

Agente Softbot para Pruebas Automáticas de GUI: Capa de Visión Computacional

Softbot Agent for Automatic GUI Testing: Computer Vision Layer

Doctorando:

Lucas Martín PRADO

<https://orcid.org/0009-0009-0391-6480>

Universidad Tecnológica Nacional, Facultad Regional Concepción del Uruguay, Grupo de Investigación en Inteligencia Computacional y Optimización de Sistemas, Concepción del Uruguay, Entre Ríos, Argentina
pradol@frcu.utn.edu.ar

Director:

Carlos Antonio CASANOVA PIETROBONI

Codirectora:

María Fabiana PICCOLI

Resumen

En el contexto de testing, si bien existen herramientas de testing automático para realizar pruebas, muchas de ellas dependen de interactuar con el componente a nivel lógico, realizan pruebas a un nivel elemental (probar componentes aislados), o interactúan a nivel de interfaz gráfica pero con técnicas desactualizadas a los últimos avances. La tesis doctoral de la cual este trabajo es parte apunta a construir un agente que sea independiente del proceso de desarrollo del sistema bajo prueba y pueda, por lo tanto, realizar el proceso de testing sin los sesgos que surgen de la autoría del código a probar, ni de la plataforma sobre la cual se ejecuta el programa. En este trabajo se desarrolla la capa de Visión Computacional que el agente utilizará para ver la interfaz gráfica del programa. Para ello, fue creado un dataset nuevo que contiene imágenes de múltiples plataformas y se entrenaron y compararon 13 versiones de modelos distintos de detección de objetos en la tarea de identificar componentes de interfaz gráfica, obteniéndose al final 325 modelos entrenados distintos. De los resultados obtenidos se destaca una correcta identificación de componentes gráficos clave en aplicaciones, y se detectaron también líneas de trabajo futuras, tanto para mejorar el dataset, como para entrenar mejores modelos, focalizando en aquellas variantes que obtuvieron mejor rendimiento.

Palabras clave: Testing; Pruebas Automáticas; Pruebas de GUI; Visión Computacional; Agentes Inteligentes

Abstract

In the context of testing, while there are automated testing tools available, many of them rely on interacting with the component at the logical level, perform tests at a basic level (testing isolated components), or interact at the graphical interface level but use techniques that are outdated compared to the latest advancements. The doctoral thesis of which this work is a part aims to build an agent that is independent of the development process of the system under test and can, therefore, perform the testing process without the biases arising from the authorship of the code to be tested or from the platform on which the program runs. This paper develops the computer vision layer that the agent will use to view the program's graphical interface. To this end, a new dataset containing images from multiple platforms was created, and 13 different versions of object detection models were trained and compared in the task of identifying graphical interface components, resulting in a total of 325 distinct trained models. The results highlight the successful identification of key graphical components in applications and also identified future lines of research, both for improving the dataset and for training better models, focusing on those variants that achieved the best performance.

Keywords: Testing; Automated Testing; GUI Testing; Computer Vision; Intelligent Agents

Introducción

El proceso de testing es una actividad que involucra ejecutar el software o componente de interés para hallar errores y corregirlos, es decir, implica detectar y solucionar problemas en el software con el fin último de que el mismo se comporte como se espera (Bourque y Fairley, 2014). Este proceso cumple un rol fundamental a la hora de mejorar la fiabilidad, seguridad y rendimiento del software, incluso desde una etapa temprana del desarrollo. Al reducir la cantidad y probabilidad de errores en el sistema se mejora la usabilidad del mismo, aumentando a su vez el nivel de satisfacción del usuario que lo utiliza. En definitiva, el proceso de testing es vital para garantizar la calidad del software. En las grandes empresas existen equipos específicos encargados de realizar el proceso de testing, mientras que en Pequeñas y Medianas Empresas (PyMEs) lo normal es que exista superposición de roles, es decir, la persona que prueba el sistema es la misma que lo desarrolla, o posee alguna familiaridad con la forma en la que está construido. Esto puede resultar perjudicial, ya que tener conocimiento sobre la implementación del sistema puede guiar al encargado de realizar las pruebas (tester) por caminos obvios y no intentar caminos menos probables que pueden ocurrir en la realidad, perdiéndose así la oportunidad de encontrar defectos. De hecho, las PyMEs enfrentan muchas dificultades a la hora de implementar buenas prácticas de testing, principalmente por falta de personal dedicado exclusivamente a probar el sistema, por lo que no solo el proceso de testing en las PyMEs recibe una menor inversión de tiempo, sino que además sufre de ser menos eficiente en comparación al de las grandes empresas (Argüello et al., 2021). Estas dificultades acarrearán consigo una pérdida de competitividad de las PyMEs contra empresas ya establecidas.

Si bien para los niveles más bajos de abstracción del sistema desarrollado (por ejemplo, probar un componente o módulo aislado) la industria adoptó rápidamente el uso de pruebas automáticas, lo cierto es que para pruebas que requieren un mayor grado de abstracción, como lo puede ser probar el sistema en su totalidad, las pruebas manuales siguen siendo la práctica más común. Esto último constituye un problema, pues este tipo de pruebas resultan un cuello de botella, retrasando el ritmo de desarrollo de las empresas y, aunque existen algunos casos de empresas de gran tamaño que utilizan algún grado de pruebas automáticas de estas características, suele limitar la velocidad (o la calidad) con la que desarrollan sus productos de software, especialmente las PyMEs, donde la superposición de roles implican una asignación de recursos mayor.

Otro de los problemas asociados a las pruebas automáticas es su mantenimiento (Alégroth et al., 2016). Para que una prueba automática tenga utilidad, la misma tiene que estar actualizada, contemplando los cambios del sistema. A medida que se añaden nuevas funcionalidades y se modifican funcionalidades existentes, se requiere verificar y, de ser necesario, actualizar las mismas. Especialmente para las pruebas de un alto nivel de abstracción, como las pruebas guiadas por la interfaz gráfica de usuario (GUI, Graphical User Interface), esta dificultad es determinante para decidir si se utilizan pruebas automáticas o manuales (Alégroth y Feldt, 2017).

Las herramientas actuales que se basan en reconocimiento de imágenes para automatizar pruebas basadas en la GUI utilizan técnicas anticuadas que provocan un rendimiento menor al que es posible con la tecnología actual. Mediante la utilización de nuevas técnicas en el campo de la Visión Computacional (CV, Computer Vision), principalmente los avances en reconocimiento de imágenes (Sultana et al., 2020), pueden mitigarse los efectos negativos derivados de realizar pruebas de GUI por reconocimiento de imágenes, sin depender de utilizar atajos que dependen de la plataforma de implementación del software desarrollado. A su vez, utilizando técnicas de Inteligencia Artificial (Russell y Norvig, 2004), como lo es el Aprendizaje por Refuerzo (RL, Reinforcement Learning) (Barto y Sutton, 2021; Sutton y Barto, 2018) o la Computación Evolutiva, es posible construir un agente que sea independiente del proceso de desarrollo del sistema y pueda realizar el proceso de testing sin los sesgos que surgen de la autoría del código a probar.

Este trabajo se enfoca en el primer componente de los dos previamente mencionados, es decir, el referente a CV. Para ello, se construyó un dataset nuevo que contiene imágenes de aplicaciones móviles y web. Posteriormente, se desarrollaron una serie de pruebas sobre el dataset resultante, utilizando varios modelos de CV en distintas variantes, con el objetivo de obtener modelos capaces de detectar componentes GUI independientemente de la plataforma y comparar su rendimiento.

Desarrollo

La capacidad de un modelo de reconocimiento basado en redes neuronales artificiales depende en gran medida del dataset de casos con el que se entrena, por lo que se realizó una revisión del estado del arte respecto de dataset compuestos de imágenes de aplicaciones y su GUI, así como también distintos casos de estudio relacionados a la problemática. Específicamente, en el contexto de detección de objetos, los datos están conformados por las imágenes en sí mismas, y las etiquetas, con sus ubicaciones, tamaños, y la clase o categoría a la que pertenece dicha etiqueta. Si bien existen múltiples

casos de estudio donde se evalúan y proporcionan distintos dataset de componentes GUI para la tarea de detección, los mismos suelen estar basados en una única plataforma, ya sean dispositivos móviles o plataformas Web (o de Escritorio).

Uno de los dataset más utilizados es RICO (Deka et al., 2017), un dataset compuesto solo de imágenes provenientes de aplicaciones de Android. Este dataset fue una pieza clave en la construcción de otro dataset mucho más completo: el dataset VINS (Bunian et al., 2021), el cual mejora a la calidad de las etiquetas e incorpora imágenes de otras fuentes, entre ellas aplicaciones que corren en iOS, Wireframes, y provenientes de uplabs. A pesar de ello, el dataset aún se compone solo de imágenes de dispositivos móviles. Existen otros dataset, compuestos por aplicaciones Web (Mejia-Escobar et al., 2023; Hotti et al., 2021), pero el problema de la segregación permanece.

Un trabajo que tiene un enfoque muy similar a esta etapa, aunque surge de una línea de investigación enfocada a la generación de código a partir de GUI, es UIED (Xie et al., 2020). En este trabajo se implementa un conjunto de herramientas para la detección de elementos de la GUI, utilizando distintos modelos entrenados a partir del dataset RICO para reconocimiento de componentes GUI (Chen et al., 2020).

Es cierto que hay características similares entre aplicaciones móviles y web, e incluso las aplicaciones web multiplataforma comparten elementos entre sus distintas plataformas, pero existen distintos mecanismos de interacción y de representación entre una plataforma y otra, incluso la relación de aspecto de la pantalla es distinta según la plataforma, necesitando muchas veces organizar los componentes gráficos de distintas maneras según el dispositivo. El problema de tener dataset de componentes de GUI segregados que no diversifican las plataformas de ejecución no solo ocurre en este dominio. GUI-WORLD (Chen et al., 2024) es un dataset compuesto de videos sobre GUI utilizado para evaluar y mejorar las capacidades de los Grandes Modelos de Lenguaje Multimodales, en este trabajo se realizó un relevamiento de distintos dataset existentes y se encontró el mismo problema: la mayoría de los datos eran dependientes de plataforma. Una de sus contribuciones principales fue la construcción de un dataset que contiene datos de múltiples plataformas. Esta etapa del trabajo apunta a lo mismo, generar un dataset que tenga las fortalezas provistas por el dataset VINS, incorporando imágenes que sean nativas de aplicaciones web en un entorno de aplicación de escritorio.

Dependiendo del origen de los datos y la motivación detrás de la generación de los mismos, tanto la estructura de clases, como el nivel de detalle o la variedad entre los dataset de aplicaciones móviles y dataset web son distintas. Para generar un dataset unificado deben resolverse esas diferencias y establecerse un criterio claro a la hora de determinar las características del dataset. Se simplificó la estructura de clases presentes en los datasets RICO y VINS, y se las agrupó en categorías, esto se hizo a modo de facilitar la comprensión de las distintas clases para ayudar al etiquetado. Las categorías y clases resultantes a utilizar por el dataset unificado las siguientes:

- Componentes de texto: A esta categoría pertenecen, como su nombre indica, aquellas clases que representan texto, o que están formadas por texto, ellas son text y link.
- Componentes iconográficos: Esta categoría se conforma por clases que representan iconos, imágenes, y se incluyen también a los pop-ups, y publicidades. Estas clases son: icon, image, backgroundimage y iframe.
- Componentes funcionales: Dentro de esta categoría hay componentes de los cuales se espera funcionalidad a simple vista, ya sean botones, campos de texto, u otros interactivables. Las clases que pertenecen son: button, textbutton, input, spinner, switch, checkbox, pageindicator y map.
- Componentes contenedores: En esta categoría se encuentran aquellos componentes que son parte de estructura de la aplicación, o que aglutinan otros componentes. Estas clases son: uppertaskbar, panel, toolbar, y multitab.

Posteriormente, se aplicó una transformación de formato y estructura de los archivos de anotaciones, y se elaboró un programa para revisar y editar el dataset, manteniendo el formato compatible con LabelMe, programa con el cual se realizó el etiquetado. Habiendo finalizado el proceso de reestructuración, se coordinaron encuentros y charlas con becarios del Grupo de Investigación en Inteligencia Computacional y Optimización de Sistemas (GIICOS), y se asignaron imágenes Web en tres etapas tanto a los becarios como investigadores del grupo, teniendo cada etapa un proceso de revisión manual por parte del equipo que lidera la línea de investigación. El dataset resultante posee 6202 imágenes, donde 4997 imágenes provienen de aplicaciones móviles, y 1205 provienen de aplicaciones o sitios web.

Para los experimentos se utilizan los modelos de la familia YOLO (Redmon et al., 2016), específicamente sus versiones v5 (Jocher, 2020), v8 (Jocker et al., 2023), v10 (Wang et al., 2024), y v12 (Tian et al., 2025), en sus variantes más pequeñas: nano, small, medium. La decisión de optar por estas versiones se debe a limitaciones de recursos y tiempo, así como también

al interés de analizar el poder de generalización de modelos con cantidades de parámetros pequeñas o medianas. A modo de tener un poco más de variedad se optó también por incluir una versión de Faster R-CNN (Ren et al., 2017) con un backbone Resnet50 (a partir de ahora, frcnn). El objetivo de las pruebas es tener un punto de referencia de distintos modelos, analizando su poder de generalización total, y por clases, para establecer líneas futuras de mejora del dataset. Se realizó un proceso de validación cruzada (cross-validation) separando 20% del dataset, a modo de ser utilizado como dataset de test, este no sería visto en ninguna instancia de entrenamiento, y sería utilizado solo al finalizar el entrenamiento para medir el rendimiento del modelo con imágenes que nunca vió antes. Con el 80% restante se crearon 5 splits, con 4 folds para entrenamiento y 1 utilizado como validación para guiar el entrenamiento, tomando todas las combinaciones posibles de folds. Se realizaron 5 ejecuciones para cada modelo y para cada combinación, cambiando las condiciones de inicialización pero manteniendo el resto de los parámetros. Los parámetros utilizados para entrenar los modelos fueron los siguientes: seeds = [10,20,30,40,50]. Para entrenamiento: batch=16, imgsz=640, epochs=300, patience=10, optimizer=AdamW, lr=0.001, weight_decay=0.01, flipplr=0.5, hsv_h=0.3. Para correr los experimentos se utilizó un único equipo, con una GPU Nvidia GeForce RTX 3080 10 GB VRAM, utilizando CUDA, 16 GB de RAM, y un procesador AMD Ryzen 5 5600x.

Resultados

Se evaluaron principalmente dos métricas para cada modelo, la Precisión Media Promedio al 50% de Intersección sobre la Unión (mAP@0.5), y la Precisión Media Promedio en el rango del 50% al 95% de Intersección sobre la Unión (mAP@0.5:0.95), estas métricas se obtienen de realizar el promedio de la Precisión Promedio obtenida para cada clase individualmente (AP@0.5 y AP@0.5:0.95, respectivamente), se utiliza para determinar el rendimiento general del modelo. En la tabla 1 se observan los resultados de las pruebas realizadas, las mismas se realizaron sobre imágenes de prueba (test images) que no fueron utilizadas por ningún modelo durante su entrenamiento. Para ordenar la tabla, se colocan los modelos por orden cronológico de lanzamiento, del más viejo al más reciente. El peor rendimiento registrado lo tienen los modelos yolov10n (mAP@0.5=0.6877) y yolov5n (mAP@0.5:0.95=0.5852), y el modelo con mejor rendimiento es yolo12m (mAP@0.5=0.7553; mAP@0.5:0.95=0.6616). Además, a partir de estos resultados se puede determinar que los modelos de la versión 12 tienen mejor rendimiento que las variantes que comparten su categoría (nano, small, medium). También, que a pesar de tener una cantidad inferior de parámetros, el rendimiento de yolo12n (2.6M de parámetros) y el de frcnn (41.7M de parámetros) son comparables. El resto del análisis se basa en las tres variantes de YOLO 12 previamente mencionadas.

Modelo	Parámetros (Millones)	mAP@0.5	mAP@0.5:0.95
frcnn	41.7	0.7187	0.5705
yolov5n	2.6	0.6939	0.5852
yolov5s	9.1	0.7223	0.6202
yolov5m	25.1	0.7439	0.6499
yolov8n	3.2	0.7048	0.6032
yolov8s	11.2	0.7282	0.6301
yolov8m	25.9	0.7427	0.6500
yolov10n	2.3	0.6877	0.5907
yolov10s	7.2	0.7169	0.6239
yolov10m	15.4	0.7342	0.6420
yolo12n	2.6	0.7112	0.6145
yolo12s	9.3	0.7373	0.6449
yolo12m	20.2	0.7553	0.6616

Tabla 1: Rendimiento de modelos en imágenes de prueba.

Por motivos de espacio no se colocan los resultados desagregados por clase o categoría (AP@0.5 y AP@0.5:0.95 de cada clase) en formato tabla, pero las observaciones a destacar de los resultados sobre las imágenes de prueba son las siguientes:

La clase `uPERTaskBar` que representa la barra superior de los dispositivos móviles se identifica con un alto nivel de precisión (AP@0.5 - yolo12n: 0.9935; yolo12s: 0.9938; yolo12m: 0.9937). Si bien los modelos identifican correctamente los botones que contienen texto (AP@0.5 - yolo12n: 0.9494; yolo12s: 0.9550; yolo12m: 0.9601), tienen algunas dificultades para identificar los botones que no (AP@0.5 - yolo12n: 0.2898; yolo12s: 0.3810; yolo12m: 0.4554). El modelo identifica bien el texto (AP@0.5 - yolo12n: 0.9331; yolo12s: 0.9367; yolo12m: 0.9401), pero no identifica correctamente links (AP@0.5 - yolo12n: 0.0333; yolo12s: 0.0868; yolo12m: 0.1472). La clase `link` es un tipo de texto, que salvo que esté destacado por formato, es difícil de diferenciar de otros tipos de texto. La tarea de identificar links podría encararse con un enfoque híbrido similar al de UIED, o mejorando la representación de la clase `link`. Respecto del AP@0.5:0.95, los componentes `input` (AP@0.5:0.95 - yolo12n: 0.9004; yolo12s: 0.9173; yolo12m: 0.9256), `textbutton` (AP@0.5:0.95 - yolo12n: 0.9071; yolo12s: 0.9182; yolo12m: 0.9249), `switch` (AP@0.5:0.95 - yolo12n: 0.8767; yolo12s: 0.8952; yolo12m: 0.9227), y `map` (AP@0.5:0.95 - yolo12n: 0.8872; yolo12s: 0.9067; yolo12m: 0.8938) tienen un alto nivel de precisión, siendo un poco menor la precisión para componentes tipo `checkbox` (AP@0.5:0.95 - yolo12n: 0.7101; yolo12s: 0.7526; yolo12m: 0.7868), `pageindicator` (AP@0.5:0.95 - yolo12n: 0.7701; yolo12s: 0.8064; yolo12m: 0.8198), `backgroundimage` (AP@0.5:0.95 - yolo12n: 0.7818; yolo12s: 0.7899; yolo12m: 0.7941) e `image` (AP@0.5:0.95 - yolo12n: 0.7487; yolo12s: 0.7785; yolo12m: 0.7909). Los modelos tienen problemas para identificar con mayor seguridad las clases `button`, `multitab`, `spinner`, y `link`, como se discutió anteriormente. El bajo nivel de detección de estas clases penaliza el mAP@0.5 y mAP@0.5:0.95 de los modelos al realizar el promedio, por lo que mejorar su identificación implicaría el mayor salto de calidad en la métrica de la precisión del modelo. Se destaca una identificación precisa de clases clave como: `text`, `input`, `map`, `textbutton`, `checkbox`, `switch`, `icon`, `image`, `pageindicator`, y `map`, que son piezas clave que construyen gran parte de las aplicaciones, considerando que hay un rumbo de mejora claro para los botones sin texto, los pop-ups (`iframe`), y los links.

Conclusiones

Se construyó un dataset de componentes GUI independiente de plataforma que contiene tanto imágenes de aplicaciones móviles como de web, en entornos de escritorio. Además, se realizaron pruebas exhaustivas sobre el mismo utilizando múltiples modelos de CV, obteniéndose resultados prometedores respecto de la capacidad de generalización de estos modelos así como también posibles caminos de mejora, tanto del dataset como del rendimiento de los modelos mismos. El proceso de entrenamiento y pruebas de estos modelos deja como resultado adicional 325 modelos entrenados con este dataset, con los cuales realizar inferencias sobre imágenes de componentes GUI, pudiendo utilizar el de mejor rendimiento para cada variante. Como trabajo futuro existe la posibilidad de agregar más imágenes para mejorar el dataset, y de entrenar más modelos de la familia YOLO (versión 12), partiendo de distintas condiciones de inicialización y utilizando distintos parámetros de entrenamiento, y su futura comparación con los mejores obtenidos hasta el momento.

Referencias

- Alegroth, E., Feldt, R., & Kolström, P. (2016). Maintenance of Automated Test Suites in Industry: An Empirical study on Visual GUI Testing. *Information and Software Technology*, 73. doi:10.1016/j.infsof.2016.01.012
- Alégroth, E., & Feldt, R. (2017). On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering*, 22(6), 2937–2971. doi:10.1007/s10664-016-9497-6
- Argüello, M., Casanova Pietroboni, C. A., & Cedaro, K. E. (2021). Proposal to Improve Software Testing in Small and Medium Enterprises. In H. Florez & M. F. Pollo-Cattaneo (Eds.), *Applied Informatics* (pp. 497–512). Cham: Springer International Publishing.
- Barto, A. G., Sutton, R. S., & Anderson, C. W. (2021). Looking Back on the Actor–Critic Architecture. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(1), 40–50. doi:10.1109/TSMC.2020.3041775
- Bourque, P., & Fairley, R. E. (Eds.). (2014). *SWEBOK: Guide to the Software Engineering Body of Knowledge* (Version 3.0).
- Bunian, S., Li, K., Jemmali, C., Hartevelde, C., Fu, Y. R., & El-Nasr, M. S. (2021). VINS: Visual Search for Mobile User Interface Design. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Retrieved from <https://api.semanticscholar.org/CorpusID:231861912>

- Chen, D., Huang, Y., Wu, S., Tang, J., Chen, L., Bai, Y., ... Sun, L. (2024). GUI-World: A Video Benchmark and Dataset for Multimodal GUI-oriented Understanding. Retrieved from <https://api.semanticscholar.org/CorpusID:270560712>
- Chen, J., Xie, M., Xing, Z., Chen, C., Xu, X., Zhu, L., & Li, G. (2020). Object detection for graphical user interface: old fashioned or deep learning or a combination? Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 1202–1214. Presented at the Virtual Event, USA. doi:10.1145/3368089.3409691
- Deka, B., Huang, Z., Franzen, C., Hibschan, J., Afegan, D., Li, Y., ... Kumar, R. (2017). Rico: A Mobile App Dataset for Building Data-Driven Design Applications. Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology. Retrieved from <https://api.semanticscholar.org/CorpusID:6623010>
- Hotti, A. M., Risuleo, R. S., Magureanu, S., Moradi, A., & Lagergren, J. (2021). The Klarna Product Page Dataset: Web Element Nomination with Graph Neural Networks and Large Language Models. Trans. Mach. Learn. Res., 2024. Retrieved from <https://api.semanticscholar.org/CorpusID:253107917>
- Jocher, G. (2020). Ultralytics YOLOv5 (Version 7.0). doi:10.5281/zenodo.3908559
- Jocher, G., Chaurasia, A., & Qiu, J. (2023). Ultralytics YOLOv8 (Version 8.0.0). Retrieved from <https://github.com/ultralytics/ultralytics>
- Mejia-Escobar, C., Cazorla, M., & Martínez-Martín, E. (2023). A Large Visual, Qualitative, and Quantitative Dataset for Web Intelligence Applications. Computational Intelligence and Neuroscience. Retrieved from <https://api.semanticscholar.org/CorpusID:264071872>
- Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016, June). You Only Look Once: Unified, Real-Time Object Detection. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 779–788. doi:10.1109/CVPR.2016.91
- Ren, S., He, K., Girshick, R., & Sun, J. (2017). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Transactions on Pattern Analysis & Machine Intelligence, 39(06), 1137–1149. doi:10.1109/TPAMI.2016.2577031
- Russell, S. J., & Norvig, P. (2004). Inteligencia Artificial: un enfoque moderno. Pearson.
- Sultana, F., Sufian, A., & Dutta, P. (2020). A Review of Object Detection Models Based on Convolutional Neural Network. In J. K. Mandal & S. Banerjee (Eds.), Intelligent Computing: Image Processing Based Applications (pp. 1–16). doi:10.1007/978-981-15-4288-6_1
- Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (Second). Retrieved from <http://incompleteideas.net/book/the-book-2nd.html>
- Tian, Y., Ye, Q., & Doermann, D. (2025). YOLOv12: Attention-Centric Real-Time Object Detectors. Advances in Neural Information Processing Systems, 38, 78433–78457. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2025/file/7103444259031cc58051f8c9a4868533-Paper-Conference.pdf
- Wang, A., Chen, H., Liu, L., Chen, K., Lin, Z., Han, J., & Ding, G. (2024). YOLOv10: Real-Time End-to-End Object Detection. Advances in Neural Information Processing Systems, 37, 107984–108011. doi:10.52202/079017-3429
- Xie, M., Feng, S., Xing, Z., Chen, J., & Chen, C. (11 2020). UIED: a hybrid tool for GUI element detection. doi:10.1145/3368089.3417940