if int(result.boxes.cls[n]) in my_dict.keys()
and my_dict[int(result.boxes.cls[n])][1]
>float(result.boxes.conf[n]):
                pass
# in this we check for unique values and in case of non-
unique we compare confedences.If small we don't take and
else we take
            else:
                my_dict[int(result.boxes.cls[n])] = [n,
float(
                    result.boxes.conf[n]),
result.boxes.xyxy[n]]
#here we take in case if we found same key with more
confedence
    if for_distance:
```

```
# in this if we just take all values
        for n in range(len(result.boxes.cls)):
            my_dict[n] = [int(result.boxes.cls[n]),
result.boxes.xyxy[n]]
    return my_dict
```

also we will have function for return values for detecting screen

```
def screendetect(arg, result):
    """
    Function to detect screen coordinates based on the
highest confidence label.
    Parameters:
    arg - index of the highest confidence argument.
    result - the result object which contains the detected
boxes (e.g., ***result = results[0]***).
    ------
    Returns the coordinates x1, y1, x2, y2 after adjusting
them.
    ------
    """
    # Extract the coordinates (x1, y1, x2, y2) of the
bounding box with the highest confidence score
    x1, y1, x2, y2 = result.boxes.xyxy[arg]  # We take
coordinates of the highest confidence label
    # Round the coordinates to avoid floating-point
precision issues
    x1, y1, x2, y2 = x1.round(), y1.round(), x2.round(),
y2.round()  # Just round it
    # Adjust the coordinates to better fit the target object
(e.g., dino)
    x1, y1, x2, y2 = x1 - 50, y1 + 60, x2 + 550, y2 - 130
    # Return the coordinates as integers to avoid problems
that can occur while slicing arrays
    return int(x1.item()), int(y1.item()), int(x2.item()),
```

```
int(y2.item())  # Returning integer values to avoid slicing
issues
```

and function to return dino,it's small but comfortable for writing:

```
def getdino(dict_):
    return dict_[5][0]
```

Moving forward, we will add more functions later, but for now, let's start with detecting the screen.

```
# Load the YOLO model with the specified weights file; set
verbose to False to suppress output
model = YOLO("modeldino.pt", verbose=False)
# Initialize mss for screen capturing
sct = mss()
# Select the monitor to capture (second monitor in this
case)
monitor = sct.monitors[2]
# Set up a flag to control when to stop the loop
stop = False
# Start an infinite loop to capture and process screen
images
while True:
    # Break the loop if stop flag is set to True
    if stop:
        break
    # Capture a screenshot of the defined monitor area
    screen_shot = sct.grab(monitor)
    # Convert the screenshot to a NumPy array
    img = np.array(screen_shot)
    # Convert the image from BGRA to BGR color format
    img = cv2.cvtColor(img, cv2.COLOR_BGRA2BGR)
```

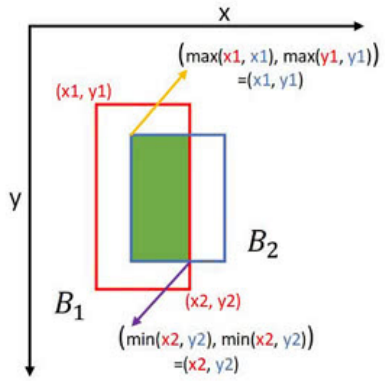
```
    # Use the YOLO model to predict objects in the image
with a confidence threshold of 0.5
    results = model.predict(source=img, imgsz=1920,
conf=0.5, verbose=False)
    # Get the first result from the predictions
    result = results[0]
    # Loop through each detected class in the result
    for clss in result.boxes.cls:
        # Print the detected class as an integer
        print(int(clss))
        # If the detected class is 5 (target class, e.g.,
"dino")
        if int(clss) == 5:
            # Get a dictionary of detected results
            dict_ = get_dict(result)
            # Check if "dino" is present in the detected
results
            dino_present = getdino(dict_)
            # Get the bounding box coordinates for the
detected "dino"
            x1, y1, x2, y2 = screendetect(dino_present,
result)
            # Set the stop flag to True to break the loop
            stop = True
            break
```

From now on, we have the coordinates to use in our model. We will capture the screens within the zone defined by these coordinates. To achieve this, we need to assign the coordinates to the monitor property in `mss`.

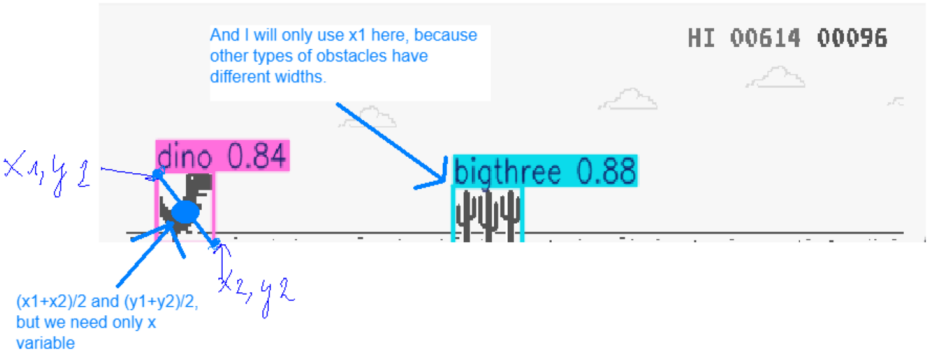
```
monitor = {
    'top': y2,
    'left': x1,
    'width': x2-x1,
```

```
    'height': y1-y2,
}
```

They may look a bit off, but this is all due to the coordinates provided by YOLO in the bounding boxes.



So, we have our screen detection set up. Now, we need to create a basic function to measure the distance between objects.



It will look like this: we will measure the distance between the middle x-coordinate of the dino and the nearest obstacle point.

```
def dino_and_near_distance(dict_):
    # Initialize an empty string to store the key associated
with the "dino"
    key_dino = ''
    # Iterate through the dictionary to find the "dino"
object (with label 5)
    for key, val in dict_.items():
        if val[0] == 5:  # Check if the first element in the
value tuple is 5 (label for "dino")
            key_dino = key  # Store the key associated with
"dino"
```

```
            break  # Exit the loop once "dino" is found
    # Calculate the x-coordinate of the center of the "dino"
bounding box
    x_dino = int(dict_[key_dino][1][0].item() +
dict_[key_dino][1][2].item()) / 2
    # Initialize an empty dictionary to store distances from
the "dino"
    dict_of_x_distances = dict()

    # Iterate through the dictionary to calculate the
distances of other objects from the "dino"
    for key, val in dict_.items():
        dist = int(dict_[key][1][0]) - x_dino  # Calculate
distance of the object's left x-coordinate from the "dino"
        if key != key_dino and dist > 0:  # Check if it's
not "dino" and distance is positive (to the right of "dino")
            dict_of_x_distances[dict_[key][0]] = dist  #
Store the distance with the object label as the key
    # Initialize variables to store the minimum distance and
the corresponding object name
    minn = 100000  # Set a high initial value for minimum
distance
    name = ''  # Initialize an empty string to store the
object name
    # Iterate through the dictionary of distances to find
the nearest object
    for key, val in dict_of_x_distances.items():
        if val < minn:  # Check if the current distance is
smaller than the stored minimum distance
            minn = val  # Update the minimum distance
            name = my_map[key]  # Get the object name using
a mapping dictionary `my_map` our mapping dictionary i will
put below
    # Return the name of the nearest object and its distance
```

```
from the "dino"
    return name, minn
```

mapping dictionry:

```
my_map = {
    0: "bigone",
    1: "bigthree",
    2: "bigtwo",
    3: "birdlow",
    4: "birdmid",
    5: "dino",
    6: "gameover",
    7: "highbird",
    8: "small1",
    9: "small2",
    10: "small3"
}
```

We have a function that measures the distance, and now we need to start capturing our screen. If the distance is lower than our input threshold, we will press the spacebar. We will use two types of jumps: short and high. Short jumps are for small obstacles, and high jumps are for large ones. For this, we need to create two arrays to store the names of the labels.

```
small_space = ["birdlow", "birdmid", "gameover", "small1",
"small2", ]
long_space = ["bigthree", "bigtwo", "bigone", "small3"]
```

And now we can almost create the final result. We need a function that will press the 'space' key. If we use `time.sleep()` in our code, it will cause freezes and prevent us from looking ahead. Instead, let's provide a function that avoids this issue:

```
in_space = False
def press_space(duration):
    global in_space
    in_space = True
    keyboard.press(Key.space)
    time.sleep(duration)
    keyboard.release(Key.space)
    in_space = False
```

We also need to initialize the controller to interact with the keyboard: `keyboard = Controller()`.

```
while True:  # Infinite loop for continuous screen capture
and processing
    screenshot = sct.grab(monitor)  # Capture screen from
the defined monitor
    frame = np.array(screenshot)  # Convert screenshot to a
NumPy array
    frame = cv2.cvtColor(frame, cv2.COLOR_BGRA2BGR)  #
Convert from BGRA to BGR format for OpenCV
    results = model(frame, verbose=False)  # Run object
detection on the frame
    result = results[0]  # Get the first detection result
    dict_ = get_dict(result, for_distance=True)  # Convert
results to a dict format optimized for distance calculation
    name, dist = dino_and_near_distance(dict_)  # Get
nearest object name and distance from "dino"
    print(name)  # Print the name of the nearest object
    # Check if distance is less than or equal to 100 and the
"in_space" flag is not set
    if dist <= 100 and not in_space:
        if name in long_space:  # If in long_space category,
press space for longer
            threading.Thread(target=press_space, args=
```

```
(0.2,)).start()
        if name in short_space:  # If in short_space
category, press space for shorter
            threading.Thread(target=press_space, args=
(0.01,)).start()
    cv2.imshow('aaaa', result.plot())  # Display frame with
detected objects
    if cv2.waitKey(10) & 0xFF == ord('q'):  # Exit loop if
'q' is pressed
        break
cv2.destroyAllWindows()  # Close all OpenCV windows after
exiting the loop
```

And now our code is ready.here is full version of code:

```
import cv2
from ultralytics import YOLO
import numpy as np
from mss import mss
from pynput.keyboard import Key, Controller
import threading
import time


keyboard = Controller()
in_space = False

short_space = ["birdlow", "birdmid", "gameover", "small1",
"small2", ]
long_space = ["bigthree", "bigtwo", "bigone", "small3"]
my_map = {
    0: "bigone",
    1: "bigthree",
    2: "bigtwo",
    3: "birdlow",
```

```
    4: "birdmid",
    5: "dino",
    6: "gameover",
    7: "highbird",
    8: "small1",
    9: "small2",
    10: "small3"
}
def press_space(duration):
    global in_space
    in_space = True
    keyboard.press(Key.space)
    time.sleep(duration)
    keyboard.release(Key.space)
    in_space = False
def screendetect(arg, result):
    """
    unpack with x1,y1,x2,y2
    """
    x1, y1, x2, y2 = result.boxes.xyxy[arg]
    x1, y1, x2, y2 = x1.round(), y1.round(), x2.round(),
y2.round()
    x1, y1, x2, y2 = x1-50, y1+60, x2+550, y2-130
    return int(x1.item()), int(y1.item()), int(x2.item()),
int(y2.item())
def get_dict(result, for_distance=False):
    my_dict = dict()
    if not for_distance:
        for n in range(len(result.boxes.cls)):
            if int(result.boxes.cls[n]) in my_dict.keys()
and my_dict[int(result.boxes.cls[n])][1] >
float(result.boxes.conf[n]):
                pass
            else:
                my_dict[int(result.boxes.cls[n])] = [n,
```

```
float(
                    result.boxes.conf[n]),
result.boxes.xyxy[n]]
    if for_distance:
        for n in range(len(result.boxes.cls)):
            my_dict[n] = [int(result.boxes.cls[n]),
result.boxes.xyxy[n]]
    return my_dict
def getdino(dict_):
    return dict_[5][0]
def dino_and_near_distance(dict_):
    key_dino = ''
    for key, val in dict_.items():
        if val[0] == 5:
            key_dino = key
            break
    x_dino = int(dict_[key_dino][1][0].item() +
dict_[key_dino][1][2].item())/2
    dict_of_x_distances = dict()
    for key, val in dict_.items():
        dist = int(dict_[key][1][0])-x_dino
        if key != key_dino and dist > 0:
            dict_of_x_distances[dict_[key][0]] = dist
    minn = 100000
    name = ''
    for key, val in dict_of_x_distances.items():
        if val < minn:
            minn = val
            name = my_map[key]
    return name, minn
model = YOLO("modeldino.pt", verbose=False)
sct = mss()
monitor = sct.monitors[2]
stop = False
while True:
```

```
    if stop:
        break
    screen_shot = sct.grab(monitor)
    img = np.array(screen_shot)
    img = cv2.cvtColor(img, cv2.COLOR_BGRA2BGR)
    results = model.predict(source=img, imgsz=1920,
conf=0.5, verbose=False)
    result = results[0]
    for clss in result.boxes.cls:
        print(int(clss))
        if int(clss) == 5:
            dict_ = get_dict(result)
            dino_present = getdino(dict_)
            x1, y1, x2, y2 = screendetect(dino_present,
result)
            stop = True
            break
monitor = {
    'top': y2,
    'left': x1,
    'width': x2-x1,
    'height': y1-y2,
}
while True:
    screenshot = sct.grab(monitor)
    frame = np.array(screenshot)
    frame = cv2.cvtColor(frame, cv2.COLOR_BGRA2BGR)
    results = model(frame, verbose=False)
    result = results[0]
    dict_ = get_dict(result, for_distance=True)
    name, dist = dino_and_near_distance(dict_)
    print(name)
    if dist <= 100 and not in_space:
        if name in long_space:
            threading.Thread(target=press_space, args=
```

```
(0.2,)).start()
        if name in short_space:
            threading.Thread(target=press_space, args=
(0.01,)).start()
    cv2.imshow('aaaa', result.plot())
    if cv2.waitKey(10) & 0xFF == ord('q'):
        break
cv2.destroyAllWindows()
```

And now, if we start our AI, it will work, and we can see how it performs. The main challenge is accurately finding the dino, as it might detect another dino, causing the program to malfunction.

Ways to Improve Our Bot's Logic:

- Instead of comparing with a fixed value like 100, we can use a speed coefficient, such as 0.02.
- We can calculate the dynamic threshold using `100 * (1 + 0.02 * (time.time() - start_time))`, where `start_time` is initialized at the beginning of the code.
- If our program detects 'game_over', it will reinitialize `start_time`.

All the code I've written here will be uploaded to my GitHub. For any questions, you can find my social media links on GitHub and contact me there. Good luck!


0:00 / 0:13