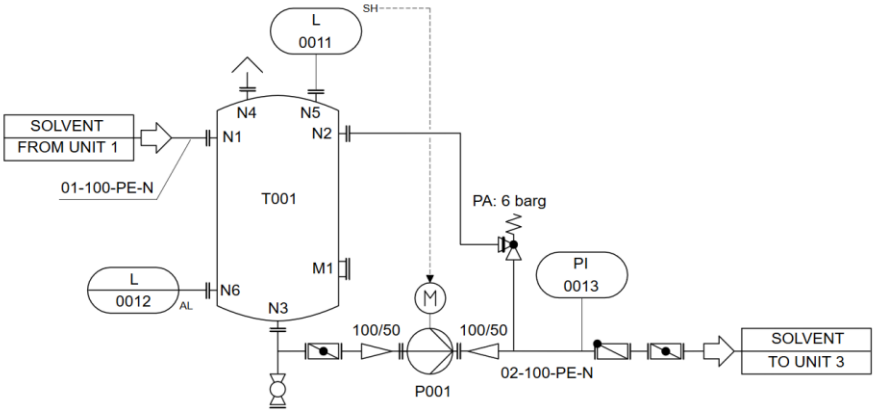
TRABAJO FIN DE MÁSTER

Sistema de detección de componentes en diagramas de tuberías e instrumentación

1. MOTIVACION DEL PROYECTO

Para muchos procesos de la industria se emplean diagramas que muestran las tuberías y componentes relacionados del proceso físico que representan. Estos diagramas de denominan P&ID (por sus siglas en inglés: Piping and Instrumentation Diagram) y se utilizan ampliamente en el campo de la ingeniería.

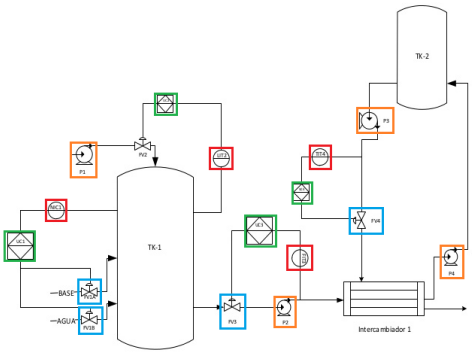


Estos diagramas son fundamentales para diversas tareas como el diseño o el mantenimiento, y en muchos casos solo se dispone de ellos en papel, lo que provoca que se tenga que realizar un trabajo manual de identificación de los componentes.

Este trabajo manual, unido a que en ocasiones se trabaja del orden de miles de planos, es lo que ha motivado la realización de este TFM para la automatización de estas tareas.

2. OBJETIVOS

El objetivo del proyecto sería el reconocimiento de los diferentes componentes que hay en un plano, identificando su tipo y la posición donde está ubicado: coordenada x, y, ancho y alto del área que engloba al componente.



3. ENTORNO

Estructura del repositorio:

- Configs: Contiene los pipelines empleadas para el entrenamiento de los modelos y la configuración del Cluster utilizada en Google Cloud.
- DatasetCreator: Código fuente del programa de generación aleatoria de diagramas PID
- DetectionComponentsAPI: Código fuente de la API utilizada por la aplicación de visualización

- Frontend: Código fuente de la aplicación de visualización
- PatternImages: Imágenes patrón para la generación aleatoria de diagramas
- TestImages: Contiene imágenes de diagramas para realizar pruebas de inferencias.
- .gitignore y .dockerignore: Archivos ignorados para Git y Docker respectivamente
- ComponentDetection.ipynb: Notebook con el proceso para la generación, entrenamiento y prueba del modelo de detección de componentes
- GoogleCloudAITrainingConfig.ipynb: Notebook con la configuración y pasos necesarios para realizar el entrenamiento en Google Cloud
- Dockerfile: Archivo para la generación de la imagen de Docker
- Start.sh: Script para iniciar los procesos necesarios dentro del contenedor Docker

Para facilitar la ejecución y evitar tener que instalar todas las dependencias necesarias se ha creado una imagen Docker con todo lo necesario ya configurado. Para crearnos un contenedor y ejecutarlo a partir de la imagen hay que ejecutar:

```
docker run -d -p 8888:8888 -p 6006:6006 -p 4200:4200 -p 5050:5050 macomino/tfm
```

Los puertos siguientes son necesarios para los siguientes servicios:

- 8888 Jupyter Notebooks
- 6006 Interfaz gráfica de tensorboard
- 4200 Frontend de visualización del usuario final
- 5050 Backend para la visualización del usuario final

La imagen Docker generada para este TFM parte de Ubuntu:18.10 donde se han ido instalando las diferentes dependencias y paquetes necesarios, además del código desarrollado. Para más detalles de la generación de la imagen se puede consultar el fichero Dockerfile ubicado en la raíz del repositorio. Los notebooks aportados y todo el código fuente se encuentran dentro de la carpeta TFM de la imagen

4. OBTENCION DE DATOS

Debido a la cantidad de imágenes etiquetadas que harían falta para poder entrenar el modelo y para evitar problemas de propiedad intelectual de los planos, se ha optado por generar de forma automática los diagramas PID.

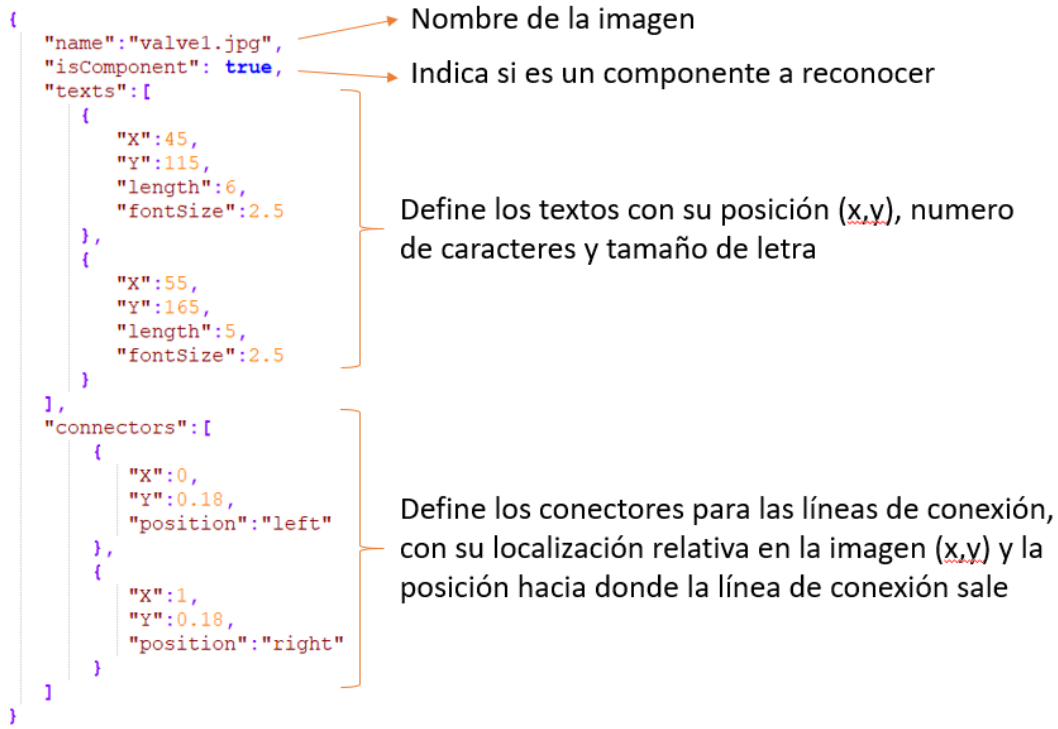
Para la generación automática de los diagramas se ha desarrollado una utilidad que a partir de las imágenes disponibles en la carpeta PatternImages, donde se almacenan los diferentes tipos de componentes, genera un diagrama de tamaño aleatorio. Los diferentes componentes se sitúan también de forma aleatoria dentro del diagrama, modificando sus propiedades como el tamaño, brillo o contraste

A su vez, para cada componente se generan, igualmente de forma aleatoria, las etiquetas con el texto que lo identifican.

Por último, algunos componentes tienen definidos una serie de conectores para poder generar las líneas de conexión con otros componentes.

Todas estas propiedades se encuentran definidas en el archivo properties.json ubicado dentro de la carpeta PatternImages

A continuación, se muestra la definición las propiedades del componente “valve1.jpg”



Para el conjunto de imágenes generadas se genera además un fichero csv que contiene las etiquetas de cada componente, la posición dentro de la imagen y su tamaño.
Por último, la herramienta empaqueta todas las imágenes, junto con sus etiquetas, en un fichero TFRecord, que es el formato de datos que Tensorflow espera para los dataset de training y test.

5. GENERACION DE DATOS PARA ENTRENAMIENTO Y TEST

La generación de los dataset de training y test se pueden realizar desde el notebook ComponentDetection.ipynb que se ubica dentro de la carpeta TFM.
Para generar los dataset de training y test creamos una nueva instancia de la clase datasetCreate, pasándole el path donde se encuentran las imágenes de los componentes que se usarán para generar el diagrama y el path de salida de los ficheros generados

```
from datasetCreate import DatasetCreate

imagesInputPath = 'input'
outputPath = 'output'
dc = DatasetCreate(imagesInputPath, outputPath)
```

Posteriormente llamamos al método generateDataset, indicándole el número de imágenes, la carpeta donde se van a generar dichas imágenes y el nombre del fichero TFRecord de salida

```
dc.generateDataset(100, 'imagesTraining', 'train.record')
```

6. ELECCIÓN DE MODELO

El problema planteado consiste en la identificación, clasificación y localización de múltiples objetos en una imagen. En la actualidad, es una de las tareas más complicadas en el campo de la visión artificial y constituye un desafío en la realización de este trabajo.

Para la realización de nuestro modelo es imprescindible el uso de redes neuronales y más concretamente un tipo específico, denominadas redes neuronales convoluciones (CNN) ya que son muy efectivas para las tareas de visión artificial.

Para la implementación del modelo usaremos Tensorflow, una biblioteca de código abierto para Machine Learning desarrollada por Google. Dentro de su repositorio en GitHub, Tensorflow ofrece un apartado denominado *research models* que contiene una gran colección de modelos implementados por investigadores. Uno de estos modelos es *Tensorflow object dectection* (https://github.com/tensorflow/models/tree/master/research/object_detection) que construido sobre Tensorflow facilita la construcción, entrenamiento y despliegue de modelos de detección de objetos

7. CONFIGURACION DEL ENTRENAMIENTO

La configuración del modelo en Tensorflow object detection se realiza mediante una pipeline que se guarda en un fichero *.config donde se define toda la configuración necesaria para entrenar el modelo.

El esquema de este fichero se muestra a continuación:

```
model {
(... Add model config here...)
}

train_config : {
(... Add train_config here...)
}

train_input_reader: {
(... Add train_input configuration here...)
}

eval_config: {
}

eval_input_reader: {
(... Add eval_input configuration here...)
}
```

8. SELECCIÓN DE PARAMETROS EN EL MODELO

Hay un gran numero de parámetros para configurar los modelos. La elección de la mejor configuración es un proceso complejo que llevaría mucho tiempo, por lo que se ha partido de modelos pre-entrenados, que además de aportar una configuración base sobre la que trabajar, reducen el tiempo de entrenamiento al partir nuestro modelo de uno ya entrenado.
Los modelos pre-entrenados se pueden obtener en https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md.
Todos los modelos evaluados en este trabajo parten del pre-entranamiento basado en el COCO dataset. Este dataset cuenta con 330000 imágenes con 1,5 millones de objetos en 80 categorías.

Los modelos que se han evaluado en este TFM son los siguientes:

R-FCN Resnet	https://arxiv.org/abs/1605.06409
SSD Mobilenet	https://arxiv.org/abs/1512.02325
Faster rcnn inception resnet	https://arxiv.org/abs/1504.08083

9. ENTRENAMIENTO DEL MODELO

Inicialmente el modelo se comenzó a entrenar desde la propia imagen de Docker, pero debido a la baja velocidad que se conseguía debido a las características con las que estaba configurado el contenedor (contenedor Linux sobre un equipo con Windows) se optó por buscar una alternativa.

La primera opción fue entrenar el modelo directamente sobre el equipo Windows, donde mejoro algo la velocidad, pero aun era insuficiente.

Como segunda alternativa se comenzó a utilizar Google Colab, ya que es una plataforma gratuita que te ofrece la posibilidad de utilizar GPUs. Utilizando Google Colab se consiguió una velocidad de entrenamiento aceptable, pero al ser un servicio gratuito no te garantizan la disponibilidad, lo que provocaba que tras un tiempo de entrenamiento te desasignaran los recursos y se parará el entrenamiento.

Como tercera alternativa y definitiva se empleo el servicio AI Platform de Google Cloud.

Mediante este servicio la tarea se puede subir a un Cluster de servidores donde se llevará a cabo el proceso de entrenamiento reduciendo los tiempos.

La configuración empleada se puede encontrar en el fichero cloud.yml dentro de la carpeta Configs

```
trainingInput:
  runtimeVersion: "1.12"
  scaleTier: CUSTOM
  masterType: standard_gpu
  workerCount: 0
  workerType: standard_gpu
  parameterServerCount: 0
  parameterServerType: standard
```

Con esta configuración disponemos de un servidor con una NVIDIA Tesla K80.

Para realizar el entrenamiento en Google Cloud se dispone de otro notebook en la carpeta TFM/ GoogleCloudAITrainingConfig.ipynb con los pasos necesarios para configurar y lanzar el proceso.

10. METRICAS

En la detección de objetos tenemos dos tareas que medir, por un lado, determinar si un objeto existe o no en la imagen, que seria un problema de clasificación, y por otro, determinar la posición de un objeto en la imagen, que sería un problema de localización.

Por lo tanto, la métrica estándar de precisión utilizada en los problemas de clasificación de imágenes no se puede aplicar directamente aquí

Para bordar esta tarea existen las siguientes métricas:

IoU (Intersection over union): Mide la superposición entre 2 límites. Se emplea para medir cuánto se superpone nuestro límite predicho con el límite del objeto real.

AP (Average Precision): La precisión mide la "tasa de falsos positivos" o la proporción de detecciones de objetos verdaderos respecto al número total de objetos que el clasificador predijo. Para realizar esta medición se emplea IoU. Si tiene un valor de precisión cercano a 1, existe una alta probabilidad de que lo que el clasificador predice como una detección positiva sea en realidad una predicción correcta.

mAP (mean Average Precision): Es una extensión de AP donde tomamos el promedio de todos los AP calculados.

Recall: Mide la "tasa de falsos negativos" o la proporción de detecciones de objetos verdaderos con respecto al número total de objetos en el conjunto de datos. Si tiene una puntuación de recuperación cercana a 1, el modelo detectará positivamente casi todos los objetos que están en su conjunto de datos

Durante el entrenamiento del modelo, el panel de Tensorflow nos ofrece diferentes gráficas basadas en las métricas anteriormente mencionadas. En la siguiente tabla se describe cada una de las medidas obtenidas:

DetectionBoxes_Precision	
DetectionBoxes_Precision/mAP	Mean average precisión de las clases promediada con la IoU que van desde 0.5 a 0.95 en incrementos de 0.05
DetectionBoxes_Precision/mAP@.50IOU	Mean average precision con 50% IOU
DetectionBoxes_Precision/mAP@.75IOU	Mean average precision con 75% IOU
DetectionBoxes_Precision/mAP (small)	Mean average precision para imágenes pequeñas (área < 32^2 pixeles).
DetectionBoxes_Precision/mAP (medium)	Mean average precision para imágenes medias (32^2 pixeles < área < 96^2 pixeles).
DetectionBoxes_Precision/mAP (large)	Mean average precision para imágenes grandes (96^2 pixeles < área < 10000^2 pixeles).
DetectionBoxes_Recall	
DetectionBoxes_Recall/AR@1	Recall medio con 1 detección
DetectionBoxes_Recall/AR@10	Recall medio con 10 detección
DetectionBoxes_Recall/AR@100	Recall medio con 100 detección
DetectionBoxes_Recall/AR@100 (small)	Recall medio para imágenes pequeñas con 100 detecciones
DetectionBoxes_Recall/AR@100 (medium)	Recall medio para imágenes medias con 100 detecciones
DetectionBoxes_Recall/AR@100 (large)	Recall medio para imágenes grandes con 100 detecciones
Loss	
BoxClassifierLoss/classification_loss	Pérdidas para la clasificación de objetos detectados con varias clases.
BoxClassifierLoss/localization_loss	Pérdidas de localización o las pérdidas del regresor del área delimitadora
RPNLoss/localization_loss	Pérdidas de localización o las pérdidas del regresor del área delimitadora para el RPN
RPNLoss/objectness_loss	Pérdidas del clasificador que decide si un área delimitadora es un objeto de interés o de fondo.

11. EVALUACION RESULTADOS

Todos los entrenamientos se han realizado con un dataset de 1000 imágenes para el training y 100 para el test (un 10%).

El número medio de componentes para reconocer por imagen ha sido de unos 45 componentes y el tamaño medio de la imagen de entrada es de 5500 pixeles x 5500 pixeles.

Los resultados de los entrenamientos se muestran en las siguientes tablas en el mismo orden en el que se realizaron. En cada tabla se visualiza además del valor de las métricas antes descritas, el numero de evaluaciones que se han realizado y el tiempo total de entrenamiento que ha llevado.

Detection Boxes Precision

Model	Step	mAP	mAP (large)	mAP (medium)	mAP (small)	mAP@.50IOU	mAP@.75IOU	Time
rfcn_resnet101	100000	0.6409	0.6574	0.63	-1	0.8786	0.7633	1d 1h
ssd_mobilenet	134000	0.028	0.026	0.039	-1	0.067	0.018	20h
faster_rcnn	47000	0.7112	0.7114	0.7402	-1	0.8421	0.8282	3d 17h

Detection Boxes Recall

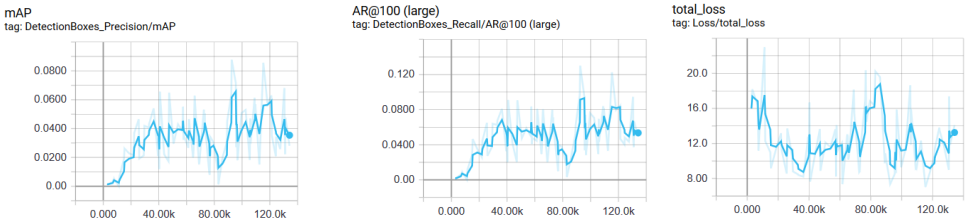
Model	Step	AR@1	AR@10	AR@100	AR@100 (large)	AR@100 (medium)	AR@100 (small)	Time
rfcn_resnet101	100000	0.2985	0.7446	0.7595	0.7712	0.7461	-1	1d 1h
ssd_mobilenet	134000	0.029	0.049	0.049	0.04897	0.05177	-1	20h
faster_rcnn	47000	0.3201	0.7726	0.7741	0.7708	0.7945	-1	3d 17h

Loss

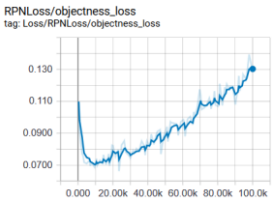
Model	Step	BoxClassifierLoss/ classification_loss	BoxClassifierLoss/ localization_loss	RPNLoss/ localization_loss	RPNLoss/ objectness_loss	Time
rfcn_resnet101	100000	0.1663	0.1183	0.1082	0.1287	1d 1h
ssd_mobilenet	134000	11.17	2.46			20h
faster_rcnn	47000	0.1872	0.07768	0.1628	0.084	3d 17h

De los resultados obtenidos podemos observar:

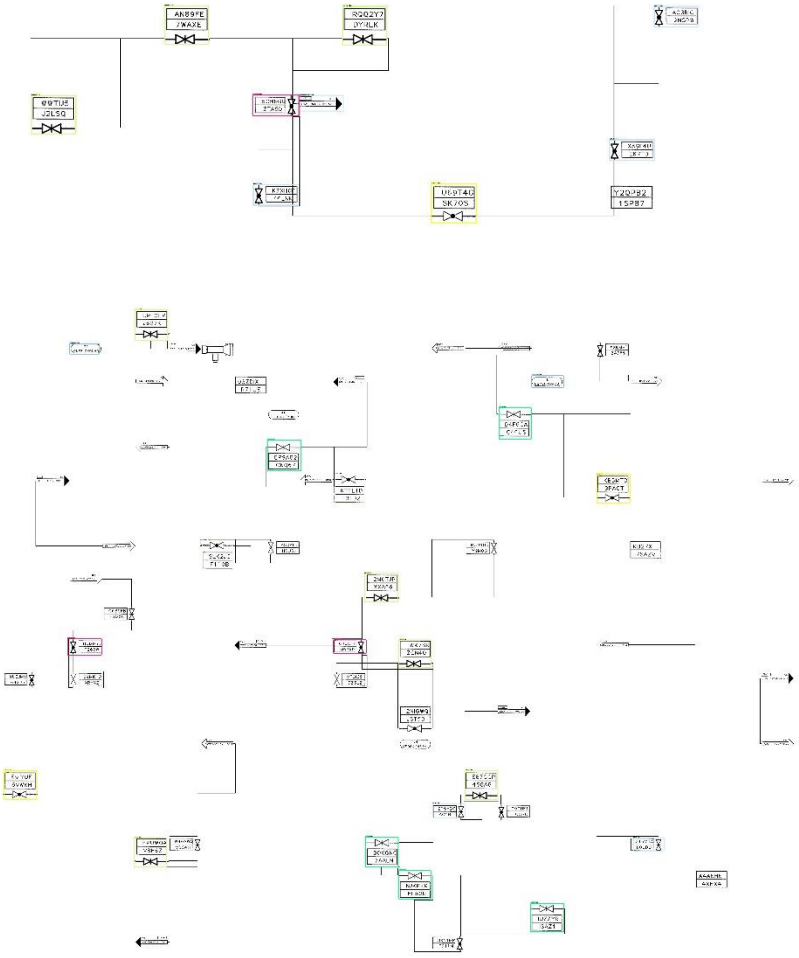
- El mAP y AR para imágenes pequeñas nunca converge. Esto es debido a que en nuestro dataset no hay imágenes que cumplan estas características.
- El AR con una detección (AR@1) obtiene resultados muy bajos debido a que hay pocas imágenes en el dataset que solo contengan un componente para reconocer
- Ssd_mobilenet no ha obtenido resultados satisfactorios. Es el modelo que más velocidad tiene de entrenamiento, en 20h ha realizado 134000 evaluaciones, pero las medidas nunca convergían. Esto puede ser debido a que en la entrada al modelo las imágenes se redimensionan para mejorar el rendimiento. Este redimensionado es posible que fuera demasiado alto para la resolución necesaria para llevar a cabo un buen reconocimiento. Otro factor que también puede haber influido es que el modelo espera imágenes cuadradas, lo cual pude provocar deformaciones en los componentes a reconocer.



- Rfcn resnet es una variación de Faster RCNN donde se aumenta la velocidad de 2.5-20x. En Rfcn resnet es considerable la mayor velocidad de entrenamiento: en 1 día ha realizado 100000 evaluaciones, mientras que Faster RCNN no ha llegado a la mitad en más del triple del tiempo. El problema de Rfcn restnet fue la aparición de overfitting:



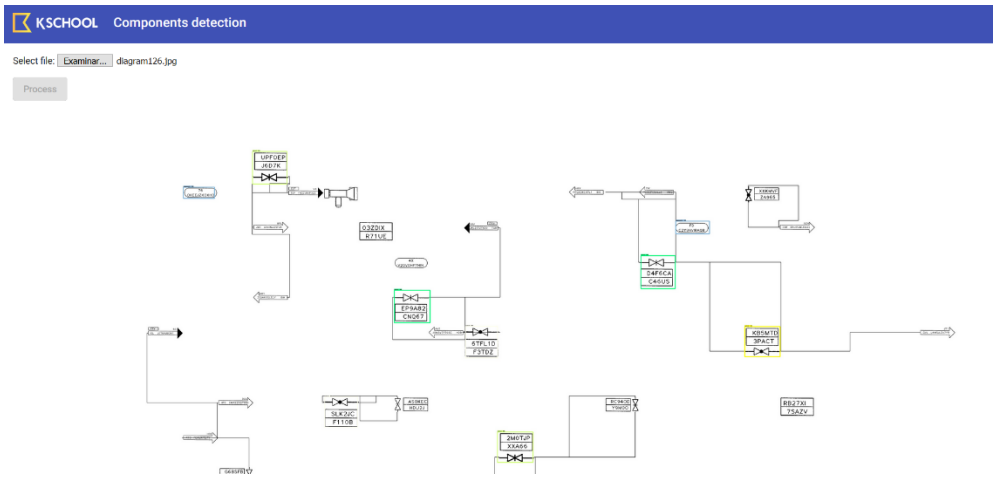
- De lo anteriormente observado podemos concluir que el mejor modelo entrenado ha sido faster rcnn. A continuación, se muestran algunos ejemplos de planos inferidos por el modelo



12. VISUALIZACION

Para el empleo y visualización del mejor modelo entrenado se ha desarrollado un API Rest disponible desde el puerto 5050. Esta API tiene un método que requiere como parámetro de entrada la imagen a la que se quiera realizar la inferencia. Como resultado, nos devuelve la imagen con los elementos marcados que se han reconocido junto con su probabilidad.

Por otro lado, para mejorar la visualización se ha desarrollado una pequeña aplicación Web basada en Angular, la cual nos permite seleccionar una imagen, la envía al API Rest y nos devuelve la imagen con los elementos reconocidos. Esta aplicación es accesible desde el puerto 4200. (http://localhost:4200/)



En el repositorio, dentro de la carpeta TestImages, hay algunos diagramas de ejemplo para realizar pruebas desde el frontend.

13. CONCLUSIONES

El objetivo principal planteado se ha cumplido.

Con el modelo entrenado se consigue reconocer satisfactoriamente los tipos de componentes y sus posiciones dentro de la imagen.

Además, con la herramienta desarrollada para la creación automática de diagramas es muy sencillo introducir nuevos componentes para reconocer e incluso entrenar el modelo para otro tipo de diagramas que usen representaciones diferentes de componentes.

Miguel Ángel Comino Mateos (@macominom), Madrid, junio 2019