LeagueAI: Análisis y Mejora
Fin de Grado

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Raúl Salguero Cárceles

Tutor/es:
Diego Viejo Hernando

Universitat d'Alacant
Universidad de Alicante

Julio 2021

LeagueAI

Automatización del jugador en el videojuego *League of Legends* basado en reconocimiento de imágenes

Raúl Salguero Cárceles

ra_salguero@hotmail.com

Universitat d'Alacant
Universidad de Alicante

Grado en Ingeniería Informática

Curso 2020-2021

Análisis y mejora del proyecto leagueai

Iniciado por Oliver struckmeier en 2017

Resumen

El siguiente trabajo consiste en realizar el análisis y mejora del proyecto LeagueAI (del cual me ha proporcionado permisos su autor original), que tiene como objetivo crear un bot que sea capaz de jugar al videojuego League of Legends a partir del reconocimiento de imágenes mostradas en pantalla.
El foco se encuentra en la utilización y manejo de diversas tecnologías y plataformas como CUDA, PyTorch o YOLO detector con el fin de obtener una detección lo más efectiva y eficiente posible para la automatización del jugador en el videojuego.

SIGLAS Y NOMENCLATURA USADA

LoL ----------League of Legends

Esports----------Electronic-Sports

YOLO----------You Only Look Once

CUDA----------Compute Unified Device Architecture

AI-IA----------Artificial Intelligence-Inteligencia Artificial

GPU----------Graphics Processing Unit

CPU----------Central Processing Unit

RAM----------Random Access Memory

RTX----------Real-Time Ray Tracing GPU

GB----------GigaBytes

MB----------MegaBytes

API----------Application Programming Interface

OpenCV----------Open Computer Vision

FPS----------Frames Per Second

RGB----------Red-Green-Blue

MOBA----------Multiplayer Online Battle Arena

BOT----------Robot

ÍNDICE

Introducción ---- 7
1. ¿Qué es League of Legends? ---- 7
2. IA y LeagueAI ---- 8
Preparando el Entorno ---- 11
1.Prerrequisitos ---- 11
2.CUDA ---- 12
3.Pytorch y TensorFlow ---- 14
PyTorch vs. TensorFlow ---- 15
Primera Aproximación ---- 19
1.Adaptación del código ---- 19
2.Entrenamiento y Conjunto de Datos ---- 20
3.Ejecución y Detecciones ---- 21
4.Toma de Decisiones ---- 23
5.Resultados de la detección ---- 25
Generación del Conjunto de Datos ---- 27
Detector YOLO ---- 34
1.YOLOv3 ---- 36
2.YOLOv4 ---- 39
3.YOLOv5 ---- 40
4.Comparación y Resultados ---- 41
Implementación ---- 49
Conclusiones y Trabajo Futuro ---- 53
1.Conclusiones Extraídas ---- 53
2.Tareas Futuras ---- 54
Bibliografía ---- 55
Agradecimientos ---- 56

Introducción

Conforme avanza la tecnología, es común, y se puede observar más frecuentemente, la aparición de nuevos nichos de mercado. Una de estas industrias que ha crecido considerablemente en los últimos años, y lo sigue haciendo, es la de los Esports o deportes electrónicos. Y uno de los videojuegos que indiscutiblemente se encuentra en la cumbre a día de hoy es el League of Legends[1].

1. ¿Qué es League of Legends?

League of Legends es un videojuego de género MOBA (Multiplayer Online Battle Arena). En este juego, se enfrentan 10 jugadores en equipos de 5c5 con el objetivo de destruir la base o nexo enemigo para ganar la partida.

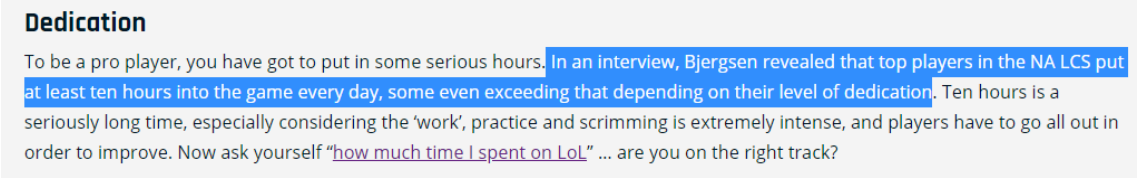
Mapa de League of Legends

El mapa se podría dividir en 4 zonas principales: Top, Medio, Bottom y Jungla (posiciones de las tres líneas marcadas y la jungla o zonas intermedias), donde los jugadores se encontrarían inicialmente en los números marcados en negro, a partir de los cuales irían avanzando conforme progresa la partida para aproximarse a la base enemiga.

En este mismo mapa también se juegan partidas 1c1 donde las reglas son encontrarse en la zona central, donde gana el primer jugador que consiga un asesinato sobre el otro jugador, elimine 100 súbditos o destruya la primera torre.

Este videojuego fue lanzado en Octubre de 2009, fecha desde la que no ha hecho más que crecer tanto en contenido (más de 140 campeones con 'skins', 'chromas' o cambios de color, habilidades únicas, etc.), como en popularidad, contando con más de 100 millones de jugadores en 2020 y competiciones semanalmente tanto a nivel nacional como internacional.

Por lo que no es de extrañar que sea de uno de los más grandes y complicados Esports en los que competir, donde los jugadores profesionales llegan a dedicar decenas de miles de horas.



Dedication

To be a pro player, you have got to put in some serious hours. In an interview, Bjergsen revealed that top players in the NA LCS put at least ten hours into the game every day, some even exceeding that depending on their level of dedication. Ten hours is a seriously long time, especially considering the 'work', practice and scrimming is extremely intense, and players have to go all out in order to improve. Now ask yourself "how much time I spent on LoL" ... are you on the right track?

Referencia de esports.net donde se comenta que los jugadores profesionales de Norte América llegan a dedicar más de 10 horas diarias

2. IA y LeagueAI

Por eso, y debido al alto requerimiento de habilidad que hace falta para competir en este tipo de videojuegos, es normal pensar en la propia tecnología como apoyo para mejorar o alcanzar niveles de jugabilidad que de otra forma llevaría más tiempo o incluso sería inviable. Y, sin duda, la Inteligencia Artificial es una tecnología que tiene mucho que aportar en este ámbito.

De hecho, no es algo nuevo, ya en 1997 el computador Deep Blue, fue capaz de ganar en una partida al entonces campeón del mundo en ajedrez, Garry Kasparov [2].

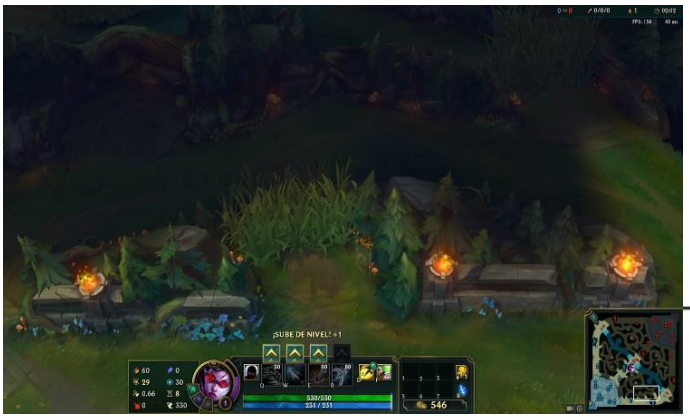
Por tanto, no es de extrañar que la Inteligencia Artificial se aplicase a más ámbitos, como los videojuegos, sobre todo teniendo en cuenta que las

posibilidades de computación actuales son gigantescas en comparación a hace años; cualquier persona con un dispositivo inteligente debería ser capaz de ejecutar, al menos, sistemas sencillos de Inteligencia Artificial.
Es por todo esto que surge el interés de crear un proyecto como LeagueAI.

LeagueAI es un programa o conjunto de programas que tiene el objetivo de crear una Inteligencia Artificial para automatizar las acciones que realizaría un jugador de League of Legends de la misma forma que lo haría un jugador real, es decir, con reconocimiento de imágenes.
LeagueAI está programado en Python, pero utiliza distintas librerías y paquetes como Pytorch para trabajar con tensores y distintos elementos de machine learning (o aprendizaje automático) y también plataformas como CUDA, para poder paralelizar los cálculos y permitir un incremento del rendimiento con el uso de la GPU.

Fotograma de un súbdito enemigo

Y en cuanto al funcionamiento del proyecto, LeagueAI permite, a partir de unos cuantos vídeos de los elementos que aparecen en el videojuego, exportar los fotogramas de esos vídeos, eliminar el fondo y colocar esos elementos generando imágenes que simulan situaciones aleatorias, a veces imposibles dentro del propio juego, para así poder obtener un conjunto de datos utilizado para el entrenamiento en el reconocimiento de imágenes.



Captura del mapa vacía

Captura del mapa procesada

Y haciendo uso de un algoritmo externo para entrenar (YOLO detector), LeagueAI utiliza los pesos generados por este algoritmo para procesar los fotogramas del videojuego en tiempo real y poder mandar acciones al personaje controlado por el jugador sobre la partida.

Mas adelante se detallará todo el proceso.

Preparando el Entorno

1.Prerrequisitos

Lo primero que hay que tener en cuenta son las especificaciones hardware. Para poder lanzar el proyecto, es muy recomendable un computador relativamente potente y preferiblemente que disponga de una GPU de NVIDIA, ya que es necesario para la compatibilidad con la plataforma de CUDA [3].
Por una parte, el hecho de que sea potente no es obligatorio, pero si recomendable ya que más adelante observaremos que no solo para entrenar es necesario una CPU o RAM actuales, sino también una buena GPU para que cuando queramos ejecutar el proyecto junto al videojuego, se puedan paralelizar ambas tareas con una tasa de refresco de fotogramas al menos aceptable (considerando este punto entre 1 y 5 fotogramas por segundo).

El proyecto se ha desarrollado para sistemas Operativos Linux y Windows, pero éste último facilita la instalación de algunos paquetes, y, sobre todo, la compatibilidad con el videojuego.
Teniendo eso en cuenta, el primer paso es instalar el Lenguaje de Programación Python [4]; la versión más actual a fecha de entrega de este proyecto es Python 3.9, y aunque la versión utilizada para el proyecto haya sido Python 3.8, posteriores versiones deberían seguir siendo compatibles. También es recomendable utilizar Visual Studio como entorno de desarrollo por su compatibilidad con CUDA.

Una vez asegurados los prerrequisitos, comienza el proceso de instalación de la principal plataforma de computación, CUDA.

2.CUDA

CUDA es una plataforma de computación en paralelo desarrollada por NVIDIA que permite codificar algoritmos en GPU para explotar las ventajas de ésta respecto a la CPU al poder lanzar un muy alto número de hilos simultáneos utilizando el paralelismo de sus múltiples núcleos [5].

Es por estar desarrollado por NVIDIA que se vuelve necesario utilizar una tarjeta gráfica de esta compañía para poder utilizar CUDA. Dentro de las tarjetas que se suelen comercializar en el mercado, no todas son compatibles o proporcionan la mayor eficiencia con la plataforma.

GeForce and TITAN Products

GPU	Compute Capability
Geforce RTX 3060 Ti	8.6
Geforce RTX 3060	8.6
GeForce RTX 3090	8.6
GeForce RTX 3080	8.6
GeForce RTX 3070	8.6
GeForce GTX 1650 Ti	7.5
NVIDIA TITAN RTX	7.5
Geforce RTX 2080 Ti	7.5
Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5
NVIDIA TITAN V	7.0
NVIDIA TITAN Xp	6.1
NVIDIA TITAN X	6.1
GeForce GTX 1080 Ti	6.1
GeForce GTX 1080	6.1
GeForce GTX 1070 Ti	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 1050	6.1
GeForce GTX TITAN X	5.2

GeForce Notebook Products

GPU	Compute Capability
GeForce RTX 3080	8.6
GeForce RTX 3070	8.6
GeForce RTX 3060	8.6
GeForce RTX 3050 Ti	8.6
GeForce RTX 3050	8.6
Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 980	5.2
GeForce GTX 980M	5.2
GeForce GTX 970M	5.2
GeForce GTX 965M	5.2
GeForce GTX 960M	5.0
GeForce GTX 950M	5.0
GeForce 940M	5.0
GeForce 930M	5.0
GeForce 920M	3.5
GeForce 910M	5.2

Tarjetas gráficas compatibles con CUDA

Y, cuanto más potente sea la tarjeta, mayor será la capacidad de computación con CUDA, siendo por tanto las tarjetas RTX las más efectivas.

Respecto a la versión elegida, actualmente es recomendable usar CUDA 10.1, ya que, aunque en 2020 se lanzó CUDA 11, aún hay muchos programas y plataformas que no se han adaptado a esta nueva versión, por ello, CUDA 10.1 es generalmente más compatible a día de hoy.

Siguiendo los pasos de instalación es sencillo instalar la plataforma, teniendo en cuenta que se cumplen los requisitos de Drivers correspondientes a la tarjeta gráfica. Adicionalmente también será necesario instalar la librería cuDNN correspondiente, la cual es una librería de aceleración de GPU para desarrollar aplicaciones de redes neuronales. La versión compatible con CUDA 10.1 es cuDNN 7.6.5.

```
C:\Users\Raul>nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Sun_Jul_28_19:12:52_Pacific_Daylight_Time_2019
Cuda compilation tools, release 10.1, V10.1.243
```

Versión de CUDA 10.1 instalada

```
 */

/*   cudnn : Neural Networks Library

 */

#if !defined(CUDNN_H_)
#define CUDNN_H_

#define CUDNN_MAJOR 7
#define CUDNN_MINOR 6
#define CUDNN_PATCHLEVEL 5

#define CUDNN_VERSION (CUDNN_MAJOR * 1000 + CUDNN_MINOR * 100 + CUDNN_PATCHLEVEL)

#include "driver_types.h"
#include <cuda_runtime.h>
#include <stdint.h>
```

Versión de cuDNN instalada correspondiente a 7.6.5
~\CUDA\v10.1\include\cudnn.h

3.Pytorch y TensorFlow

Otro de los requisitos vitales en el proyecto son las librerías de TensorFlow y Pytorch que se detallarán y compararán en el siguiente apartado. Y, aunque la versión desarrollada y mejorada haya sido la de Pytorch, existe una primera versión de LeagueAI desarrollada en un único script con TensorFlow con la que se pueden hacer comparaciones y estadísticas.

Por eso, junto a ambas librerías se instalan diversos paquetes para disponer de diversas herramientas que se utilizarán en el código.

Algunas de estas librerías o paquetes usados son:

- MSS [6] (Para obtener los fotogramas mostrados en pantalla)
- Pillow [7] (Para transformar en datos manejables los fotogramas)
- OpenCV [8] (Para trabajar sobre estos datos/fotogramas)
- NumPy [9] (Es la librería de Python que permite trabajar con Arrays y matrices de diferentes dimensiones, proporcionando una gran colección de funciones matemáticas de alto nivel)

Para la instalación de estos paquetes, se ha hecho uso de los administradores de paquetes Conda y PIP, que facilitan su instalación, uso y despliegue.

PyTorch vs. TensorFlow

Si hablamos de Deep Learning o Aprendizaje Profundo, es complicado no encontrarse con las librerías de PyTorch o TensorFlow.
En los últimos años PyTorch y Tensorflow se han convertido en dos excelentes herramientas para el desarrollo de redes neuronales.

Tensorflow fue lanzado inicialmente en 2015, desarrollado por el Google Brain Team.
PyTorch, desarrollado por el Laboratorio de Investigación de IA de Facebook fue lanzado en 2016.

Ambas son librerías de código abierto centradas en el Aprendizaje Automático que proporcionan características de alto nivel y generación de grafos sobre el rendimiento del modelo durante el entrenamiento. Proporcionan clases de Python para programar en este lenguaje, aunque PyTorch también permite hacerlo en C/C++, así como TensorFlow introdujo interfaces del lenguaje R.

Tanto PyTorch como TensorFlow operan con Tensores, que son listas individuales o listas de listas que contienen números, es decir, una generalización más amplia de matrices o vectores, utilizando para ello la librería de NumPy.
Y en cuanto a la aceleración de GPU, PyTorch está escrito con la extensión de CUDA, lo que hace sencillo su uso. Y TensorFlow tiene soporte de CUDA.

Para entrar más en detalle, a finales de 2019 en la Universidad de Carolina Occidental se hizo un estudio más completo comparando el rendimiento de ambos Frameworks [10]

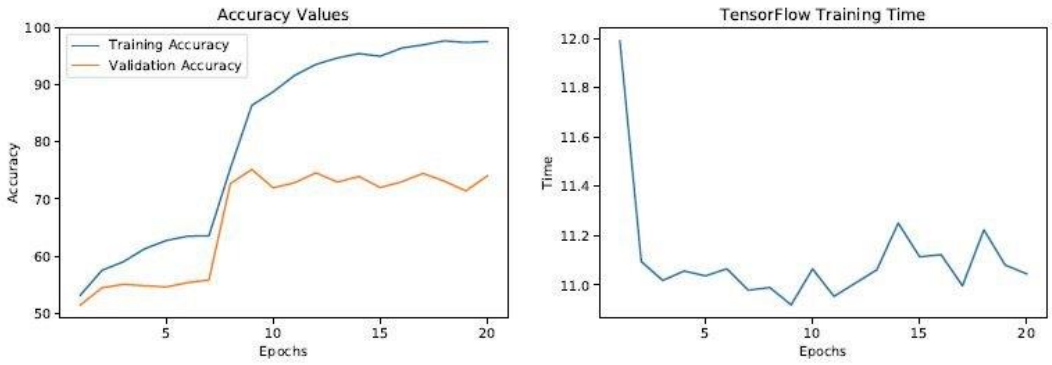


Figure 1: TensorFlow Accuracy and Training Time

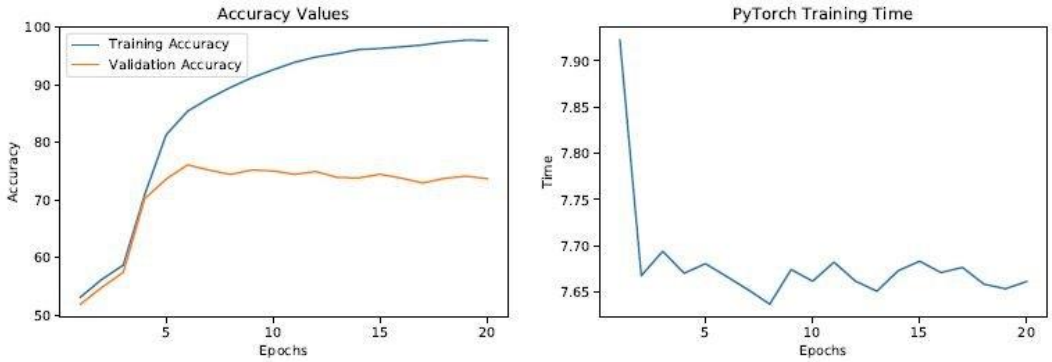


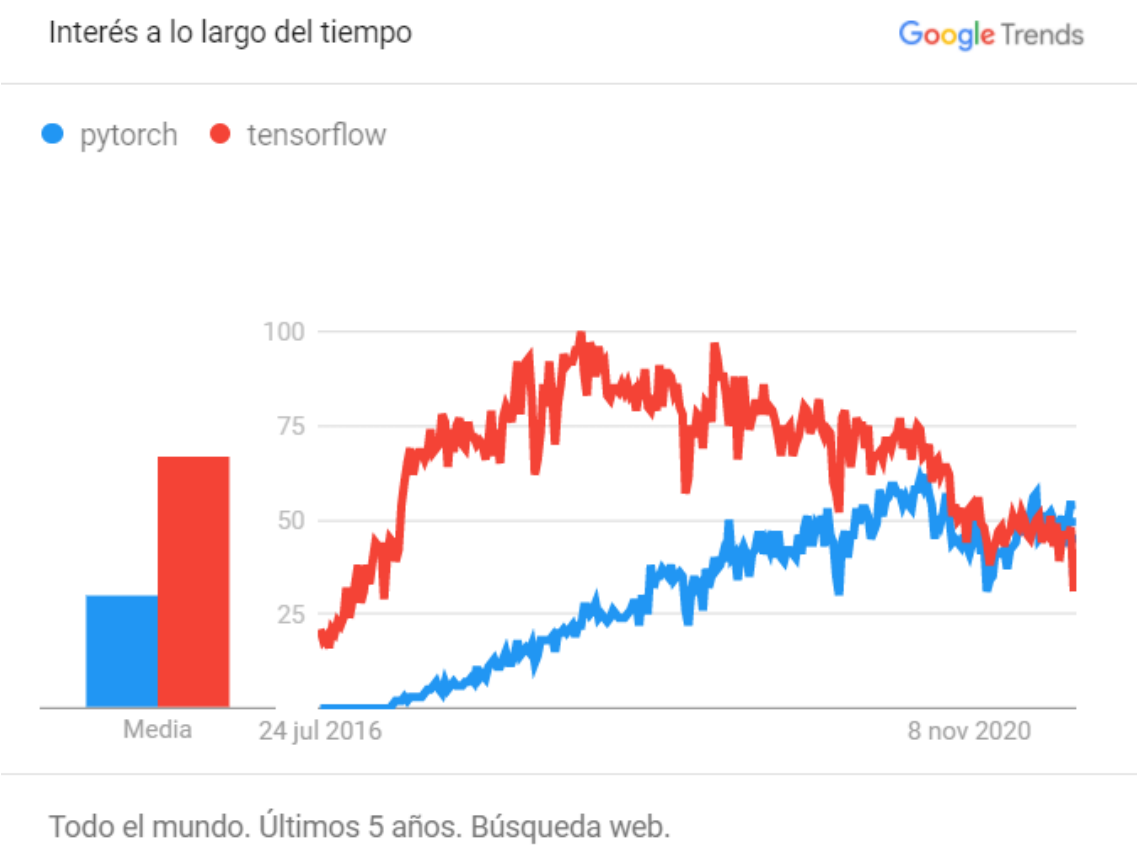
Figure 2: PyTorch Accuracy and Training Time

Gráficas de Precisión y Tiempo de entrenamiento para comparar TensorFlow (superiores) y PyTorch (inferiores)

Donde se pudo observar que para un mismo modelo con tres capas de 32x32x1 nodos, entrenado con el mismo dataset en ambas librerías, los resultados fueron que el tiempo de entrenamiento en TensorFlow fue sustancialmente mayor (de 11,2 segundos de media respecto a los 7,7 segundos de media en PyTorch), sin embargo, fue mucho menor el uso de memoria de TensorFlow durante el entrenamiento de 1.7GB de RAM a 3.5GB de RAM en PyTorch.

En cuanto a la precisión, ambas librerías mostraron ser muy similares en los experimentos.
PyTorch parecía tener mejor prototipado, pero TensorFlow podría ser una mejor opción para utilizar características personalizadas en la red neuronal.

Por todo lo dicho anteriormente, no es una tarea sencilla decantarse por PyTorch o TensorFlow, ya que, hasta la fecha actual, ambas librerías han ido mejorando y copiando características cuya falta las hacía relativamente inferior en algunos aspectos respecto a la otra librería, lo que, en consecuencia, ha conllevado que a día de hoy sean muy similares [11].
Y esto es algo que podemos comprobar en el uso y popularidad de estas.



Búsquedas en Google sobre las librerías PyTorch y TensorFlow

Por esta razón LeagueAI tiene dos versiones, una inicial en la que se utiliza la librería de TensorFlow desarrollada en 2017, y la otra, actual, que utiliza PyTorch.
Sin embargo, hay una razón muy clara por la cual la versión actual utiliza PyTorch y en posteriores versiones se seguiría utilizando.

Como se ha comentado, la versión actual utiliza YOLO detector, ésta se implementó en 2019, cuando PyTorch tenía ciertas ventajas como el uso de gráficos dinámicos para evaluar las operaciones en tiempo de ejecución. Hasta avanzado 2019 no se lanzó TensorFlow 2.0 que solventaba gran parte de estos problemas.
Por lo que se decidió trasladar el proyecto de TensorFlow1.0 a PyTorch y hacer uso de YOLOv3 que contaba con una gran documentación sobre su integración junto a PyTorch.

Además, en este proyecto, se han realizado correcciones y mejoras en base al código escrito con PyTorch y una de estas mejoras ha sido la integración de la nueva tecnología del detector YOLO: YOLOv5, el nuevo modelo que fue implementado en 2020 nativamente en PyTorch.
Mas adelante se detallará el funcionamiento y uso de este modelo, pero es importante saber que ésta es la razón por la cual se continúa usando PyTorch en este proyecto, ya que al estar implementado con la misma librería significa una mejor compatibilidad y sencillez de uso.

Primera Aproximación

1.Adaptación del código

Como se ha comentado antes, la primera versión de LeagueAI fue implementada en TensorFlow en 2017, por lo tanto, el primer paso consistía en adaptar el código de TensorFlow 1.0 a TensorFlow 2.0.

Para ello hay dos formas de hacerlo. La primera y más sencilla es deshabilitando el comportamiento de TensorFlow 2.0

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

La desventaja de esto se encuentra en que no se aprovechan los beneficios de TensorFlow 2.0 que, como su nombre indica, es una versión mejorada y optimizada de TensorFlow 1.0.

La otra forma de migrar el código entre estas dos versiones es seguir ciertos patrones de modificación como, por ejemplo, reemplazar las llamadas a 'Session.run', utilizar objetos para realizar el seguimiento de las variables o eliminar los símbolos 'tf.compat.v1' [12].

Pero ya que la mejora del proyecto no consistía en la versión de Tensorflow, se decidió optar por la 1ª forma de migrar el código usando 'compat.v1' ya que sólo se necesitaron modificar un par de líneas de código respecto a este punto que no iban a producir cambios sustanciales.

2.Entrenamiento y Conjunto de Datos

Antes de entrar en el funcionamiento, es necesario conocer los objetivos, que para este primer proyecto eran diferentes a los de la siguiente versión en PyTorch.

La meta entonces para esta primera versión consistía en crear un bot que tuviese una buena toma de decisiones para que pudiese exitosamente jugar una partida en solitario y, a largo plazo, que fuese capaz de tomar macro decisiones (para partidas con 10 jugadores). Es por ello que el foco no está orientado en la detección de objetos, que, como veremos a continuación resulta poco eficaz.

El primer paso consiste en la detección de objetos en el entorno, donde se usó OpenCV para el manejo de las imágenes y el API de detección de objetos de TensorFlow para las detecciones. Para que éste último funcione, se creó un dataset con capturas manuales del estado del juego etiquetadas con la herramienta LabelImg (herramienta cuyo uso consiste en remarcar objetos manualmente en las imágenes para distinguir las clases en su posterior entrenamiento [13]).

El siguiente paso consiste en entrenar, el cual es sencillo con el API de TensorFlow una vez que se ha creado el dataset con el siguiente formato:

```
<object>
        <name>Vayne</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
                <xmin>609</xmin>
                <ymin>638</ymin>
                <xmax>726</xmax>
                <ymax>800</ymax>
        </bndbox>
</object>
```

Donde la etiqueta *<bndbox>* corresponde a las coordenadas del cuadrado o 'caja' donde se encontraría el objeto.

3.Ejecución y Detecciones

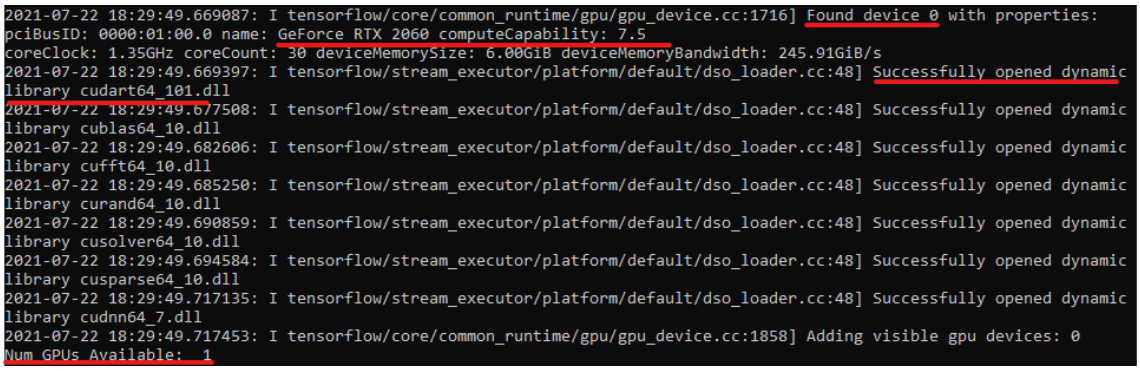
El dataset usado contiene 481 imágenes donde se entrenaron tres clases:

-Vayne (Jugador)

-Torre enemiga

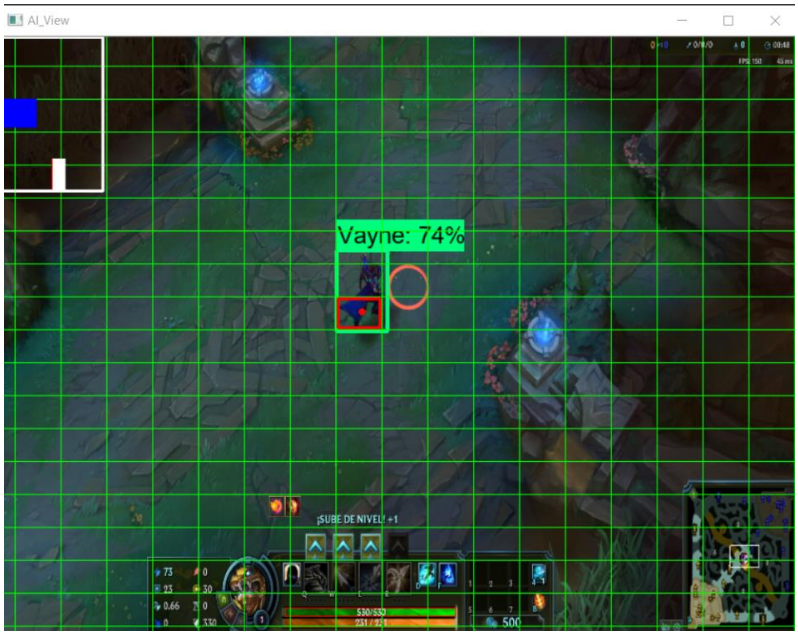
-Súbdito enemigo

Una vez entrenado, hay que asegurarse de que la ejecución del agente se realiza con el uso de la GPU mediante CUDA



Entonces al disponer de todos los elementos, el paso previo a la toma de decisiones es la detección.

Ya que es complicado definir el estado y posiciones a través de pixeles individuales, una solución inicial fue la de discretizar el entorno también porque no se pueden hacer clics de ratón sobre pixeles individuales en el juego y los elementos suelen ocupar un tamaño de pixeles variable, así que la discretización se realiza dividiendo la pantalla en una malla donde cada cuadrilátero tiene un tamaño relativo al del jugador detectado, produciendo así que la precisión de los clics sea bastante acertada.



Así se logra dividir en unidades la pantalla para poder determinar posteriormente la posición de nuevos elementos de la partida como súbditos o Torres.

El rectángulo rojo marca la casilla del elemento, en este caso el jugador, cuya posición es determinada por el punto rojo. Esto es así ya que en el videojuego la zona inferior del personaje es la más complicada de solapar visualmente.
En caso de súbditos enemigos la casilla sería azul y blanca para las torres enemigas.
Teniendo dividida de esta forma la pantalla, el sistema de clicado del ratón es sencillo, se utiliza el clic derecho tanto para mover como para atacar. Si no hay un enemigo en esa casilla donde apunta el ratón, el jugador se movería. En caso contrario, de haber un súbdito enemigo o torre enemiga, se atacaría a ese elemento.

4.Toma de Decisiones

La toma de decisiones es un proceso estocástico de 4 acciones:

-Atacar al súbdito enemigo

-Atacar a la torre enemiga

-Acercarse a la base enemiga

-Retirarse

Cada acción tiene una probabilidad calculada entre 0 y 1 y estas se normalizan para elegir una aleatoriamente.

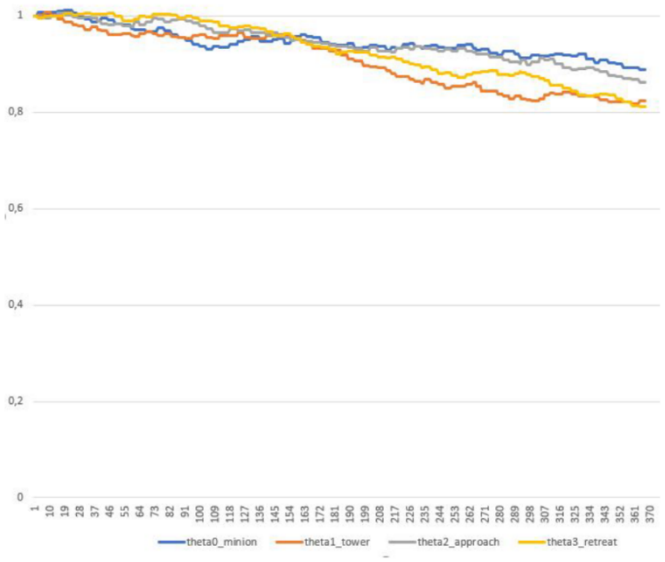
$$P_{[\text{action}]} = \frac{\text{Prob}_{[\text{action}]}}{\sum \text{Prob}_{[\text{all action}]}}$$

Se utiliza un proceso de mejora de aprendizaje de ascenso por gradiente, cada 5 acciones (para evitar fluctuaciones aleatorias) se actualiza el gradiente moviéndolo positivamente una pequeña cantidad 'δ'.

- La probabilidad de atacar a un súbdito enemigo es directamente proporcional a la distancia al súbdito y la vida del jugador.
- La probabilidad de atacar a una torre es mayor cuanto mayor sea el porcentaje de vida del jugador. Sin embargo, esta probabilidad se ve comprometida por el hecho de que en el juego siempre se deben realizar estos ataques junto a súbditos aliados, los cuales no se detectan en esta implementación.
- La probabilidad de avanzar hacia la base enemiga es constante mientras no se detectan enemigos, ya que así se evita que el jugador permanezca inmóvil. Ésta se reduce prácticamente a 0 cuando se detectan enemigos.

- La probabilidad de retirarse también fluctúa en base a la cantidad de vida del jugador; con poca cantidad de vida la decisión tomada debería ser retroceder en vez de atacar o avanzar hacia el enemigo.

La recompensa para el aprendizaje se ha aproximado en base a los movimientos posibles a realizar por simplicidad.



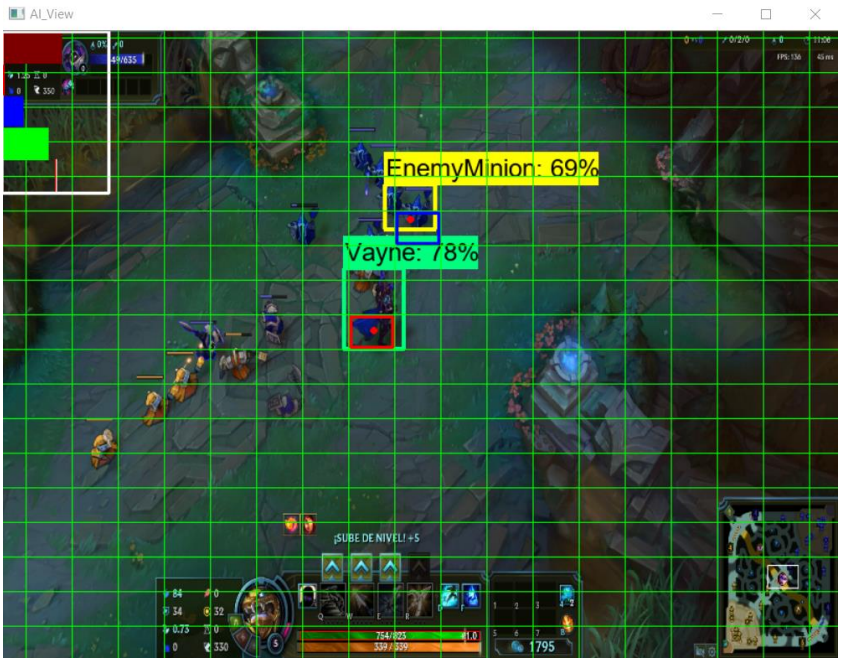
En la gráfica se puede observar el aprendizaje para las 4 posibles acciones, donde se puede comprobar que todas son decrecientes ya que el hecho de mantener el máximo porcentaje de vida es una recompensa que cualquier acción evita. Sin embargo, la acciones de atacar a súbditos reciben mayor recompensa, lo que hace que se reduzca menos. Siendo lo contrario la acción de atacar a la torre enemiga, que tal y como se ha comentado antes, faltaría más información en la detección para que se pueda tomar correctamente esta acción.

5.Resultados de la detección

A pesar de todo lo anterior, los resultados obtenidos en cuanto a la eficacia y eficiencia en la detección son relativamente aceptables, teniendo también en cuenta que el dataset no es muy grande debido al coste temporal que supone la obtención de las imágenes, su etiquetado y revisión para evitar errores.

Figura 4.5.1: Captura de pantalla original del videojuego LoL

Figura 4.5.2: Captura de pantalla de la 'visión' del Agente



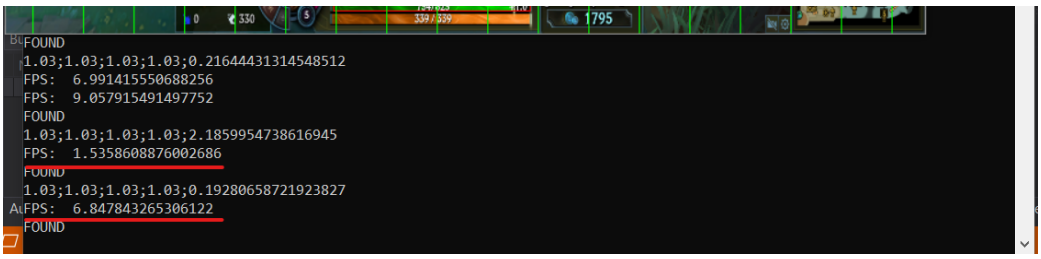


Figura 4.5.3: Terminal correspondiente a la Figura 4.5.2

En la figura 4.5.1 se puede observar lo que muestra el ordenador y vería un jugador real, hay 8 súbditos, el personaje Vayne, y parcialmente una torre enemiga.
En la Figura 4.5.2 se observa lo que ve el Agente correspondiente al código documentado en esta sección, se observa que sólo detecta al jugador y a un súbdito, y en la Figura 4.5.3 la última marca de FPS, corresponde a los fotogramas por segundo para la detección y toma de decisiones de la Figura 4.5.2, lo que supone una buena tasa de refresco de 6.84 por segundo. No obstante, en la detección anterior, los súbditos detectados fueron 5, lo que hacía un total de 6 objetos detectados y redujo los FPS a menos de 2, como se puede observar en la penúltima línea de FPS de la figura 4.5.3.

Y es por esta variación tan drástica en la tasa de refresco cuando la detección de objetos y la toma de decisiones es más compleja, que se vuelve necesaria la mejora en ambos aspectos.

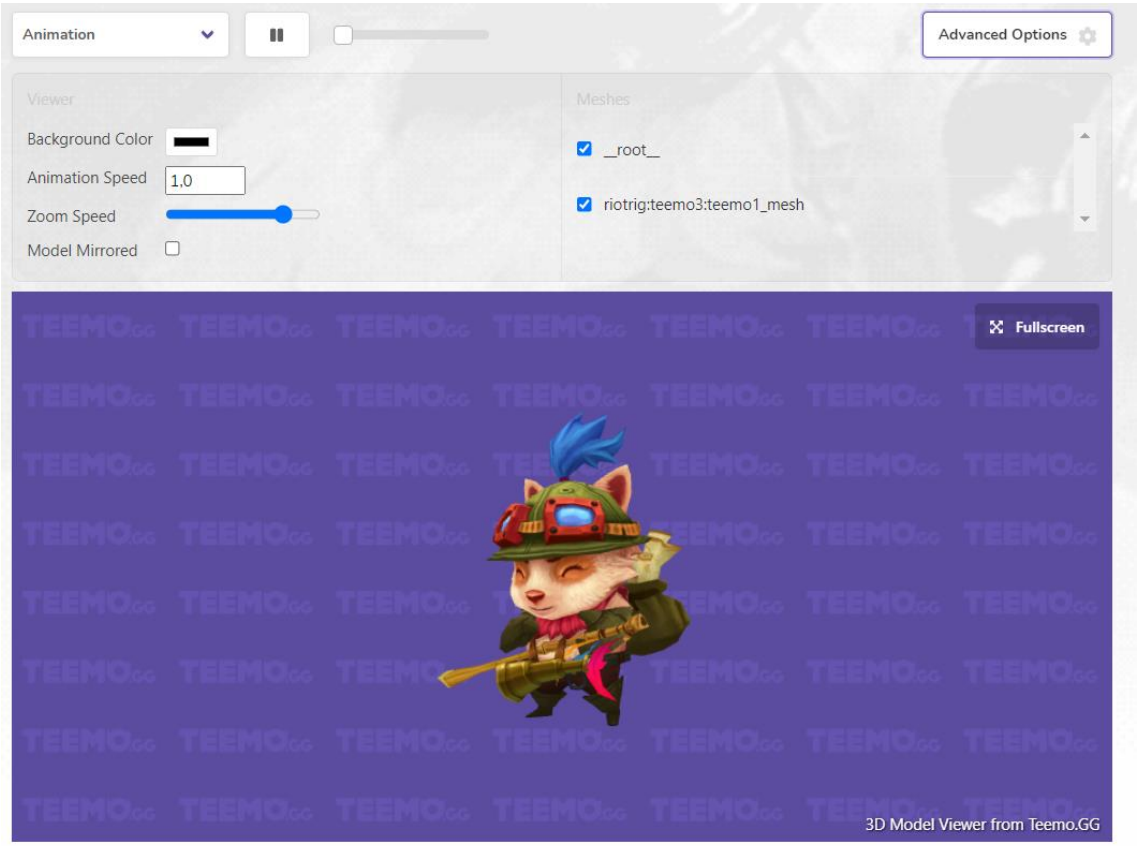
En los siguientes apartados se detallarán unas tecnologías y mejoras realizadas que solventarán los problemas de generación y tamaño del dataset, así como efectividad y eficiencia a la hora de detectar objetos.

Generación del Conjunto de Datos

Teniendo una idea más detallada de cómo funciona o, debería funcionar el proyecto, se puede observar que el proceso más costoso es la creación del conjunto de datos (o dataset). Obtener cientos o miles de imágenes y editarlas manualmente para su posterior entrenamiento supone un consumo de tiempo realmente grande, y más aun teniendo en cuenta el hecho de que si se quiere aumentar el número de objetos a detectar, mejorar la calidad de las imágenes o se realiza un cambio visual de algún aspecto en el videojuego, todo el dataset puede dejar de ser útil.

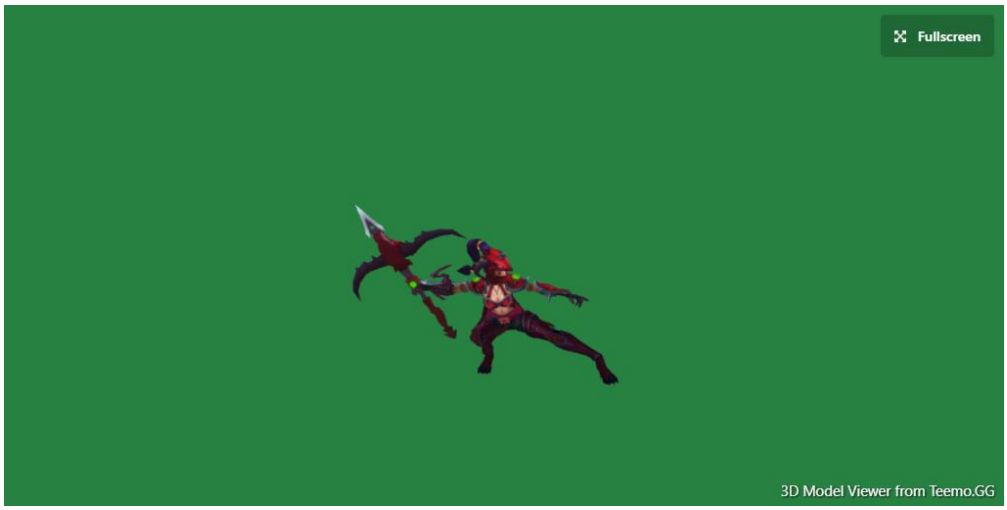
Este es uno de los principales problemas solventados en la versión implementada en PyTorch.

Para ello se hace uso de una herramienta online que permite el visualizado de los modelos en 3 Dimensiones de los personajes de League of Legends, teemo.gg.



Esta herramienta permite seleccionar casi cualquier objeto animado del juego (más de 140 campeones, súbditos enemigos o aliados, monstruos neutrales de la jungla, etc.), rotar y modificar el tamaño de visualización del modelo, elegir animación, velocidad de animación y color de fondo.

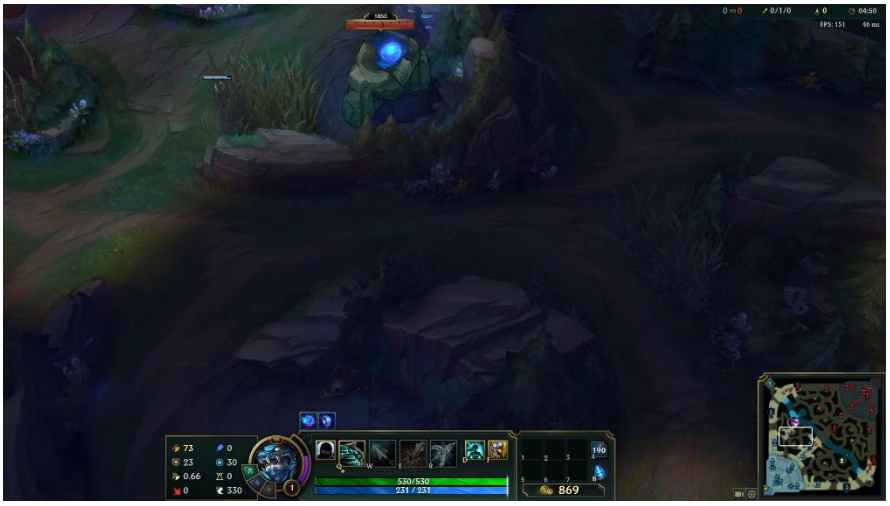
Por lo tanto, es una tarea sencilla seleccionar los objetos sobre los que se desee aprender la detección y grabar un vídeo no muy extenso variando las animaciones que interesen.



Captura del personaje Vayne con el traje (o 'skin') Matadragones chroma rojo realizando animación de baile sobre un fondo de color verde

Y, para continuar la preparación del dataset, aparte de esos vídeos, es necesario sacar manualmente:

-Capturas vacías del mapa (Algunas con interfaz y/o niebla de guerra)

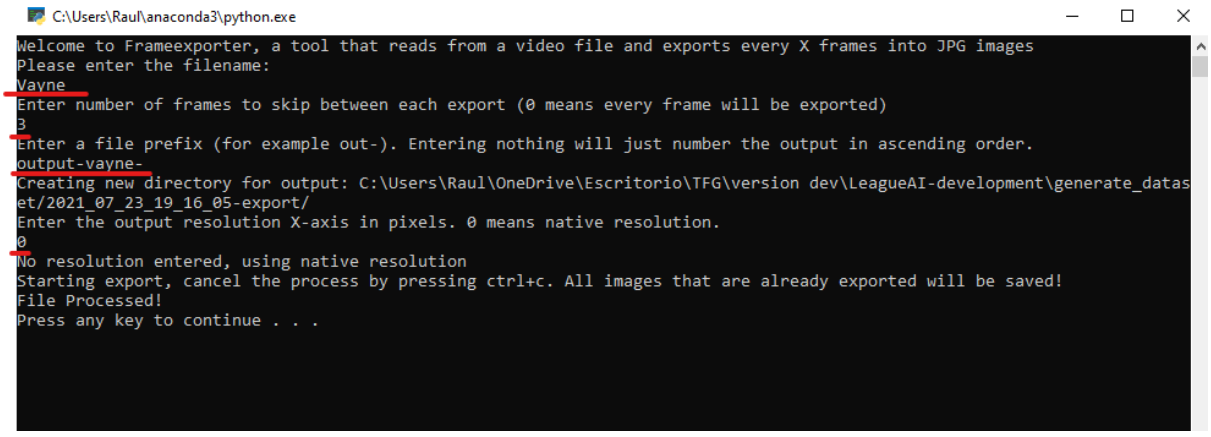


-Capturas de las torres sin fondo (enemigas y aliadas)

-Cursores del juego[1] (para que el detector se acostumbre a encontrarlos en la imagen)

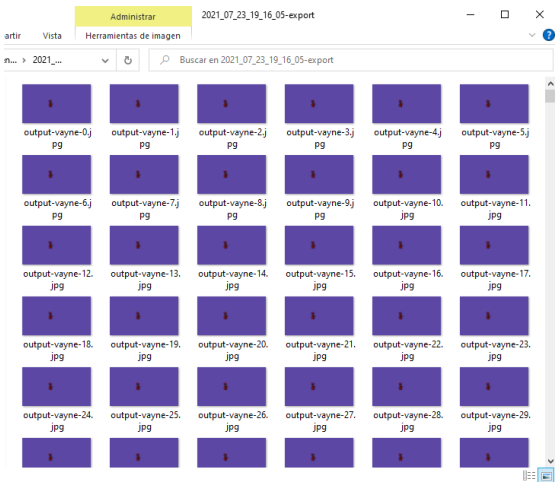
Una vez recopilados todos estos datos ya se puede proceder a la generación del dataset en tres pasos dentro de la carpeta 'generate dataset' del proyecto:

I. El primer paso es utilizar el programa pyFrameexporter.py. Especificando nombre del archivo de vídeo (formato '.mp4' o '.mkv' funcionan bien), número de fotogramas entre capturas (cada 3 suele ser buena cantidad), nombre de archivos de salida y resolución, se exportarán a imágenes '.jpg' capturas consecutivas del vídeo

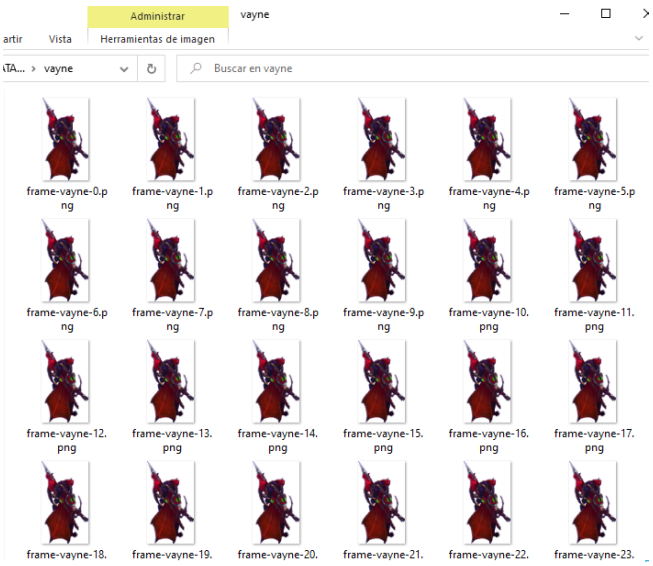


Indicaciones para la ejecución del aislamiento de frames

[1] Nota: Estos dos cursores (movimiento y ataque) pertenecen a los cursores de legado, de la antigua versión de League of Legends. Encontrar las imágenes aisladas de las versiones del cursor moderno es más complicado, por tanto, una solución sencilla es usar los cursores antiguos y activarlos dentro de las opciones del videojuego.



II. El segundo paso consiste en utilizar esas imágenes generadas para recortarles el fondo y obtener las figuras aisladas. Por eso es importante determinar antes el color de fondo usado en la herramienta de visión de modelos 3D de teemo.gg. En este caso el color del fondo tiene que ser RGB (95, 80,170) para eliminar solo los pixeles detectados en ese color. De esta tarea se encarga el script 'pyExportTransparentPNG.py', que devuelve las imágenes con el formato '.jpg'



III. El último paso consiste en la generación del dataset como tal, para ello se usa el script 'boostrap.py' al que se le especifican dentro del código todas las rutas de las imágenes generadas anteriores, mapas, torres y cursores. También se determina el tamaño del conjunto (2000 o 3000 imágenes, por ejemplo), y diversos factores con relativa importancia:

-Números máximo y mínimo de campeones, súbditos, torres y cursores que pueden aparecer en una misma imagen

```
characters_min = 0
characters_max = 4
```

```
minions_min = 3
minions_max = 10
order_minions_min = 3
order_minions_max = 10
```

```
towers_min = 0
towers_max = 1
order_towers_min = 0
order_towers_max = 1
```

```
cursors_min = 0
cursors_max = 5
```

-Rotación

```
# Random rotation maximum offset in counter-/clockwise direction
rotate = 10
```

-Escalado

```
scale_champions = 0.7
random_scale_champions = 0.1
scale_minions = 0.6
random_scale_minions = 0.25
scale_towers = 1.0
random_scale_towers = 0.2
```

-Factor de agrupación entre objetos (cuanto mayor, más grande será la distancia)

```
bias_strength = 220
```

Y también otros factores que no son tan decisivos pero podrían ayudar a mejorar la precisión en el entrenamiento como el ruido o la difuminación (para simular la posición dentro de un arbusto en el juego).

Para que así el programa se encargue de colocar aleatoriamente en capturas del mapa también elegidas aleatoriamente los elementos aislados anteriormente y modificados con los parámetros especificados y, por tanto, se formen imágenes de manera procedural.

El resultado de todo esto es la generación de un dataset del tamaño y propiedades indicadas, con el etiquetado requerido para el detector YOLO en apenas unas horas (entre 10 y 12 con la máquina utilizada para las pruebas) dependiendo de la capacidad del computador donde se realice.

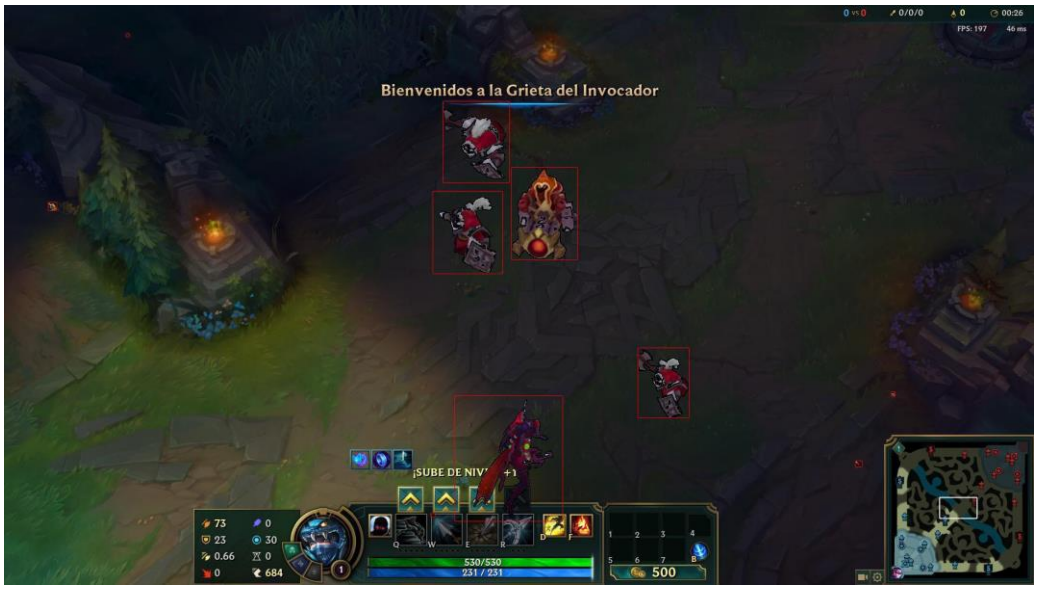


Imagen generada automáticamente (sintética)

141.txt: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
4 0.4895833333333333 0.7814814814814814 0.10520833333333333 0.21666666666666667
3 0.45 0.39166666666666666 0.06822916666666666 0.1425925925925926
1 0.5244791666666667 0.3592592592592593 0.06458333333333334 0.16018518518518518
3 0.4583333333333333 0.2361111111111111 0.06458333333333334 0.14074074074074075
3 0.6401041666666667 0.6509259259259259 0.05 0.12037037037037036
```

Etiquetado de la imagen anterior en formato YOLO

DETECTOR YOLO

Teniendo el conjunto de datos, como ya se ha hablado antes, es necesario utilizar algún tipo de Sistema para poder entrenar y que, además, proporcione una detección efectiva y eficiente.

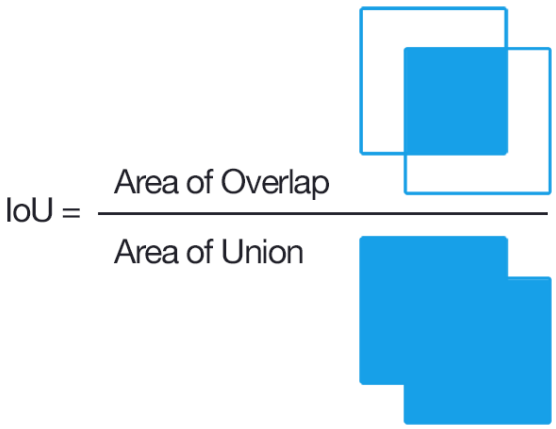
Para lo cual, se utilizará el detector YOLO, una estructura de red neuronal convolucional desarrollada desde 2015.

En un primer momento se utilizó YOLOv3, que fue la última versión de YOLO implementada por el autor original y en la que están basadas las 2 versiones posteriores.

Es importante conocer primero tres algoritmos o técnicas utilizados por YOLO:

- IoU (Intersection Over Union): Es una métrica que permite evaluar la precisión del detector en un conjunto de datos. Para esto es necesario saber cuál es la bounding box[2] real (ground-truth) y la predicha (predicted) por el algoritmo.

  Sabiendo estas dos bounding boxes, IoU evalúa la precisión con un sencillo ratio entre el área de solapamiento o superposición y el área de unión de ambas [14].



[2] Literalmente traducido como cuadro delimitador, corresponde a la 'caja' donde se encapsula visualmente un objeto en la imagen en base a las coordenadas ('x' e 'y' para marcar el centro del objeto, y 'w' y 'h' para marcar la anchura y altura del objeto)

- NMS (Non-Maximum Supression): Este algoritmo permite mejorar la lista de propuestas de bounding boxes devueltas por un detector. Después de recibir la lista de propuestas de bounding boxes, se escoge la que mejor puntuación tenga y se calcula el IoU de esta sobre el resto de propuestas, eliminando las que tengan un IoU que sobrepase un threshold (o límite) establecido anteriormente, ya que un alto IoU en este caso significaría que esa bounding box estuviese marcando el mismo objeto (menos eficientemente). Después de eliminar las bounding boxes sobrantes (si las hubiese) se eliminaría de la lista de propuestas la bounding box inicial con la mejor puntuación y se añadiría a la lista de bounding boxes final, repitiendo el proceso sobre la lista de propuestas restante [15].

Resultado de aplicar NMS (a la derecha)

- Bounding Box Regression: Esta técnica tiene el objetivo de aplicar regresión para determinar la relación entre las cuatro coordenadas (x,y,w,h) de una bounding box propuesta y las mismas coordenadas de la bounding box real, que se detallará a continuación ya que tiene una gran importancia para hacer las detecciones.

1.YOLOv3

[16]Esta versión de YOLO es una red conectada formada por 75 capas convolucionales, lo que también supone que las imágenes de entrada tengan un tamaño fijo, el cual se reduce en base al tamaño del stride (zancada) que divide la imagen a número de neuronas (o celdas para que sea más intuitivo de entender), así, por ejemplo, una imagen de 416 x 416 tendría como salida 13 x 13 para un stride de 32.

Esta salida es el mapa de características donde la celda del centro de la bounding box real es la encargada de detectar el objeto. Esta celda tiene tres anclas (aproximaciones sobre las dimensiones de la bounding box a predecir), y estas se usan generando en cada ancla una bounding box sobre las que se aplica el ancla para elegir finalmente la que tenga el mayor IoU sobre la bounding box real siguiendo la siguiente fórmula para la creación de la bounding box (Bounding Box Regression).

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$

Donde las coordenadas de la bounding box a predecir (b_x y b_y) se obtienen con la suma de la función sigmoidea aplicada a las coordenadas de la bounding box real (t_x y t_y) más las coordenadas de la celda (c_x y c_y). Y las dimensiones de anchura y altura (b_w y b_h) se obtienen aplicando una predicción (p_w y p_h) y normalizándolas posteriormente con las dimensiones de la imagen.

Aunque sean las 75 capas convolucionales de YOLOv3 las que se encargan de aprender y detectar objetos mediante convoluciones, la red en realidad tiene 106 capas que son las que construyen el archivo de configuración (.cfg) para determinar el procesado de las imágenes a partir de unos pesos, conformando así 5 tipos adicionales de capas al margen de la convolucional:

I. Shortcut

Es una capa de salto que obtiene la salida de añadir características del mapa de la capa anterior y de la capa indicada en la variable 'from' (En este caso, de 3 capas anteriores a la actual).

```
[shortcut]
from=-3
activation=linear
```

II. Upsample

Realiza un muestreo ascendente bilineal, modificando el mapa de características de la capa anterior por el factor de 'stride'.

```
[upsample]
stride=2
```

III. Route

Las Capas Route devuelven la concatenación del mapa de características de las 2 capas indicadas en 'layers' (sólo devuelven la de una capa en el caso de que solo tenga un valor la variable 'layers')

```
[route]
layers = -4

[route]
layers = -1, 61
```

IV. YOLO

Es la capa de detección la cual contiene el número de clases a detectar y las anclas explicadas anteriormente, de las cuales se cogen las indicadas en 'mask' (en este caso las 3 primeras hasta la tupla '33,23', ya que cada celda predice 3 cajas de detección).

```
[yolo]
mask = 0,1,2
anchors = 10,13,  16,30,  33,23,  30,61,  62,45,  59,11
classes=80
num=9
jitter=.3
ignore_thresh = .5
truth_thresh = 1
random=1
```

V. NET

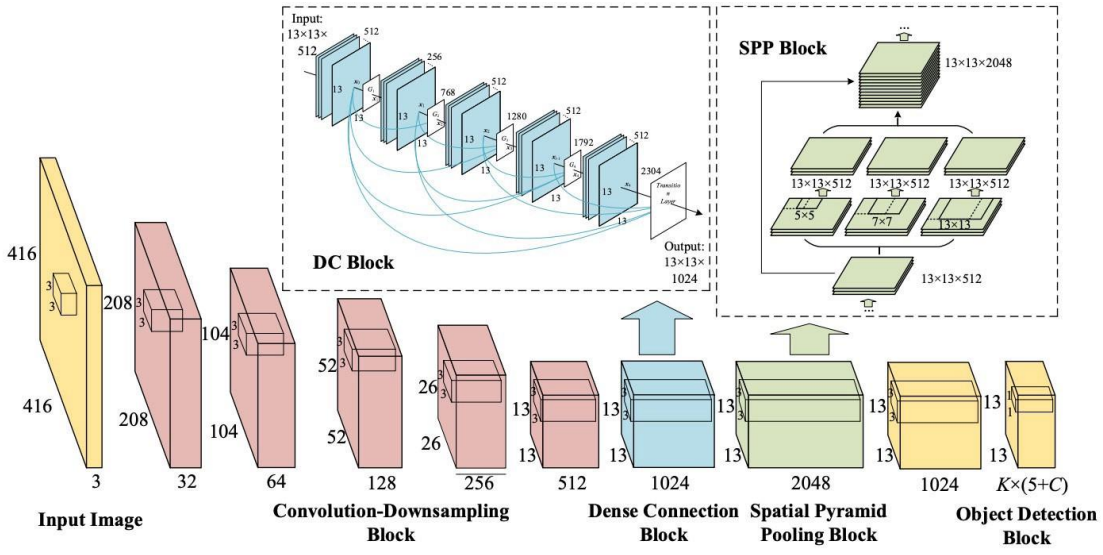
En realidad, el funcionamiento de esta capa es distinta; se encuentra la primera de todas y sirve para describir la información sobre la red neuronal (como las características de la entrada de la red)

```
[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=16
width= 320
height = 320
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
```

Utilizando estas 106 capas, YOLOv3 realiza detecciones en 3 diferentes escalas en base a strides de tamaño 32, 16 y 8. Por lo tanto si tiene que predecir bounding boxes para cada celda esto crearía un número muy grande de predicciones, las cuales se encarga de reducir en base a un thresholding y la utilización del Non-maximum Supression comentada previamente.

2.YOLOv4

En 2020, dos años después del lanzamiento de YOLOv3 y después de que el autor original decidiese no continuar con el proyecto, fue lanzado YOLOv4, una mejora del algoritmo anterior que proporcionaba un aumento de la precisión media de hasta un 10% y una mejora del número de fotogramas por segundo de un 12%. Donde aparte de añadir distintas técnicas como el Bag of Freebies (para aumentar variaciones en las imágenes durante el entrenamiento a coste de mayor tiempo de duración de este) o el Bag of Specials (que mezcla diferentes métodos tales como la función de activación Mish, que pueden producir mejoras drásticas en la detección a cambio de una ligero retraso de tiempo), el principal cambio fue en la columna vertebral usada: CSPDarknet53 en vez de Darknet53 de YOLOv3, el cual particiona el mapa de características de la capa base en 2 partes, una que pasa por las capas convolucionales y otra que podría no pasar por estas para mezclarlas al final y obtener una estrategia que proporciona mayores flujos del gradiente.



Flujo de procesamiento de YOLOv4 dada una imagen de entrada (416x416)

3.YOLOv5

Apenas 2 meses después del lanzamiento de Yolov4, Glenn Jocher, autor distinto al de los YOLO anteriores, sorprendió con el lanzamiento de YOLOv5. Es más complicado obtener información detallada de las tecnologías usadas en YOLOv5 ya que el autor no ha proporcionado un informe en detalle.
Aun así, esta versión también utiliza CSPDarknet53 como columna vertebral, también se sabe que en vez de usar la función de activación Mish, en la capa central utiliza la función de activación Leaky ReLU [17]

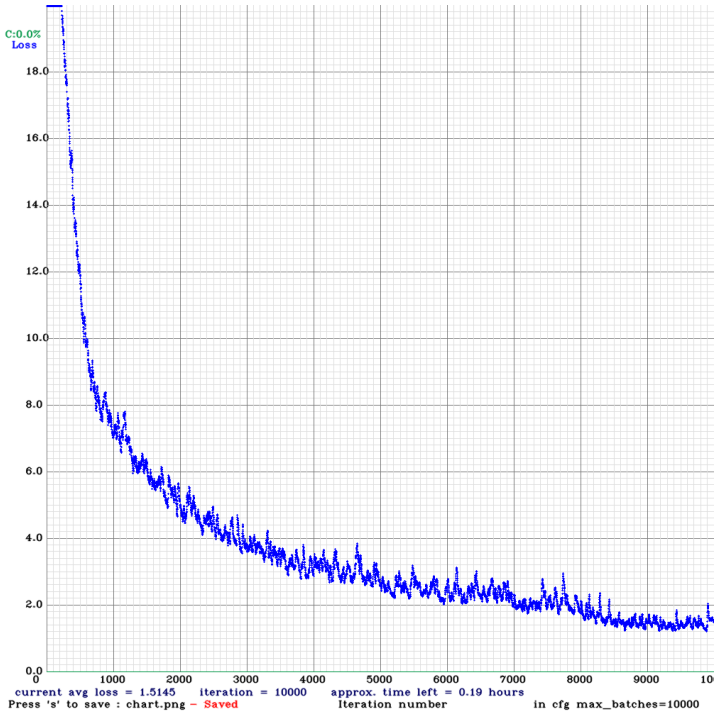
$$f(x)=max(0.01*x\ ,\ x).$$

Donde en vez de despreciar los valores negativos, se les proporciona un valor muy pequeño al multiplicar por una constante.
También utiliza función de activación Sigmoidea en la capa final.
Como función de entrenamiento, al igual que YOLOv4 utiliza Descenso por gradiente estocástico (SGD), sin embargo, para conjuntos de datos más pequeños YOLOv5 puede utilizar ADAM, que es un reemplazo de SGD (ya que ADAM tiene peor rendimiento general, aunque dependa de la situación).

4.Comparación y Resultados

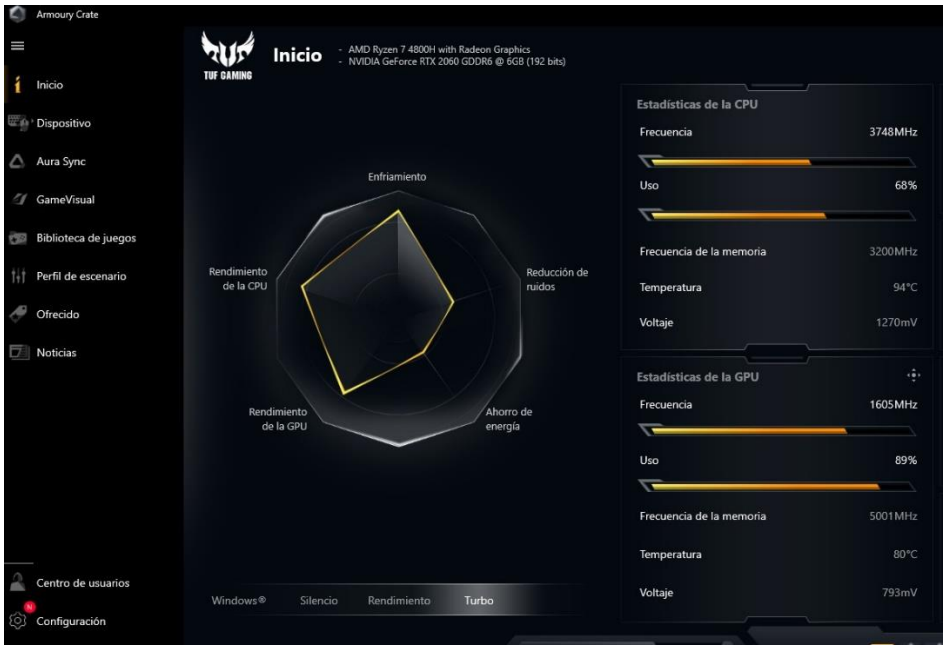
Ya que hay 3 versiones de YOLO, se van a mostrar comparaciones para decidir cuál será la versión final elegida para el proyecto.

El entrenamiento en YOLOv3 y YOLOv4 es muy similar, se utiliza el mismo formato de carpetas y archivos para el dataset, y para comenzar el entrenamiento desde el mismo proyecto sólo hay que hacer unas sencillas modificaciones en los archivos de configuración explicados en el documento oficial.



Entrenamiento de YOLOv3 con 3000 imágenes de dataset

Y observaremos la gráfica de aprendizaje que tiene el mismo estilo para ambas versiones.

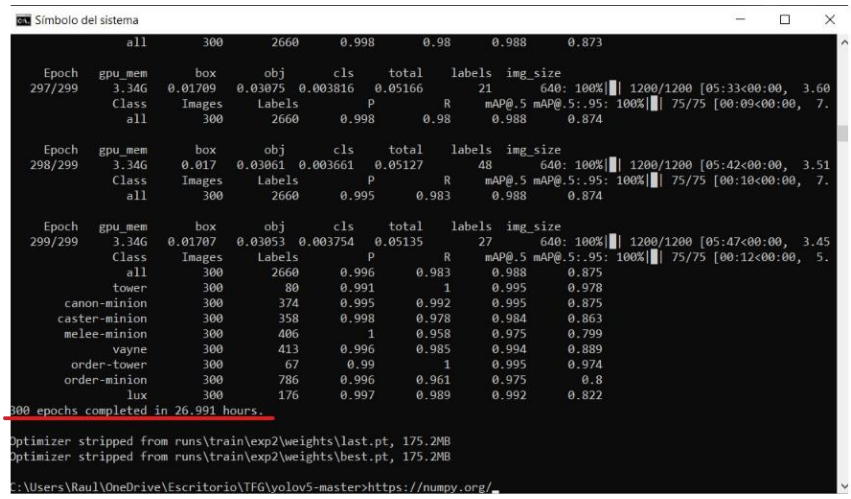


Características y rendimiento de la computadora Asus usada durante el entrenamiento

Para una computadora de las siguientes características (y 16GB RAM) funcionando a máximo rendimiento, el entrenamiento tarda entre 20-24 horas con un tamaño máximo de lote (batch) de 16.

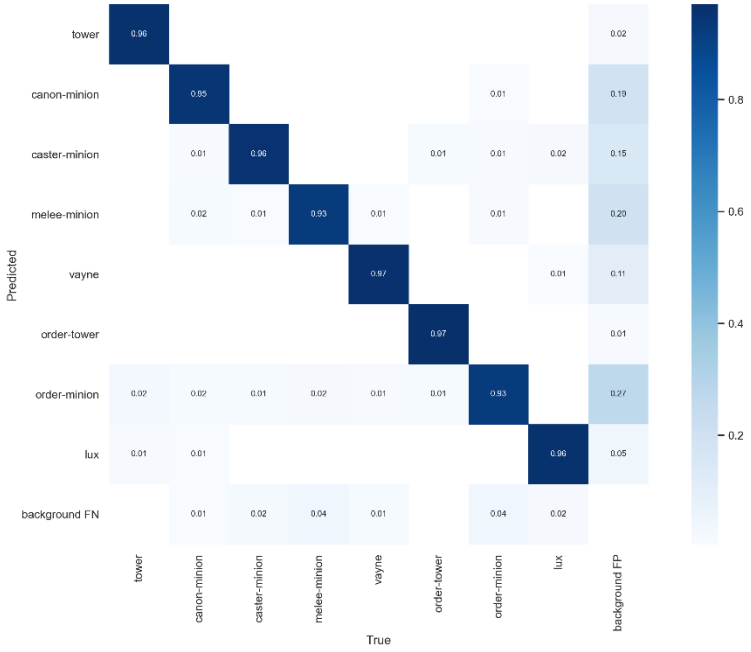
Por su parte YOLOv5 al estar en un código distinto tiene algunas variaciones más. El conjunto de datos y etiquetado es el mismo, sin embargo la referencia a los datos cambia en cuanto al formato y se debe hacer en fichero '.yaml'.

También YOLOv5 permite entrenar en 5 formatos diferentes en caso de que haya restricciones computacionales: pequeño (Small-s), mediano (Medium-m), grande (Large-l) y Extra grande (Extra Large-x). En primera instancia se decidió utilizar el Extra-Large que proporciona la mayor precisión



Terminal tras la ejecución de YOLOv5x

Donde el entrenamiento tuvo que hacerse con un tamaño de lote de 2 unidades y duró casi 27 horas, que se redujeron a más de 13 horas para entrenar en Large, algo más de 8.5 horas en Medium y poco más de 7 horas para entrenar en Small. YOLOv5 no proporciona gráficas dinámicas de mejora durante el entrenamiento, pero si distintos análisis tras terminar éste, como, por ejemplo, la matriz de confusión

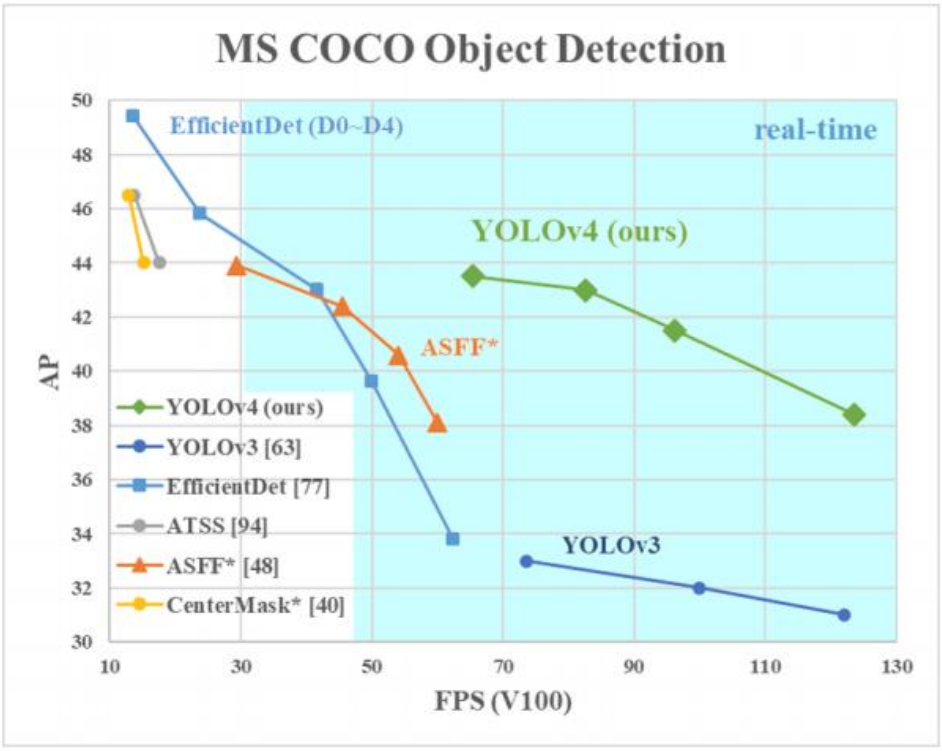


Matriz de confusión para las 8 clases entrenadas en YOLOv5x

Dónde se puede observar que el mayor problema de la detección se encuentra en detectar algunos elementos del fondo como objetos detectables y viceversa (como en el caso de los súbditos aliados u order-minions).

Por todo esto se observa que, en cuanto al entrenamiento para la detección, cualquiera de los 3 algoritmos de detección es viable, pudiendo ser preferible YOLOv5 en caso de realizarse con una máquina menos potente, también debido a que produjo menor tamaño de archivo de pesos: 175MB en Extra-Large, 95MB en Large, 42MB en Medium y menos de 15MB en Small, respecto a los 240MB y 250MB de YOLOv3 y YOLOv4 respectivamente.

Las comparaciones en eficiencia y efectividad post-entreno son más complicadas, ya que dependen de diversos factores, así como de la adaptación de los algoritmos al programa, por tanto, se hizo uso de estudios públicos [17] y breves pruebas con el código de LeagueAI para decantarse por una de estas 3 versiones de YOLO:

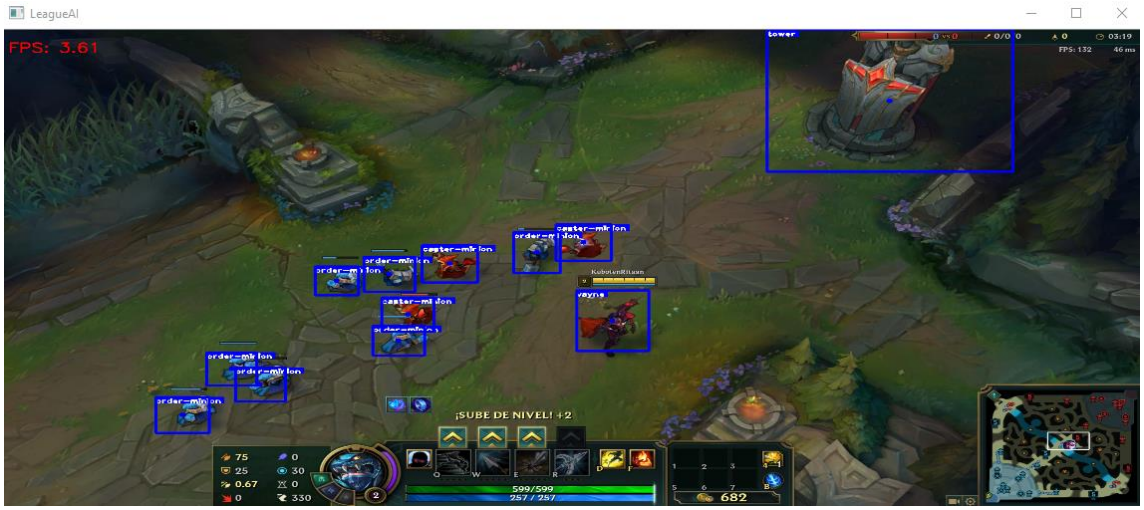


Resultados de FPS respecto a precisión media (AP) de YOLOv3 y YOLOv4 para el Dataset COCO

En primer lugar, YOLOv3 respecto a YOLOv4; como se observó previamente YOLOv4 aporta aproximadamente una mejora del 10% tanto en velocidad como precisión, y esto se pudo comprobar al entrenar ambos algoritmos con el conjunto de datos de COCO, donde se observó mayor número de detecciones para el mismo número de fotogramas por segundo.

En segundo lugar, se comparó directamente sobre ejecuciones de LeagueAI YOLOv3 respecto a YOLOv5 (versión Extra-Large).

YOLOv3 entrenado con 5 clases de objetos



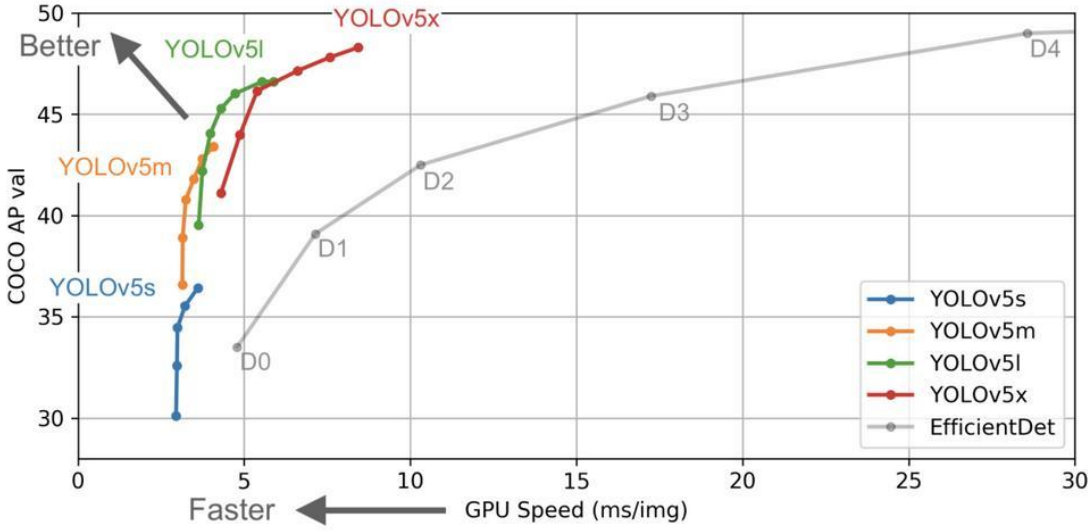
YOLOv5x entrenado con 8 clases de objetos

Donde se concluyó que YOLOv3 no era el detector óptimo a elegir, ya que YOLOv5x seguía proporcionando una ligera mejora de fotogramas cuando la carga de detecciones era muy grande (que prácticamente se duplicaba al reducirse el número de objetos), además de la eficiencia de estas (en la 1ª imagen correspondiente a YOLOv3 no detecta 3 súbditos enemigos, y en la 2ª, correspondiente a YOLOv5x, se detectan correctamente los 12 objetos visibles).

Por lo tanto, la última cuestión respecto al detector era si usar YOLOv4 o YOLOv5.

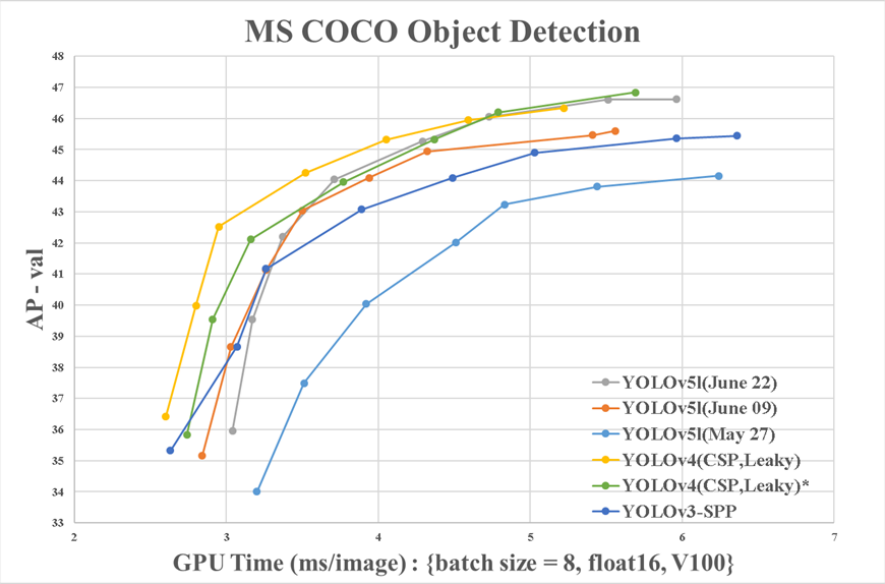
Para esto pueden ser de ayuda de nuevo estudios públicos [17].

Primero es conveniente echar un vistazo a las comparaciones entre las distintas versiones de YOLOv5



Comparaciones de las versiones s, m, l y x de YOLOv5 usando el dataset de COCO

Donde a mayor versión se observa mejor eficiencia a coste de mayor tiempo.



Comparación de las versiones 3, 4 y 5 de YOLO para el Dataset de COCO

El estudio reveló que YOLOv5l y YOLOv4 son muy similares, siendo YOLOv4 ligeramente más efectivo y eficiente, sin embargo, la desventaja en la precisión se puede resolver e incluso mejorar en la versión x de YOLOv5 y extrapolando los resultados de YOLOv4 a las capturas mostradas anteriormente de LeagueAI, para la misma imagen de YOLOv5x, YOLOv4 tendría una tasa mejorada de FPS de entre 0 y 1 aproximadamente, sin asegurar que la detección sea igual de correcta.

Se puede deducir que YOLOv4 podría proporcionar una muy ligera mejora sobre YOLOv5, sin embargo, teniendo en cuenta que esta mejora no es decisiva en la extracción de los datos, hay una razón que decanta la balanza a favor de YOLOv5, y consiste en que la sencillez, flexibilidad y compatibilidad de uso que proporciona YOLOv5 sobre un código en PyTorch hace que se reduzca en gran medida la complejidad del proyecto, al contrario que YOLOv4.

Para poder aplicar YOLOv4 sobre un Dataset personalizado es necesario conocer en profundidad los detalles de su implementación para que a partir de los pesos del entrenamiento se puedan aplicar correctamente los bloques del archivo de configuración para extraer las características aplicadas sobre los fotogramas en tiempo real y obtener el tensor con los resultados de la detección. En cambio, YOLOv5 permite cargar el modelo desde PyTorchHub,

```
# Model
model = torch.hub.load('ultralytics/yolov5', 'yolov5s')
```

con lo que sólo es necesario aplicar el modelo sobre la imagen para obtener los resultados de la detección.

```
# Inference
results = model(img)
```

Además, con la posibilidad de entrenar YOLOv5 en otros formatos, se dedujo que a partir del formato Medium, ya se obtienen resultados de detección considerables para su uso (prácticamente sin fallos en la detección de los objetos entrenados), y un aumento del rendimiento sustancial (entre 1 y 2.5 FPS incrementados de media respecto a YOLOv5x)

Resultados de YOLOv5m entrenado con 8 objetos

Todo esto lleva a la conclusión de que YOLOv5 sea la versión más recomendable del detector YOLO.

Implementación

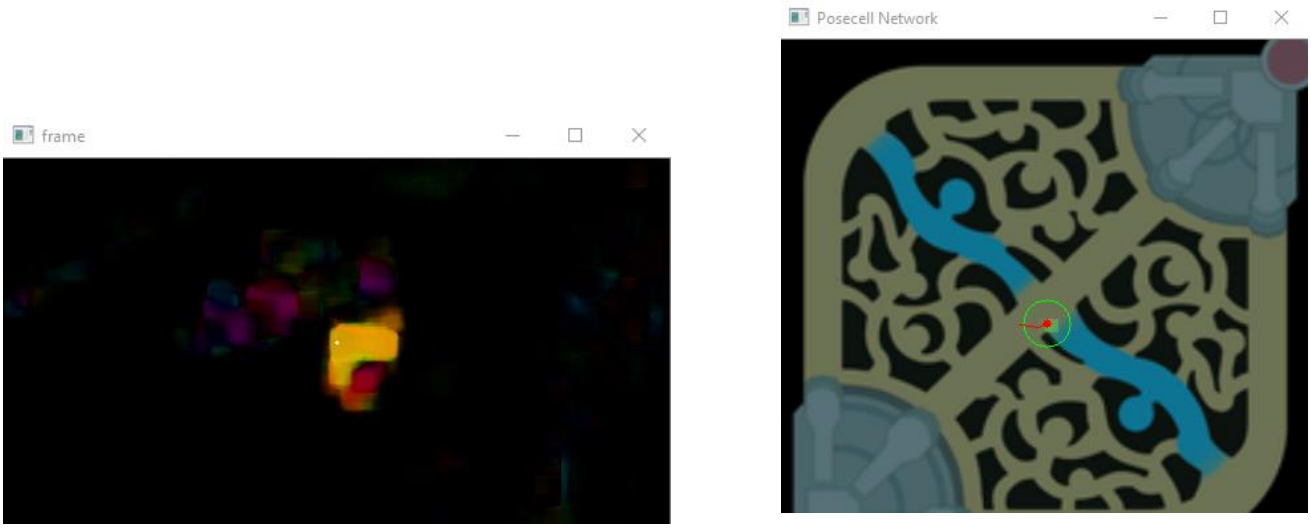
LeagueAI, aparte de todos los scripts comentados, cuenta con algunos auxiliares para, por ejemplo, evaluar la calidad de aprendizaje con dos datasets diferentes o comprobar la integridad de un dataset.

Pero sin lugar a dudas el núcleo del proyecto se encuentra en el archivo "LeagueAI_helper.py".
Este archivo contiene la definición de las clases de detección para manejar las coordenadas y tamaño de estas, y una clase 'input_output' para obtener la localización de donde se extraigan los fotogramas, y a su vez se obtengan los píxeles de ésta.
Pero la mayor complejidad que contiene este archivo se encuentra en la clase del Framework de LeagueAI donde:

- Se carga el modelo del detector (YOLOv5)
- Se recogen los resultados de pasarle los pixeles del fotograma al detector
- Se dibujan los resultados en la imagen
- Se devuelve la imagen con la información al programa que lo invoque

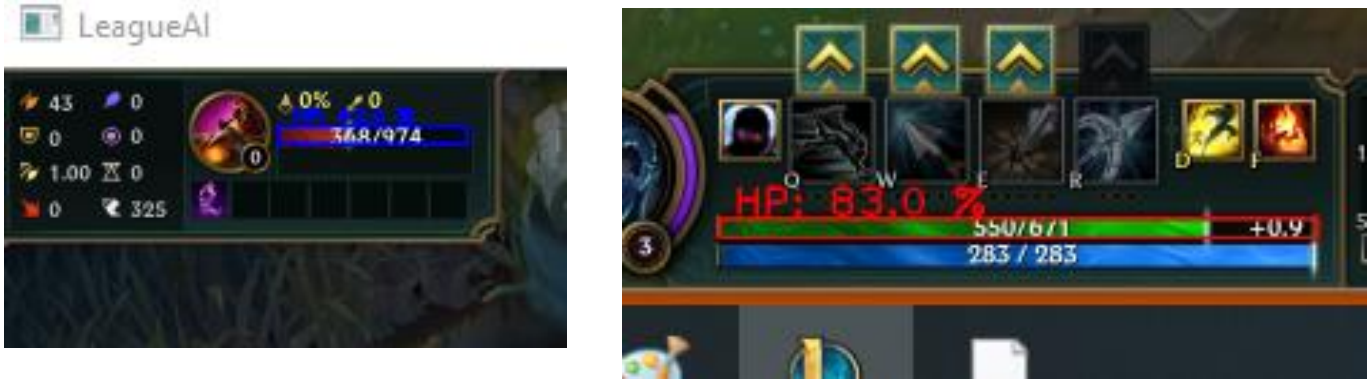
Con esto se puede llamar al script principal "LeagueAI_vayne.py" donde se utiliza esta información y se hace uso tanto de la carpeta "location" como de "agent_helper.py" para obtener datos y realizar cálculos adicionales:

-Odometría Visual para estimar movimientos en el mapa utilizando el método diferencial de Lucas-Kanade para estimar el flujo óptico



La desventaja es que este método asume como constante la velocidad de los píxeles vecinos [18], lo que supone un problema al no tener fotogramas constantes en la aplicación con la detección de objetos (a mayor número de objetos a detectar, menor tasa de FPS).

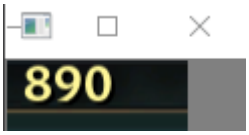
-Cálculo de los porcentajes de vida de enemigos y Jugador



Aprovechando la funcionalidad de analizar los píxeles de los fotogramas, simplemente controlando el color de las posiciones del fotograma correspondientes a las barras de vida se pueden obtener estos datos sin

necesidad de utilizar más detectores que puedan ralentizar el funcionamiento del programa (lo que se podría extrapolar para obtener más datos como escudos de personajes, maná, energía, etc.).

-Cálculo del oro disponible



Recortando la zona correcta del frame se puede obtener la cantidad de oro pasándole esta zona de la imagen a la herramienta para el reconocimiento de caracteres (OCR) Tesseract de Python.

La forma normal de saber cuándo se ha eliminado a un enemigo en el juego es ver la cifra amarilla que aparece encima de la eliminación.

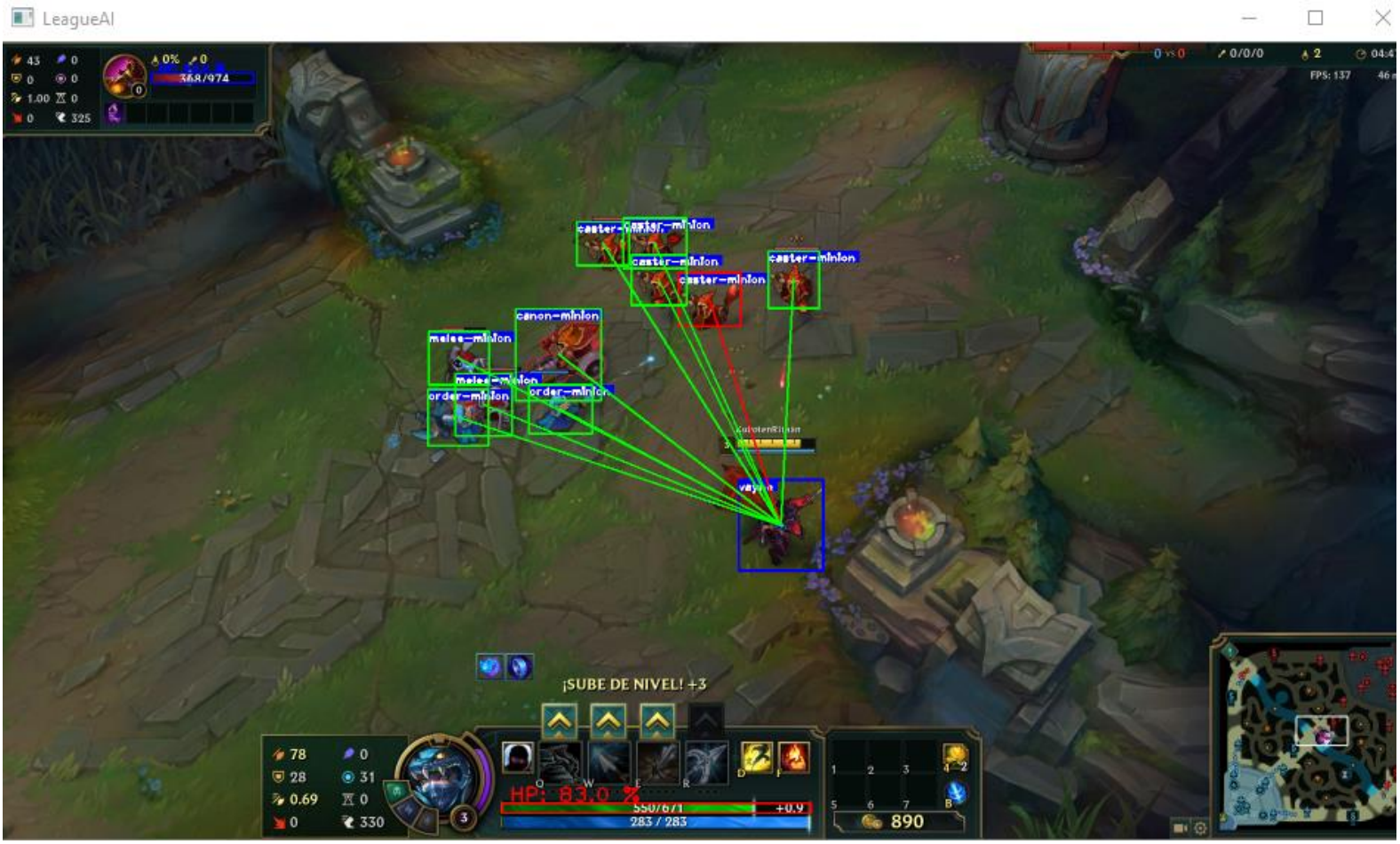
Marcas de oro mostradas al eliminar a un enemigo

Pero al aparecer en posiciones indeterminables, se vuelve muy complicado obtener esos datos.

Por ello una solución es usar el detector de dígitos/caracteres sobre este tipo de elementos que son estáticos en la partida, y, cuando la cifra de oro aumente

sustancialmente entre 2 iteraciones consecutivas del bucle de LeagueAI, significará que se ha realizado una eliminación/asesinato sobre el enemigo seleccionado previamente.

Esto resulta realmente útil para poder crear recompensas (‘rewards’) para el aprendizaje de toma de decisiones del Bot con redes neuronales. Además, también podría llevarse a otras zonas de la imagen para obtener distintos datos como niveles, enfriamiento de habilidades, etc.



Resultado de la toma de decisiones (vector de selección rojo) tras unas detecciones y procesado de fotograma satisfactorios

Conclusiones y Trabajo Futuro

1.Conclusiones Extraídas

A simple vista puede parecer que un videojuego como League of Legends tiene pocos factores a tener en cuenta, pero la conclusión que he obtenido en el estudio de este proyecto, es que hay tantas variables cuando se profundiza en el entendimiento de su jugabilidad, que prácticamente se vuelven inmanejables tanto para crear Bots manualmente que automaticen el juego, como para competir en él profesionalmente.

Por esta razón, la investigación sobre las distintas versiones de YOLO, dónde finalmente concluía en la utilización de YOLOv5 sobre YOLOv3 debido a su mejor rendimiento general y sobre YOLOv4 por su mejor flexibilidad y sencillez (con un rendimiento similar), y el uso de PyTorch frente a TensorFlow, por su mejor compatibilidad con YOLO y CUDA, hacen que se pueda obtener eficazmente la información proporcionada por el videojuego.

Y, aparte de estas tecnologías, considero que el estudio y creación de un Dataset correcto es crucial para proporcionar un nivel de eficiencia suficientemente bueno para aprovechar estas tecnologías en la detección de objetos.

Esto supone el primer y más importante paso para la automatización de los jugadores en este videojuego. Ya que, con estos avances, se extrae de cada imagen más del doble de información que al principio de este proyecto (cuantificando cada distinta detección como objetos del mismo valor), y todos esos datos, obtenidos con mejores resultados y en menor tiempo.

Por ello, todo lo logrado y estudiado en esta última versión proporciona suficiente conocimiento para avanzar al siguiente paso de creación de una IA para el Bot, o, incluso, gracias a la flexibilidad en muchos aspectos, extrapolar el funcionamiento a otros proyectos, como, por ejemplo, videojuegos del mismo género (MOBA). Lo que supone un cumplimiento de los objetivos establecidos de análisis y mejora del proyecto.

2.Tareas Futuras

En cuanto a las posibles mejoras, la principal sería implementar un Bot que fuese entrenado con Aprendizaje por Refuerzo, es decir, que jugase partidas contra jugadores online o contra Bots predeterminados del LoL (aunque tienen un nivel de habilidad inferior) y así aprendiese en base al oro ganado, súbditos eliminados, muertes provocadas, etc.

Una vez completado este aspecto, el siguiente punto podría ser crear archivos de detecciones, más concretamente, pesos del detector YOLO, para detectar todos los campeones del juego, ya que en un solo archivo de pesos no es viable que la detección tuviese que reconocer más de 140 personajes distintos sumados a todas las criaturas del juego.

Y así, obtenidas todas estas mejoras, el último paso a plantear, el cual corresponde a entrenar el Bot en partidas normales de 5c5, lo que supone una complejidad realmente alta, ya que se necesitarían, entre otras muchas cosas, macro decisiones en el juego.

Bibliografía

[1] Videojuego *League of Legends*

[2] Deep Blue vs Kasparov: How a computer beat chess player in the world.

[3] Plataforma de computación y modelo de programación desarrollado por NVIDIA: CUDA

[4] Lenguaje de programación: Python

[5] Compute Unified Device Architecture CUDA

[6] An ultra-fast cross-platform multiple screenshots module: MSS

[7] Python Imaging Library PIL

[8] Real-Time Computer Vision Library: OpenCV

[9] Python's scientific computing package NumPy

[10] A comparison of Two Popular Machine Learning Frameworks

[11] PYTORCH VS TENSORFLOW: Comparing Deep Learning Frameworks

[12] Migra tu código de TensorFlow 1 a TensorFlow 2

[13] Graphical image annotation tool

[14] Intersection over Union for object detection

[15] Non-maximum Suppression

[16] How to implement a YOLO (v3) object detector from scratch in PyTorch

[17] Read YOLO V5 and YOLO V4 in one article

[18] Método Lucas-Kanade

Agradecimientos

En primer lugar, agradecer a Oliver Struckmeier, autor original de LeagueAI, que me proporcionó permisos y ayuda para iniciar y profundizar en este proyecto. También a mi tutor Diego Viejo Hernando por ayudarme a plantear y cerrar el trabajo.

Y, por supuesto doy las gracias a mis padres por apoyarme incondicionalmente a pesar de todos los problemas que doy. A mi abuela, tíos y primos, y a mi grupo de amigos, que siempre están ahí, proporcionandome positividad y ayudándome a alcanzar nuevas metas.

El conocimiento no es suficiente, debemos aplicarlo.
El querer no es suficiente, debemos hacer.
-Bruce Lee