Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Visualización e interpretación de redes neuronales convolucionales con Dial-app

Visualization and interpretation of convolutional neural networks with Dial-App

Javier Duque Melguizo

La Laguna, 12 de Junio de 2021

D. **Rafael Arnay del Arco**, con N.I.F. 78.569.591-G profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Jose Francisco Sigut Saavedra**, con N.I.F. 12.345.678-X profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

“Visualización e interpretación de redes neuronales convolucionales con Dial-app”

ha sido realizada bajo su dirección por D. **Javier Duque Melguizo**,

con N.I.F. 54.109.078-Z.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 28 de Junio de 2021

Agradecimientos

A mi tutor Rafael Arnay del Arco, y cotutor, José Francisco Sigut Saavedra, por su apoyo y compromiso demostrado para sacar este proyecto adelante

A mi familia y amigos, por haberme apoyado durante toda mi etapa universitaria hasta llegar a este momento

Licencia

Resumen

El Aprendizaje Profundo o Deep Learning basado en redes neuronales convolucionales ha presenciado un gran aumento en su rendimiento y complejidad en esta última década, lo que ha permitido resolver problemas muy complejos de clasificación y segmentación de imágenes

Sin embargo, el enorme rendimiento alcanzado por estos sistemas choca con la poca o nula capacidad de los mismos para explicar el porqué de sus decisiones. Este carácter de cajas negras compromete seriamente su aplicabilidad en campos donde los expertos necesitan "entender" las razones del porqué de las decisiones que toma el sistema.

Por este motivo, en los últimos años se han desarrollado multitud de técnicas para "abrir" esta caja y visualizar lo que está ocurriendo dentro accediendo a las activaciones de las neuronas en capas intermedias de la red o evaluando las salidas de la misma cuando alteramos la entrada de forma controlada.

En este sentido, el objetivo de este TFG es desarrollar un plugin para la herramienta Dial con la finalidad de incorporar funcionalidades de visualización sobre las activaciones de las neuronas.

Palabras clave: Aprendizaje Profundo, Depuración, Redes Neuronales Convolucionales

Abstract

Deep Learning or Deep Learning based on convolutional neural networks has witnessed a great increase in the performance and complexity of these networks in the last decade, which has made it possible to solve very complex problems of classification and segmentation of images, such as the problem of autonomous driving.

However, the enormous performance achieved by these systems collides with their little or no capacity to explain the reasons for their decisions.This character of black boxes seriously compromises their applicability in fields where experts need to "understand" the reasons for the decisions made by the system.

For this reason, in recent years a multitude of techniques have been developed to "open" this box and visualize what is happening inside by accessing the activations of neurons in intermediate layers of the network or evaluating its outputs when we alter the network. controlled entry.

In this sense, the objective of this TFG is to develop a plugin for the Dial tool in order to incorporate visualization functionalities on the activations of neurons..

Keywords: Deep Learning, Debugging, Convolutional Neural Networks

Índice general

Índice de figuras

Índice de tablas

Índice de Listados

Capítulo 1

Introducción

Este capítulo servirá para poner en contexto el proyecto desarrollado. Se presentarán los objetivos del mismo junto con los resultados obtenidos, comparándolos con otros proyectos presentes hoy en día que busquen cumplir los mismos objetivos.

1.1 Antecedentes

El primer prototipo de red neuronal convolucional estaba basado en el Neocognitron [2] modelo introducido por Kunihiko Fukushima [1] en 1980. Esta arquitectura fue recuperada y mejorada en 1998 por Yann LeCun et al [3], mediante la inclusión de la popular técnica del *backpropagation* que es hoy la base del aprendizaje supervisado de todas las redes neuronales modernas. Sin embargo, las limitaciones en el procesamiento de datos en la época retrasó su popularización hasta 2012, cuando Dan Ciresan y otros [4] implementaron la parte del procesamiento de datos de estas redes en los núcleos de las *unidades procesamiento gráfico* o *GPU,* y ello ha conducido al auge del *Aprendizaje Profundo* o *Deep Learning* [5] que hoy contemplamos.

El desarrollo de las redes neuronales es, cada día, una tarea más sencilla gracias al desarrollo o liberación de librerías de código abierto como TensorFlow [6] y Keras [7] en 2015. A pesar de este desarrollo, el Deep Learning sigue arrastrando una problemática desde antes incluso de su auge, la capacidad de explicar al usuario las razones o causas que llevan a una CNN (*convolutional neural network*, por sus siglas en inglés) a una decisión o clasificación concreta.

La necesidad de técnicas o herramientas para depurar redes neuronales con el objetivo de entender las decisiones finales de estas, lleva siendo un problema recurrente en el desarrollo de redes neuronales desde su existencia. Un problema que ha ido en crecimiento a medida que la complejidad de estos modelos escalaba hasta el punto de situarse como una de sus principales desventajas a día de hoy, de modo que estas redes son directamente entendidas como *cajas negras* por muchos de los usuarios de las mismas.

Con el objetivo de solventar este problema, se han propuesto varias técnicas, las cuales

se pueden clasificar en 4 grupos, según lo que visualizan:

- Visualización de pesos.
- Visualización de activaciones.
- Visualización de filtros.
- Mapa de atribuciones. A su vez, desglosados en varios tipos según los fundamentos que exploten:
    - Basados en gradientes (*Gradient-based backpropagation methods*)
    - Basados en mapas de activación de clase.(*Class Activation Map*)
    - Basados en perturbaciones (*Perturbation-based forward propagation methods*) .

Estas técnicas serán introducidas y explicadas en detalle más adelante en el capítulo 4 de esta memoria.

1.2 Objetivos

Como ya se ha comentado anteriormente, las redes neuronales presentan la desventaja de comportarse como *cajas negras* y así son entendidas por muchos desarrolladores. A diferencia de la depuración de código, dada la complejidad de la arquitectura de las redes neuronales y el fuerte componente matemático sobre el que se basan, intentar entenderlas a bajo nivel es una tarea titánica que, en la mayoría de las ocasiones, no valdrá la pena, porque lo que al final se obtiene es un escalar, un vector o una matriz que no aporta nada a nuestro entendimiento.

Así, el objetivo principal de este proyecto es implementar varias técnicas que permitan entender, **visualmente**, la razón detrás de las decisiones finales tomadas por una **red entrenada**. Para ello, deben implementarse algoritmos que permitan, para cada capa convolucional dentro de una red neuronal convolucional:

- **Visualizar pesos**. (pesos y bias de cada neurona de una capa)
- **Visualizar salidas**. (mapas de características o predicciones finales)
- **Visualizar filtros.**
- **Visualizar mapas de atribución**.

Y todo ello debe realizarse con una interfaz que cumpla los siguientes objetivos:

- **Usable**. La herramienta debe ser intuitiva de usar. Su funcionamiento debe poder entenderse a simple vista.
- **Potente**. La interfaz admite ser ampliada fácilmente, por ejemplo, para incorporar nuevos algoritmos de explicación de decisiones.

También se planteó como objetivo que estas técnicas de visualización deben ayudar a entender las razones detrás de la decisión final de una red entrenada, sin importar el

grado de especialización del público.

Como resultado de lo anterior, se desarrolló Dial-Visualization [48]

1.2.1 Dial Visualization

Dial-Visualization [48] es un plugin de la aplicación Dial-App [15] escrito en Python [16] que incorpora una herramienta de interfaz gráfica desarrollada en QT [17] que, entre sus funcionalidades se listan:

- Permite cargar un modelo de red neuronal convolucional previamente entrenado
- Permite visualizar su estructura interna. (Capas, pesos y bias)
- Permite visualizar varias técnicas que permiten ,o ayudan, al usuario a entender las razones en la clasificación final del modelo.

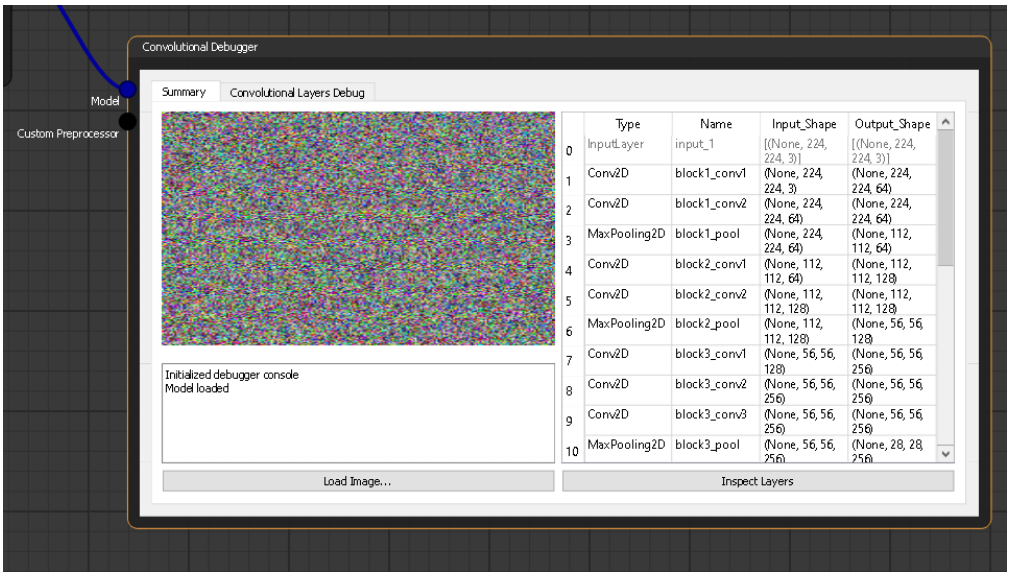


Figura 1.1: Ejemplo de la primera vista del nodo Dial-Visualization

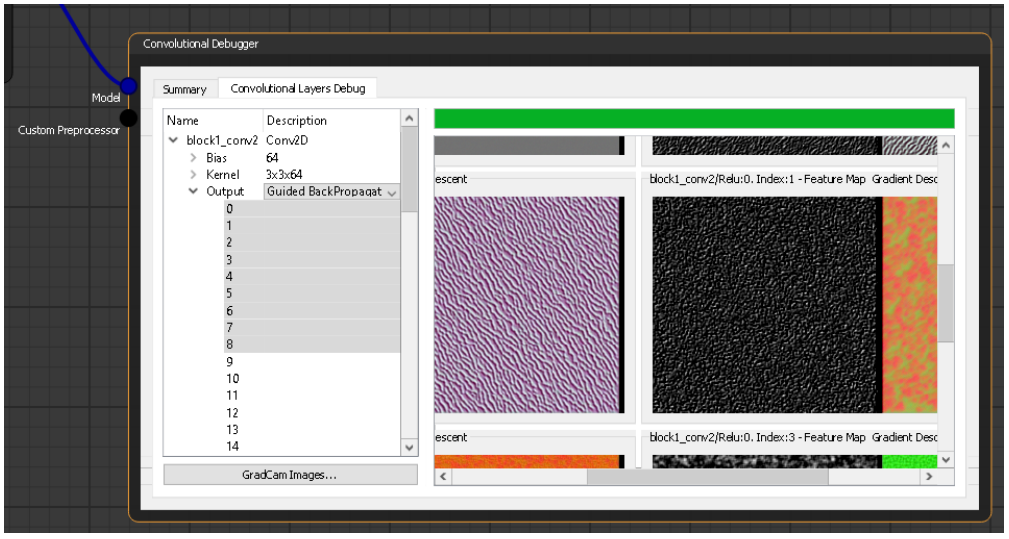


Figura 1.2 : Ejemplo de la segunda vista del nodo Dial-Visualization

1.3 Estado del Arte

Antes del desarrollo, se llevó a cabo una fase de investigación para conocer algunas de las herramientas ya existentes que buscan, mediante una interfaz y/o la implementación de diversas técnicas, ayudar al desarrollador a entender la razón de la clasificación final de redes neuronales convolucionales.

1.3.1 TensorWatch

TensorWatch [10] es una herramienta de depuración y visualización diseñada para la ciencia de datos, el aprendizaje profundo y el aprendizaje por refuerzo desarrollado por la división de *Microsoft*™ dedicada a la investigación, *Microsoft*™ *Research* [11] . Es una herramienta que funciona en los *cuadernos Jupyter* [12] para mostrar, en tiempo real, visualizaciones del proceso de una red neuronal en entrenamiento aunque también implementa diversas tareas claves para el análisis del modelo y la información.

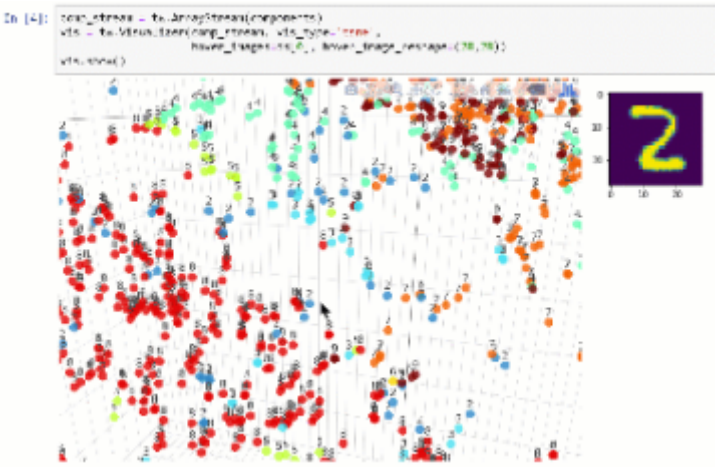


Figura 1.3: Visualización con TensorWatch del conjunto de datos en espacio dimensional inferior mediante técnica t-SNE

TensorWatch es una herramienta bastante completa que, además de implementar técnicas para las tareas mencionadas anteriormente, incorpora varias para visualizar y explicar neuronas o predicciones realizadas por las redes neuronales, como son: *Lime* [13] , *Gradient CAM, Guided Backpropagation, Deep Lift,Smooth Gradient,* entre otras.

1.3.2 CNN Explainer

CNN Explainer [14] es una herramienta interactiva desarrollada para ayudar a los desarrolladores iniciados a entender, de forma totalmente visual, el proceso de *feedforward* de una red convolucional, mostrando cómo se realiza y calcula la convolución de la imagen de entrada con cada kernel de cada neurona de la capa, así como mostrar el proceso del cálculo de resultados de las operaciones ReLu, Max Pooling y, finalmente, SoftMax .

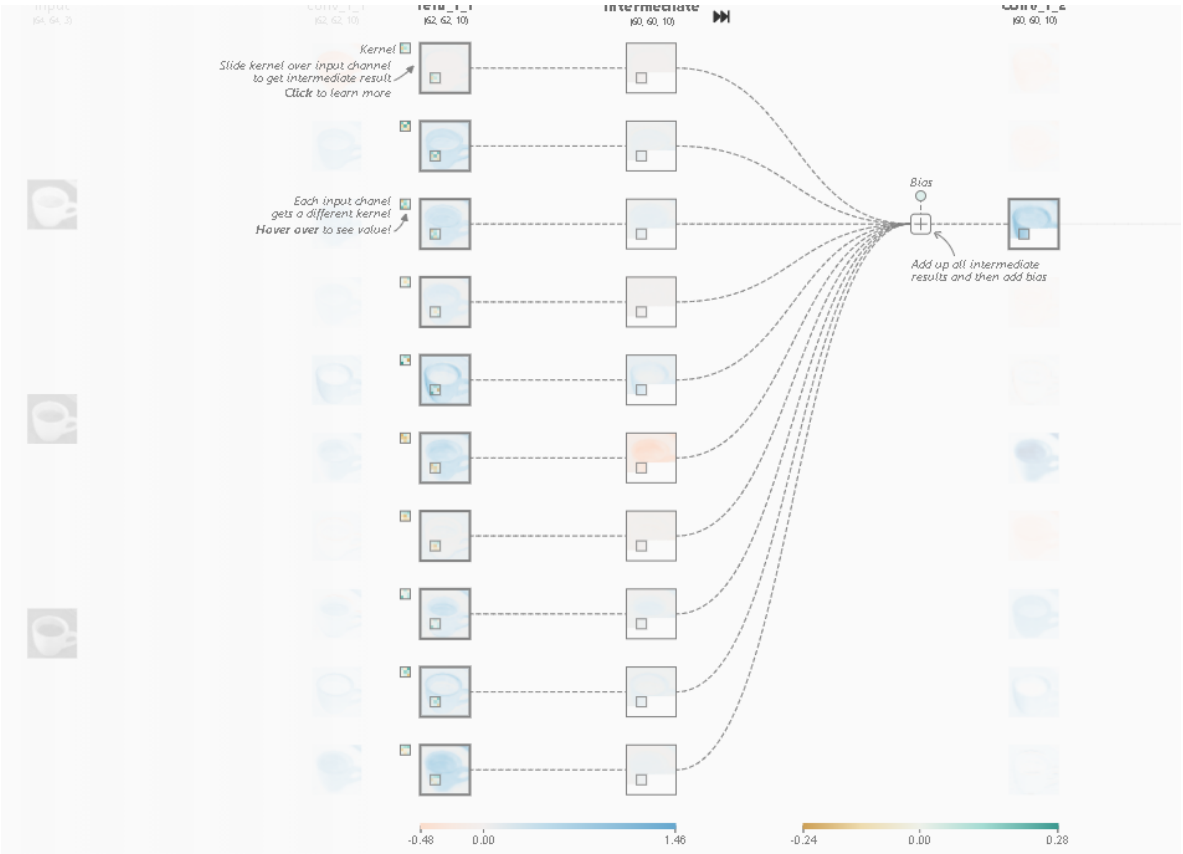


Figura 1.4: Demo de la herramienta CNN Explainer

Esta herramienta no busca explicar la decisión final, ni siquiera busca hacer un seguimiento del proceso de aprendizaje, así que en realidad no puede considerarse como un antecedente al trabajo desarrollado. Sin embargo, es una herramienta de aprendizaje a tener en cuenta, sobretodo por la forma tan cómoda, sencilla y elegante que tiene de, no solo presentar un elemento tan complejo como es una red de neuronas convolucionales, sino también por la forma de mostrar visualmente la explicación del proceso matemático interno que puede servir de inspiración.

Capítulo 2

Tecnologías

2.1 Tecnologías utilizadas en el desarrollo

En este apartado se recogerán las tecnologías (librerías, lenguajes de programación, frameworks y dependencias) utilizadas en el desarrollo del proyecto, así como las ventajas y funcionalidades que aportan.

2.1.1 Python

Figura 2.1 : Logo de Python

Python es un lenguaje de programación interpretado, dinámico, multiplataforma y multiparadigma, ya que soporta ,parcialmente la programación orientada objetos, la programación imperativa y, en menor medida, la programación funcional.

Este ha sido el lenguaje de programación utilizado para el desarrollo de este proyecto TFG.Las razones principales para usar este lenguaje de programación han sido:

- La aplicación sobre la que se monta este proyecto (Dial) se encuentra desarrollada en Python.
- Es un lenguaje bastante popular hoy en día por lo cual en Internet es posible encontrar abundante documentación e información para solventar casi cualquier problema que ocurra durante el desarrollo.
- *TensorFlow* y *Keras*. Aunque la librería TensorFlow también se encuentra escrita en otros lenguajes (C++, Android(Java) y Javascript), Keras solo se encuentra disponible en Python, bien como parte del core de TensorFlow o bien en su versión standalone.

También a consecuencia de lo anterior, se ha utilizado la versión 3.8, por ser la versión más estable y la última sobre la que la librería TensorFlow v2.5 puede funcionar.

Todo lo anterior tiene como consecuencia que el proyecto no pueda funcionar en la versión 2 de Python y sea extremadamente dependiente de las versiones en las que se desarrolló, no se garantiza en ningún momento que la aplicación funcione correctamente en versiones diferentes.

2.1.2 Git, Github

Figura 2.2: Logotipos de Git y Github

Git [18] es un software de control de versiones, gratis ,de código abierto y fácil de usar, bien desde línea de comandos o desde alguna de las diversas interfaces desarrolladas. GitHub [19] es una plataforma de desarrollo colaborativo casi exclusivo para alojar repositorios GIT de forma remota aunque con el tiempo ha ido ampliando sus funcionalidades y hoy también ofrece servicios de alojamiento web, herramientas de trabajo colaborativo, …

Por lo tanto, Git es , a día de hoy, una herramienta inseparable del desarrollador , como lo puede ser los IDE (*Integrated Development Environment,* por sus siglas en inglés) ya que permite, fácilmente, controlar las versiones de nuestro proyecto y en muchas ocasiones, los propios IDE traen GIT implementado o admiten ser configurados para acabar integrando el gestor de versiones en ellos. Y por eso se utiliza. Además, al ser una herramienta esencial para el desarrollo, existe muchísima documentación por Internet que permite conocer al instante como ejecutar casi cualquier operación de control de versiones que se quiera realizar.

GitHub otorga al desarrollador un servicio totalmente gratuito, con ciertas restricciones que no suelen afectar al desarrollador de código abierto, de alojamiento para poder subir el repositorio local a la nube, de forma que se pueda recuperar en cualquier momento y en cualquier lado, y esta son las razones principales por la cual ha sido usado en este proyecto.

Además, GitHub también incorpora características, como los *Issues* (tickets donde registrar tareas a realizar, nuevas características, bugs encontrados...) o los *Actions (*herramienta que permite instalar y configurar la ejecución de diversos scripts para la

gestión automatizada de los repositorios) que son útiles por sí solos para el control de versiones.

2.1.3 Qt Designer

Figura 2.3: Logo de Qt Designer

Qt Designer [33] es la herramienta oficial de QT para diseñar y construir interfaces de usuario gráficas(GUIs por sus siglas en inglés) mediante el arrastrar y soltar widgets desarrollados por QT.

Esta herramienta destaca por su facilidad de uso y su potencial. Es una herramienta que vengo utilizando desde que empecé a trabajar con QT, originalmente buscando por alguna alternativa a Tkinter en Python que admitiese programar interfaces desde algún lenguaje de marcas, y que me ha permitido saltarme el largo y tedioso proceso de prueba y error que supone programar o codificar el diseño de la interfaz. Con esta herramienta he podido ignorar ese proceso por completo y centrarme directamente en su parte lógica, lo cual, además, me permite tener un código mucho más limpio.

2.1.4 Dial-App

Figura 2.4: Logo de Dial-App

Durante el curso 2019-2020, el alumno de la Universidad de La Laguna, David Afonso [20], desarrolló en su TFG [21] la herramienta *"Dial-App"* [22] , un framework basado en Keras y con una interfaz para la programación visual utilizando nodos, con el objetivo de agilizar la aplicación de técnicas de Deep Learning a problemas de diversa índole. A través de una interfaz gráfica, el usuario puede interactuar con esta herramienta para construir y entrenar las redes sin necesidad de programar, lo que permite a usuarios sin conocimientos de programación poder usar estos modelos con sus propios datos, aumentando enormemente la aplicabilidad de estas librerías.

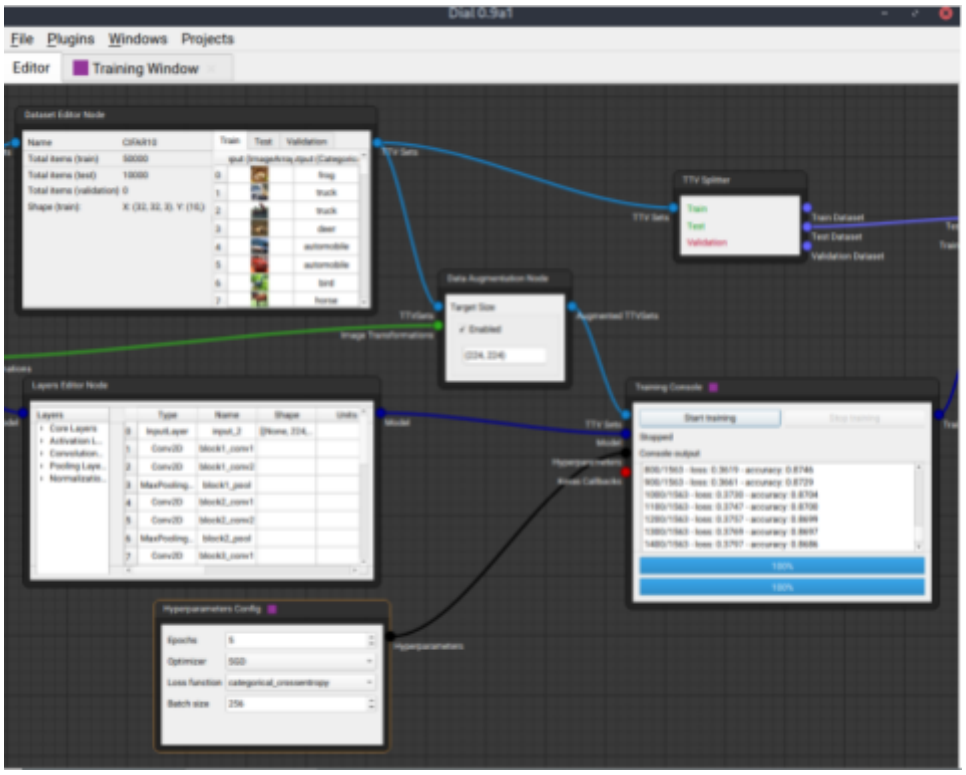


Figura 2.5: Ejemplo de la interfaz de Dial-App

La aplicación Dial-App viene dividida en 3 módulos principales o necesarios para su funcionamiento más básico. Estos módulos son los listados a continuación:

- **Dial-Core** . Implementa toda la lógica de grafos direccionados.
- **Dial-GUI** . Visualiza en una interfaz la lógica de grafos anterior, de forma que despliega el entorno necesario para la programación por nodos.
- **Dial-Basic-Nodes**. Implementa varios nodos con la finalidad de permitir el desarrollo, modificación, entrenamiento o uso de redes neuronales convolucionales.

2.2 Dependencias

Dado las tecnologías elegidas por el desarrollador inicial de Dial-App, la decisión de usarlas, para la mayoría, no ha sido propia sino impuesta por el entorno en el que se quería trabajar. Sin embargo, varias de ellas podrían ser seleccionadas por decisión propia dado que, en mi opinión, no solo son las más fáciles de usar, también las más potentes.

2.2.1 TensorFlow y Keras

Figura 2.6 : Logos de Keras y TensorFlow

TensorFlow es una plataforma de código abierto de extremo a extremo, con diversas funcionalidades e implementaciones orientadas a la programación para el aprendizaje automático mientras que Keras es una librería de código abierto ,amigable para el usuario, modular y extensible especialmente orientada a facilitar el diseño y desarrollo de redes neuronales y por ello que trabaja como una API que funciona como capa de abstracción del core de TensorFlow. Además, Keras forma parte de TensorFlow desde la publicación de la versión 2.0 de este.

Fué también la decisión del desarrollador de Dial-App para trabajar con redes neuronales convolucionales y aprendizaje automático y es por eso que estas dos librerías han tenido que ser usadas.Personalmente agradezco que así sea, porque Keras permite un nivel de abstracción lo suficientemente alto como para que nunca haya que trabajar con TensorFlow directamente.

2.2.2 Dependency-Injector

Figura 2.7: Logotipo de Dependency Injector

Dependency Injector [24] es un framework para Python que sirve para implementar el patrón de diseño basado en *inyección de dependencias* [25], un patrón de diseño por el cual dejan de ser las propias clases las responsables de crear instancias de sí misma, y

delegan esa funcionalidad a un contenedor u otra clase que será la especializada en crear las instancias de esa y otras clases.

Originalmente , el desarrollador de Dial-App emplea este diseño, y para no romperlo, he decidido continuar aplicándolo en mi propio proyecto TFG: Dial-Visualization.

2.2.3 Qt5 y PySide

Figura 2.8: Logos de QT y PySide

QT es un framework multiplataforma orientado a objetos ampliamente usado para desarrollar programas que utilicen interfaz gráfica de usuario, así como también diferentes tipos de herramientas para la línea de comandos y consolas para servidores que no necesitan una interfaz gráfica de usuario. Originalmente era un framework con soporte para solo C++ y QML(JavaScript) [26], hasta que en 2018 anunció que daba soporte a PySide2 [27] , ampliando su soporte a Python también.

PySide, en cualquiera de sus versiones, es una biblioteca para Python que hace de binding para las herramientas de interfaz gráfica de usuario de Qt y es junto a PyQT [28] una de las dos alternativas para desarrollar y programar interfaces QT en Python.

Aunque el desarrollo de la interfaz usando el framework QT ha sido una imposición a causa de la elección inicial del desarrollador de Dial-App, que ha desarrollado toda la interfaz usando esta herramienta, mi elección hubiera sido la misma pues también contaba con experiencia trabajando con este framework antes de empezar el desarrollo de mi TFG y es una herramienta que implementa muchos de sus controles bajo el popular patrón MVC [30]

También la elección de PySide2 sobre su alternativa, PyQT, ha sido debido al anuncio de que el primero está bajo soporte de la propia empresa que desarrolla el framework QT, *QT Company* [29], y la esperanza de que ello beneficiará a mi proyecto para poder seguir ampliandolo a futuro a largo plazo sin temor a que la herramienta deje de recibir soporte y a mi mismo, pues me estaré acostumbrando a trabajar con una biblioteca popular sobre la que más experiencia demandarán en el mercado de trabajo.

Capítulo 3

Arquitectura y diseño

En este capítulo se hablará del diseño general de la aplicación, y de cómo interaccionan las diferentes clases que definen la interfaz de *Dial-Visualization*. A continuación, se explicarán las clases responsables de aplicar las diferentes técnicas de visualización de *CNN*s implementadas y cómo es el flujo por el cual son ejecutados y cargados en la interfaz.

3.1 La interfaz de ConvVisualizationWidget

Esta es la clase que puede considerarse como el núcleo del proyecto y que es el responsable de realizar todas las operaciones que tengan que ver con la interfaz, en la figura 3.1 se detallan los atributos, métodos y relaciones de **ConvVisualizationWidget** con el resto.

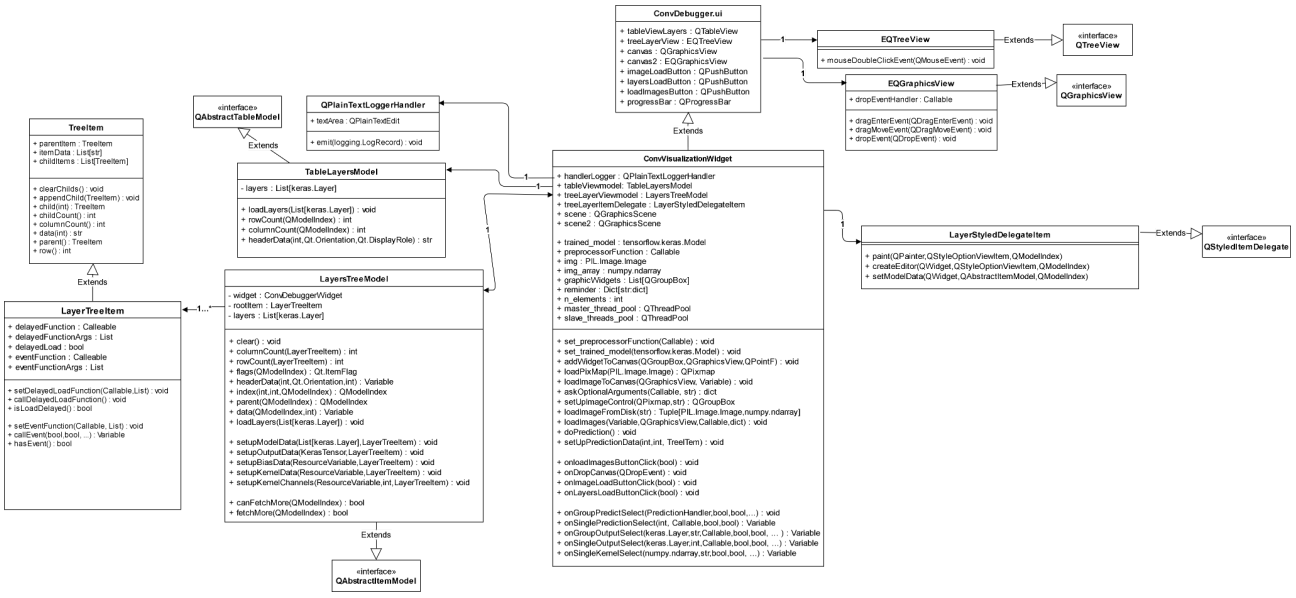


Figura 3.1: Diagrama de clases de interfaz de Dial-Visualization(click para enlace con zoom)

Toda la estructura del diagrama ha venido altamente condicionada por la propia forma de trabajo que QT demanda. El framework QT trabaja utilizando el patrón MVC [31] (Modelo-Vista-Controlador por sus siglas) y eso se puede ver en muchas de las clases presentes en el diagrama, aunque QT les da otro nombre, Model-View-Delegate, y suelen

designar el rol de una clase por su terminación(p. ej. *QTableView* es una vista) .

Normalmente no es necesario, para cada elemento vista (p. ej. *QTreeView*), crearle su correspondiente clase *modelo* y *controlador* ya que, por defecto, QT instancia en el constructor de la vista un controlador para poder tener una funcionalidad mínima, y para rellenar de datos existen clases como *QStandardItemModel*, para cargar estructuras de datos sencillas. Sin embargo, no es lo recomendable, pues su uso tiene un impacto directo en la escalabilidad y eficiencia computacional de nuestro proyecto.

En este proyecto se ha seguido la ruta recomendada: la implementación del patrón MVC. La implementación de una clase modelo de la vista *QTableView* consiste en heredar *QAbstractTableModel* (para otras vistas, consultar documentación [32]) y sobreescribir ciertos métodos que serán usados por el controlador para entender cómo debe cargar los datos y cómo y dónde debe visualizarlos. Una explicación de qué métodos deben ser sobreescritos y para qué sirven se puede ver en el Listado 3.1.

Listado 3.1: Ejemplo caso más básico de implementación de una clase modelo para una vista QTableView

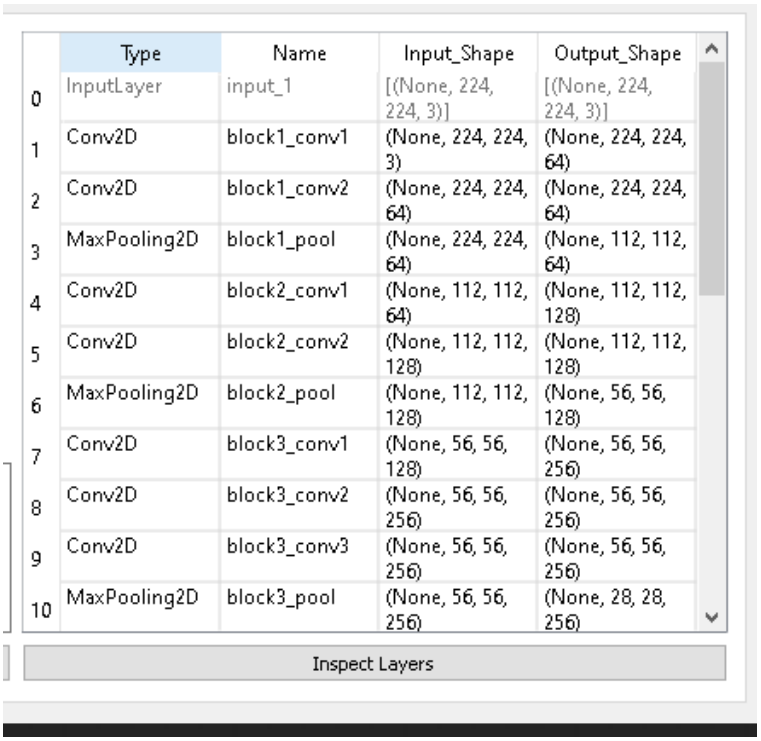
```
class TableModelExample(PySide2.QtCore.QAbstractTableModel):
    #Los Enum en Python son una estructura bastante conveniente en la que
guardar la representación de la cabecera, pero otras formas pueden usarse.
    class Column(IntEnum): #from enum import IntEnum
        NameCol1 = 0
        NameCol2 = 1
        NameColN = 2

    def __init__(self, parent:PySide2.QtCore.QObject = None):
        super().__init__(parent)
        self.data : List[Str] = []

    def setData(self,data: List[str]) -> None:
        #El modelo consultará 'self.data' para decir a la vista que debe
        mostrar.
        self.data = data
        #Por defecto, la vista no pregunta por datos que considera ya
cargados
        y mostrados,lo siguiente fuerza a que lo haga.
        self.modelReset.emit()

    def rowCount(self,parent=PySide2.QtCore.QModelIndex()) -> int: #overrided
        #Sirve al controlador para saber cuantas filas le falta por cargar
        return len(self.data)

    def columnCount(self,parent=PySide2.QtCore.QModelIndex())->int:
#overrided
```

```
        #Sirve al controlador para saber cuantas columnas debe cargar
        return len(self.Column);


    def headerData(self, section:int, orientation:QtCore.Qt.Orientation,
role:
    int = QtCore.Qt.DisplayRole)->str: #overrided
        #Solo celdas en formato texto. Mas info:Qt Namespace | Qt Core 5.15.4
        if role != QtCore.Qt.DisplayRole: return None
        #Si se habla de la cabeza horizontal
        if orientation == QtCore.Qt.Horizontal:
            return str(self.Column(section).name)
        #Si se habla de la cabeza vertical(enumeración filas):
        if orientation == QtCore.Qt.Vertical: return str(section)
        return None

    def flags(self, index:QtCore.QModelIndex)->QtCore.Qt.ItemFlag: #overrided
        #Aquí se asignan flags a las celdas según los datos que almacenan
         cumplan unas condiciones u otras. Flags disponibles:
         Qt Namespace | Qt Core 5.15.4
        if not index.isValid(): return QtCore.Qt.NoItemFlags
        row = index.row()
        value = self.data[row]
        flags = QtCore.Qt.NoItemFlags
        if 'a' not in value[0]:#Si cumple, será un elemento habilitado
            flags = QtCore.Qt.ItemIsEnabled
        return flags

    def data(self,index:QtCore.QModelIndex,role:int=Qt.DisplayRole )
            ->Optional[Any]: #overrided
        #Este metodo es multiproposito y se utiliza para resolver cada uno de
        los posibles roles que pueden asignarse aquí:
          Qt Namespace | Qt Core 5.15.4
        #En este ejemplo, sirve para responder con el contenido de la celda
en
        la fila 'row' y la columna 'col'.
        if role == QtCore.Qt.DisplayRole:
            row = index.row()
            col = index.column()
            value = self.data[row][col]

            if col == self.Column.NameCol1: return str(value)
            elif col == self.Column.NameCol2 : return str(value)
            elif col == self.Column.NameColN : return 'Cadena constante'
            return None
        #O sirve para responder como debe desplegarse el texto en la vista.
```

```
        if role == QtCore.Qt.TextAlignmentRole:
            return (QtCore.Qt.AlignRight | QtCore.Qt.AlignVCenter)
        return None
```

*Los métodos a sobreescribir están marcados con **#overrided***

El siguiente paso es crear una instancia de **TableModelExample**, pasarle un conjunto de datos (de dimensiones (-1,2)) llamando al método *"setData"* y añadirlo a un *QTableView* llamando a su método *"setModel"* . Obteniéndose un resultado semejante al de la Figura 3.2.



	Type	Name	Input_Shape	Output_Shape
0	InputLayer	input_1	[(None, 224, 224, 3)]	[(None, 224, 224, 3)]
1	Conv2D	block1_conv1	(None, 224, 224, 3)	(None, 224, 224, 64)
2	Conv2D	block1_conv2	(None, 224, 224, 64)	(None, 224, 224, 64)
3	MaxPooling2D	block1_pool	(None, 224, 224, 64)	(None, 112, 112, 64)
4	Conv2D	block2_conv1	(None, 112, 112, 64)	(None, 112, 112, 128)
5	Conv2D	block2_conv2	(None, 112, 112, 128)	(None, 112, 112, 128)
6	MaxPooling2D	block2_pool	(None, 112, 112, 128)	(None, 56, 56, 128)
7	Conv2D	block3_conv1	(None, 56, 56, 128)	(None, 56, 56, 256)
8	Conv2D	block3_conv2	(None, 56, 56, 256)	(None, 56, 56, 256)
9	Conv2D	block3_conv3	(None, 56, 56, 256)	(None, 56, 56, 256)
10	MaxPooling2D	block3_pool	(None, 56, 56, 256)	(None, 28, 28, 256)

Inspect Layers

*Figura 3.2: Ejemplo de vista final de **QTableView** tras cargar datos con **TableLayersModel***

El ejemplo anterior es extrapolable a cualquier modelo para cualquier vista que se quiera implementar, cambiando unos pocos detalles. Una vez se ha explicado cómo desarrollar una clase modelo sencilla, toca el turno de como modificar un controlador o delegado.

Si los modelos se encargan de guardar y gestionar los datos que la vista va a mostrar, los delegados son los encargados de contar a la vista cómo se van a dibujar. A continuación, en el Listado 3.2, se expone de forma resumida los métodos a sobreescribir y cuándo son llamados.

Listado 3.2: Ejemplo de implementación de clase delegado para QTreeView

```
class StyledDelegateItemExample(QStyledItemDelegate):
    #Cuando la celda/control no está en estado de edición, este
    método es constantemente llamado:
```

```
    def paint(self,painter:QPainter, option:QStyleOptionViewItem,
    index:QModelIndex): #overrided
        return super().paint(painter,option,index) #Comportamiento por
        defecto: Dibujar texto plano. Importante llamarlo o no se verá
        texto alguno.

    #Llamado en el instante en que la celda/control entra en estado
    edición.
    def createEditor(self,parent:QWidget ,option:QStyleOptionViewItem
    ,index:QModelIndex):#overrided
        return super().createEditor(parent,option,index)#Comportamiento por
        defecto. Nuevamente se llama para manejar los editores estándar
  #Llamado cada instante que se ejecuta una edición sobre el control
    def setEditorData(self, editor:QWidget,index:QModelIndex):#overrided
        return super().setEditorData(parent,option,index)
    #Llamado cuando se produce el evento que cierra el estado de
    edición. Utilizado normalmente para confirmar valor en el modelo
    def setModelData(self, editor:QWidget, model:QAbstractItemModel,
    index:QModelIndex):#overrided
        return super().setModelData(editor,model,index)
```

De la forma anterior, se implementó **LayerStyledDelegateItem** para permitir en una vista *QTreeView* mostrar listas desplegadas, tal y como se muestra en la figura 3.3.

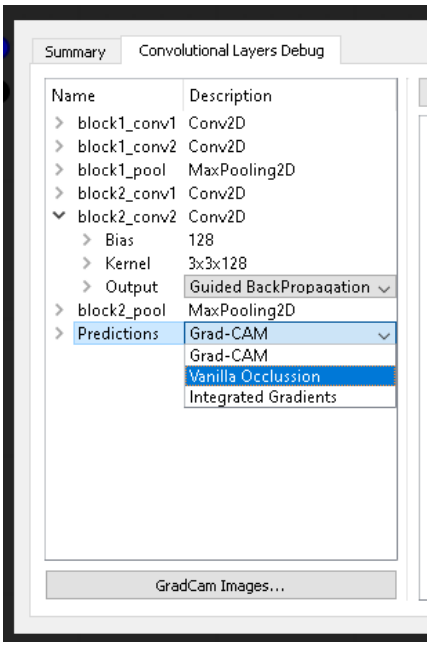


Figura 3.3: Resultado de implementar el delegado LayerStyledDelegateItem

Se ha explicado la utilidad de una clase modelo, una clase delegada, ¿Cuál es la utilidad de una clase vista? En principio, como su nombre indica, es la responsable de combinar el trabajo de las dos clases anteriores y visualizar el resultado, aunque su utilidad puede ir

más allá. En el caso de las clases **EQTreeView** y **EQGraphicView,** sirven para obtener un mayor acceso y control sobre la lista de eventos que maneja internamente**.** El primero sirve para obtener un mayor control sobre los eventos del ratón para poder priorizar o filtrar un tipo de acción sobre otro. El segundo sirve para dar soporte al evento de arrastrar elementos desde un *QTreeView* al lienzo que la clase **EQGraphicsView** describe como se ve en la figura 3.4.

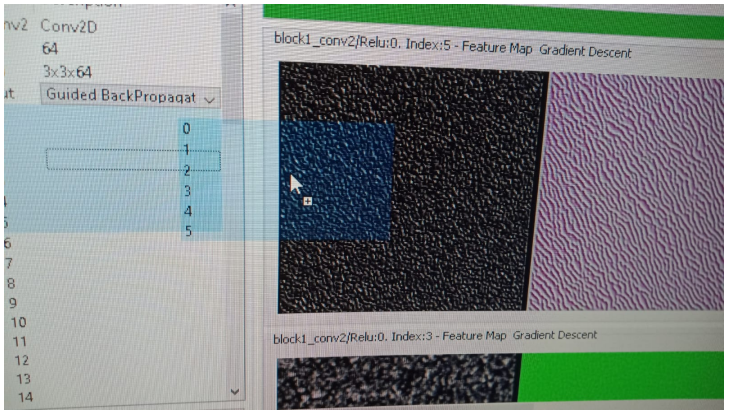


Figura 3.4: Soporte del evento drag sobre lienzo

Y para terminar de explicar las clases involucradas en la definición y comportamiento de la interfaz de **ConvVisualizationWidget**, solo quedaría hablar de **ConvVisualization.ui**. Este elemento no es en realidad una clase, sino que se trata de un documento XML creado con la herramienta *Qt Designer* mencionada en el capítulo de tecnologías empleadas. Este documento puede abrirse con la misma herramienta para visualizar o modificarla en cualquier momento y para cargarlo en Python con el fragmento de código que se muestra en el Listado 3.3:

Listado 3.3: Proceso de carga de ConvVisualizer.ui

```
import os
current_dir = os.path.dirname(os.path.abspath(__file__))
from PySide2.QtUiTools import loadUiType
Form, Base = loadUiType(os.path.join(current_dir,
"./ConvVisualizer.ui"))
class ConvVisualizationWidget(Form, Base):
    def __init__(self, parent: PySide2.QtWidgets.QWidget = None):
        super(self.__class__, self).__init__(parent)
        self.setupUi(self)
```

Nota:ConvVisualizer.ui y ConvVisualizationWidget para el ejemplo se encuentran en el mismo directorio

Explicado el rol de casi todas las clases mencionadas en el diagrama de la figura 3.1, restan por explicar dos.

LayerTreeItem extiende de **TreeItem** y, como su nombre indica, funciona como los ítems o nodos de un árbol. En este caso, es la estructura de datos elegida para almacenar y

gestionar los datos del modelo en **LayersTreeModel**. **TreeItem** no guarda nada especial, tiene un atributo para guardar los datos (*itemData*), otro para tener un enlace al nodo padre (*parentItem*) y un último para referenciar a sus nodos hijos (*childItems*). Esta clase, entonces, define la estructura de datos básica que sirve para definir un árbol. **LayerTreeItem** es una clase *ad-hoc* que guarda y gestiona los eventos que se ejecuten a nivel vista sobre la interfaz, de forma que sea el propio nodo el responsable de informar cuál es el método que debe ejecutarse en respuesta a un evento, tal y como se describe en la figura 3.5.

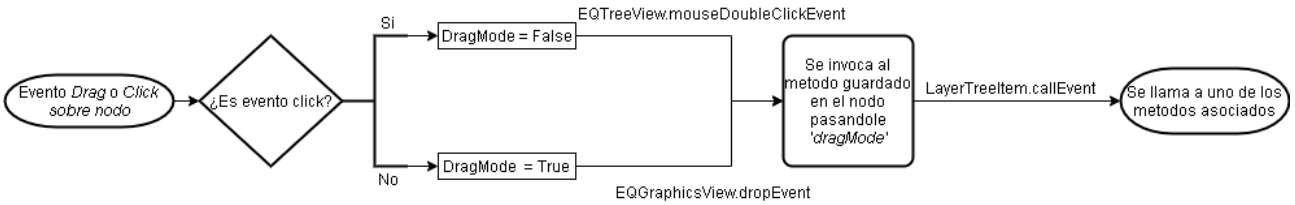


Figura 3.5. Diagrama de la secuencia de ejecución y manejo de un evento

También incorpora atributos y métodos para la carga diferida (*lazy load*) en la clase **EQTreeView** (explotando los métodos sobreescritos *canFecthMore* y *fecthMore* en el modelo). Para más detalles, consultar los comentarios en el repositorio con el código [34].

3.2 La ejecución de ConvVisualizationWidget

Anteriormente se ha hablado de todas las clases relacionadas con la definición y control de eventos de la interfaz, pero no se ha entrado en cómo es el proceso de obtención de los resultados de ejecutar las técnicas de visualización implementadas.

Antes de proseguir conviene declarar que, conceptualmente, hay tres tipos de ítems o nodos clasificados por la información que manejan o muestran:

- Los nodos Kernel, subdivididos, a su vez, en dos: padre e hijos. Son los nodos responsables de mostrar los kernels o matriz de pesos asociados a cada neurona de la capa. Solo el hijo tiene eventos asociados.
- Los nodos Output o Activación, también padre e hijo. El padre muestra información de las dimensiones de la salida de la neurona y el hijo solo lista cada uno de los nodos con salidas asociados a la capa. Los dos tienen eventos asignados.
- Los nodos Predicción padre e hijo. Son los responsables de visualizar, en lista ordenada de mayor a menor, la clase y el valor de su predicción realizadas por el modelo con la imagen cargada. También ambos tienen un evento asociado.

Cada uno de estos nodos tiene los siguientes métodos asociados de la clase **ConvVisualizationWidget** :

- Los nodos Kernel hijos tienen asociados el método *onSingleKernelSelect*.
- Los nodos Output padre tienen el método *onGroupOutputSelect* y los hijos *onSingleOutputSelect*.
- Los nodos Predicción tienen el método *onGroupPredictSelect* para el padre y

onSinglePredictionSelect para el hijo.

Estos métodos son los mismos que se han referenciado indirectamente llamándolos *"métodos asociados"* en la figura 3.5. En cada uno de estos métodos se crea una instancia de un controlador o *Handler* con el mismo prefijo que indica el tipo de nodo para el que está diseñado, estos controladores son los mostrados en la figura 3.6.

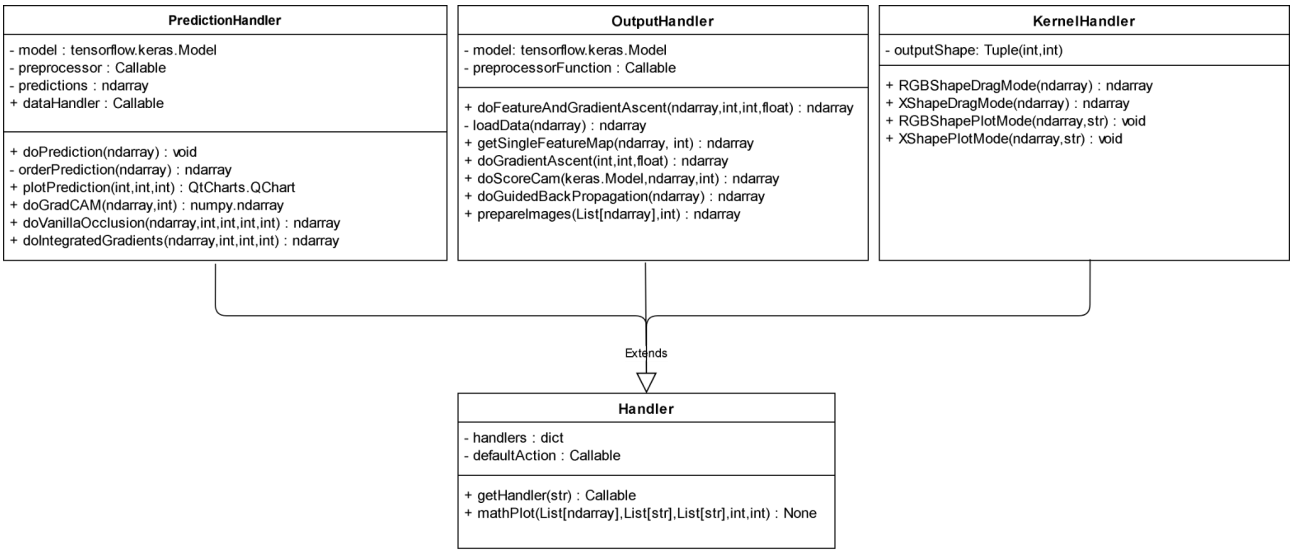


Figura 3.6: Diagrama de clase. Clase Handlers e hijas(click para enlace con zoom)

Y son los que se les pasan los parámetros y, en función de ellos, ejecutan un algoritmo u otro. Se inspecciona el método *__call__* de uno de ellos en el listado 3.4:

Listado 3.4. Ejemplo de manejo eventos del handler *PredictionHandler*

```
def __call__(self,**kwargs):
    who,dragMode=kwargs["who"],kwargs["dragMode"]
    #Proceso:
    if who in self._handlers:
        img_array = kwargs["img_array"]
        predictionIndex = kwargs["predictionIndex"]
        heatMap = None
        if who == "Grad-CAM":
            heatMap = PredictionHandler.doGradCAM(self,img_array,
                                                  predictionIndex)
        elif who == "Vanilla Occlussion":
            occluding_size,occluding_pixel,occluding_stride =
kwargs["occluding_size"],kwargs["occluding_pixel"],kwargs["occluding_st
ride"]
            heatMap = PredictionHandler.doVanillaOcclusion(self,
            img_array, predictionIndex, occluding_size=occluding_size,
            occluding_pixel=occluding_pixel,
```

```
            occluding_stride=occluding_stride)
        elif who =="Integrated Gradients":
            num_steps,batch_size = kwargs["num_steps"],
                                   kwargs["batch_size"]
            heatMap = PredictionHandler.doIntegratedGradients(self,
                      img_array, predictionIndex, num_steps=num_steps,
                      batch_size=batch_size)
        else: #Custom one
            kwargs["self"] = self
            kwargs["model"] = self.__model
            kwargs["preprocessorFunction"] = self.__preprocessorFunction
            return self._handlers[who](**kwargs)
        who = "Prediction Index: {} - {}".format(predictionIndex,who)
        if dragMode:
            return heatMap,who
        else:
            self.mathPlot([heatMap],[who],[None],1,1)
        return
```

Se puede ver que el procedimiento *__call__* lo que hace es consultar las claves de un diccionario, *'handlers'*, y ejecuta el método asociado en función del valor de *'who'*. Nótese que se considera que es un nuevo algoritmo si la clave no es alguno de los ya codificados.

¿Cómo se implementa un nuevo método? Aunque la interfaz no incorpora métodos para añadir nuevos algoritmos de procesamiento desde otro nodo, el código en sí está preparado para que el trabajo a realizar sea mínimo. El proceso es bastante sencillo, solo hay que modificar la clase **HandlerConfiguration** en *Handler.py* y añadir una nueva entrada al diccionario que ahí se encuentra. Por ejemplo, si se quiere añadir el algoritmo *dummy* que se muestra en el listado 3.5.

Listado 3.5. Ejemplo de algoritmo de procesamiento dummy

```
class Test:
    @staticmethod
    def dummy(**kwargs):
        return kwargs["img_array"][0].astype(np.uint8),"Dummy"
```

Nótese que debe devolver dos valores, un array de enteros sin signo 8-bits con dimensiones W(idth)xH(eight)x3 ,siendo W y H enteros >0, y un string que será el título. Además, debe ser accesible desde fuera en todo momento (por ello es un método estático).

Para añadirlo al listado de procesadores, por ejemplo, de los nodos Predicción, bastará con añadir una entrada en el atributo *handlers* de la clase **HandlerConfiguration** dentro de la clave *PredictionHandler,* quedándose, como resultado, el indicado en el listado 3.6:

Listado 3.6. Añadiendo una nueva entrada

```
...,PredictionHandler : {
    "Grad-CAM" : PredictionHandler.doGradCAM,
    "Vanilla Occlussion": PredictionHandler.doVanillaOcclusion,
    "Integrated Gradients": PredictionHandler.doIntegratedGradients,
    "Dummy" : Test.dummy #Nueva entrada
    },...
```

Al ejecutar el programa, se puede notar que la entrada se ha añadido automáticamente a la lista desplegable de los nodos *Predicción* para seleccionar el algoritmo de procesamiento, tal y como se muestra en la figura 3.6.

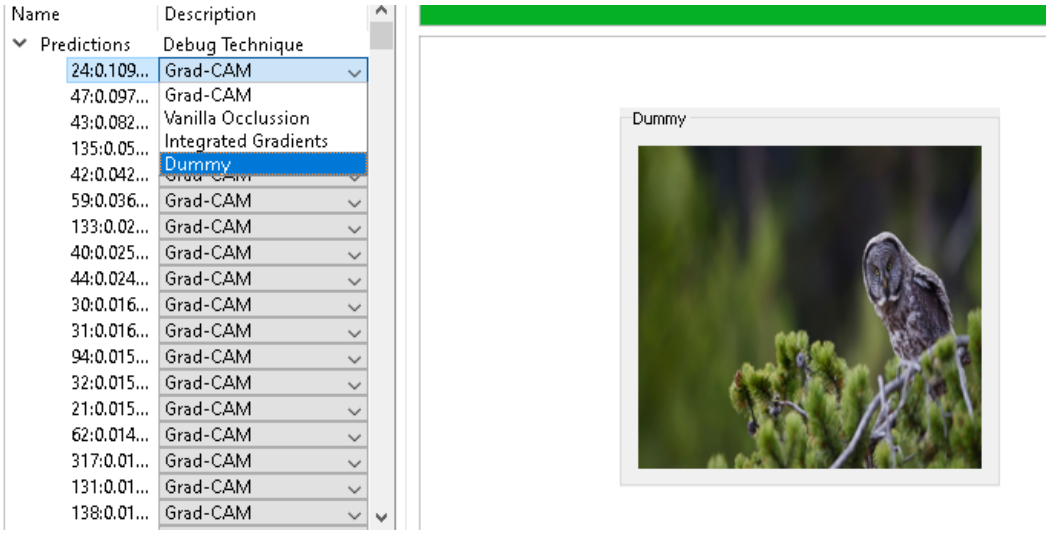


Figura 3.7: Resultado tras añadir entrada algoritmo 'Dummy'

Capítulo 4

Técnicas de visualización

En este capítulo se va a hacer una revisión de las técnicas implementadas, intentando explicar el concepto o fundamento matemático que buscan explotar si lo hubiese. Para agilizar la explicación, se supondrá que se tienen nociones básicas de la composición y funcionamiento de una red neuronal convolucional y que los conceptos *feedforward [14]*, *back-propagation [35]*, *operación de convolución* [36], *operación de pooling* [37] y *funciones de activación* [38] están asimilados.

También conviene definir el concepto *gradiente*. El gradiente, en este contexto, es mejor entendido como la generalización de la derivada para funciones de más de una variable. Y dado que las redes neuronales son funciones de no pocas variables, no solo por las dimensiones de la entrada, sino por el número y dimensiones de los propios pesos, es conveniente hacer esa distinción. En la práctica, el gradiente en una red neuronal se calcula como la derivada parcial de múltiples funciones compuestas explotando la regla de la cadena [39], de forma que la computación de estas en capas iniciales puede aprovecharse para el cálculo del gradiente en capas más profundas. Como ejemplo para entender lo anterior en profundidad, se sugiere la lectura del artículo en inglés redactado por Pierre Jaumier [35] .

4.1 Visualización de pesos

Esta técnica busca visualizar los pesos/kernels de los canales de una capa convolucional. La técnica para visualizarlos es sencilla y se puede distinguir dos casos:

- Cuando las dimensiones del kernel son *WxHx3*, normalmente, sólo posible en la primera capa convolucional que actúa sobre la imagen de entrada. Los pesos se muestran como una imagen RGB, pudiendo asociar los colores con los valores de dichos pesos.
- Cuando las dimensiones del kernel son *WxHxN* con *N != 3*. En este caso, se muestra cada canal individualmente en escala de grises, donde el negro significa que minimiza valor de salida y blanco que lo maximiza.

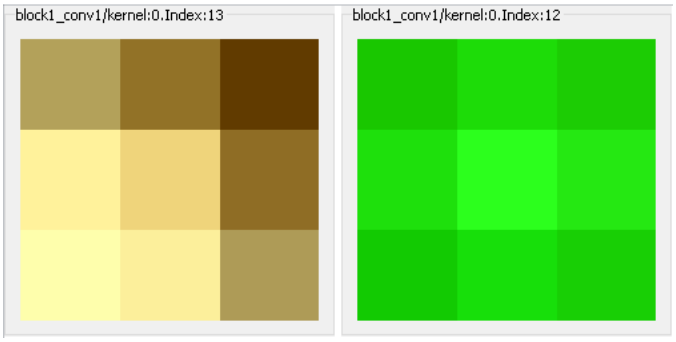


Figura 4.1. Ejemplo de visualización de Kernels de dimensiones 3x3x3

¿Qué utilidad tiene? Permiten entender la operación que la neurona realiza sobre la imagen (detección de bordes, cálculo de valores medios, maximización/minimización patrones de valores, …). Sin embargo, a medida que se profundiza en la red, dejan de ser útiles, desde un punto de vista intuitivo, para explicar la salida de las neuronas, dado el aumento de la complejidad de la operación al combinarse con anteriores realizadas durante el proceso de *feedforward*.

4.2 Visualización de activaciones

Este procedimiento busca visualizar las activaciones de las neuronas de una capa convolucional. Para visualizarlas, el procedimiento es también simple. Se crea un modelo con la misma arquitectura que el modelo en estudio pero seccionando, justo, la salida de la capa convolucional que se quiere estudiar. De esta forma, el procedimiento es directo y al pasarle una entrada y ejecutar *feedforward,* se obtiene en la salida el conjunto de mapas de activaciones o de características, uno por cada canal de la capa convolucional elegida, pudiendo posteriormente seleccionar y estudiar cada uno de estos mapas, por separado.

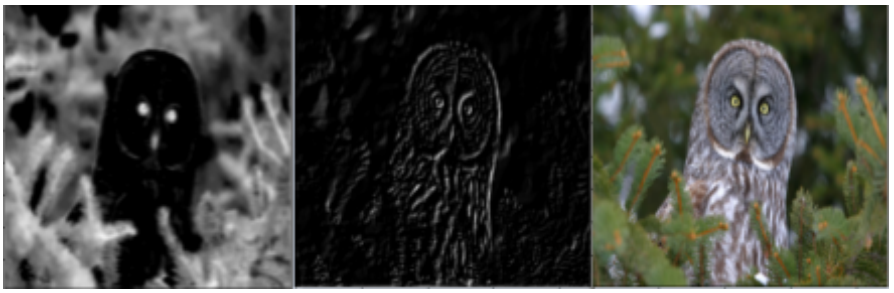


Figura 4.2: Ejemplo de mapa de activaciones(izq. y centro) junto a imagen original(der.)

En la figura 4.2 se puede observar como los valores maximizados por la capa resaltan en blanco frente a los valores que minimiza, en negro.

La utilidad de esta visualización reside en inferir la función de la neurona a partir de su resultado. Nuevamente, dada la reducción de dimensionalidad del resultado obtenido y el aumento de la complejidad de las operaciones combinadas realizadas sobre la imagen a

medida que profundiza en la red, su interpretación deja de ser intuitiva en capas profundas.

4.3 Visualización de filtros

Se requieren nuevas técnicas para descifrar la utilidad de una neurona en capas más profundas de la red. Una técnica implementada de visualización de filtros es la *maximización de las activaciones* mediante *ascenso del gradiente*. El algoritmo para implementar esta técnica fue propuesto por y Erhan et al. en 2009 [40] .

Este técnica parte del resultado recibido, producto del *feedforward,* para que, en el *backpropagation,* en lugar de modificar pesos que minimizan el error de la función coste, calcula el gradiente para ascender por él y modificar píxeles de la imagen original con el objetivo de obtener una imagen que maximice las activaciones de la neurona en estudio. Entonces, matemáticamente, lo anterior se puede definir como:

$$Max\ (f_{i,l}(\theta,\ x))$$

Donde:

- θ son los parámetros de la red neuronal convolucional (pesos y bias).Constantes
- x es la imagen usada como entrada de la red.
- $f_{i,l}$ es el mapa de activación resultante de la neurona i de la capa l.

Entonces, se presenta un problema de optimización cuya resolución es heurística y se puede describir de la siguiente manera:

1. Se crea una imagen entrada inicial x con valores aleatorios
2. Se calcula el gradiente del mapa de activación respecto a la entrada ($\frac{\delta f_{i,l}(\theta, x)}{\delta x}$)
3. Se eligen 2 parámetros, número de repeticiones (s) y ratio de aprendizaje (α)
4. Para maximizar la salida de la neurona, se realiza ascenso por el gradiente s veces, modificando la imagen inicial iterativamente de la siguiente forma:

$$x \leftarrow x + \alpha * \frac{\delta f_{i,l}(\theta, x)}{\delta x}$$

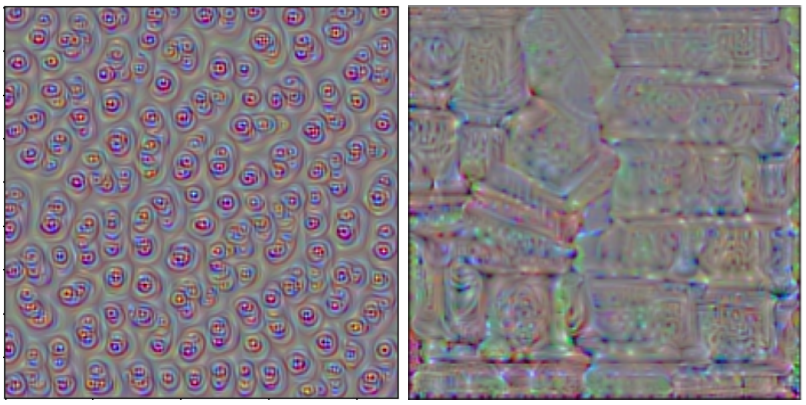


Figura 4.3.1: Ejemplo de imágenes para maximización de activaciones de una neurona

A diferencia de la técnica de visualización de activaciones, que depende de una imagen de entrada dada, esta técnica permite inferir, de una manera más global, el tipo de patrones a los que responden las neuronas de los canales de las diferentes capas de la red convolucional. En el ejemplo de la figura 4.3.1, la primera imagen por la izquierda, las neuronas de este canal parece que están entrenadas para detectar lo que parecen ser ojos, en la segunda imagen su interpretación es difícil.

Esta técnica presenta varios problemas. El primero, el patrón que detecta no siempre queda bien definido y a veces ni responde al ascenso del gradiente, y simplemente devuelve la imagen aleatoria original, no otorgando información relevante. El segundo, para capas muy profundas, el patrón que reconoce se vuelve irreal comparado con las imágenes del mundo real pasadas durante su entrenamiento y, por lo tanto, puede ser difícil de interpretar.

También conviene destacar que en capas superficiales, los patrones son más *“planos”* y no buscan maximizar formas sino colores, y cuanto más se profundiza en la red, ocurre lo contrario, y empiezan a aparecer patrones más definidos.

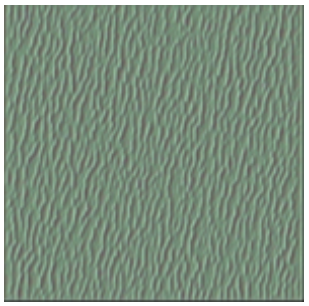


Figura 4.3.2: Ejemplo de filtro de nodo en capa superficial(VGG16:block1_conv2)

4.4 Mapas de atribuciones/saliency

Si las técnicas de visualización anteriores buscaban interpretar la operación de cada neurona convolucional, los mapas de atribuciones buscan indicar la importancia de cada pixel de la imagen de entrada en la predicción de la red. Existen varias técnicas para

obtener estos mapas de atribuciones, y dependiendo del método elegido, se pueden subdividir en 3 categorías.

4.4.1 Basados en gradientes

Estos métodos se basan en el *backpropagation* o propagación hacia atrás del gradiente de una clase con respecto a la imagen de entrada para resaltar aquellas partes de la misma que más influyen en la clasificación o predicción final de la red. La primera técnica documentada de cálculo de mapa de atribuciones basado en gradiente fue propuesta por Simonyan et al. en 2013 [41], técnica que fue mejorada en 2014 por Springenberg et al. [42] y conocida con el nombre de:

Guided Backpropagation

La técnica de Simonyan et al se basa en obtener el gradiente de la activación de una determinada neurona de la red respecto a la imagen de entrada:

$$\frac{\delta f_{i,l}(\theta, x)}{\delta x}$$

El problema de lo anterior es que , en el proceso del cálculo del gradiente, para capas que utilicen la función de activación ReLu:

$$ReLu(x) = \begin{Bmatrix} 0, & if \quad x < 0 \\ x, & otherwise \end{Bmatrix}$$

Su derivada es tal que:

$$\frac{\partial ReLu(x)}{\partial x} = \begin{Bmatrix} 0, & if \quad x < 0 \\ 1, & otherwise \end{Bmatrix}$$

Sigue manteniéndose en función de *'x'* y no del resultado del *backpropagation*, por lo tanto, los valores negativos computados en el gradiente de capas anteriores se siguen propagando hacia atrás, tal y como se muestra en la figura 4.4

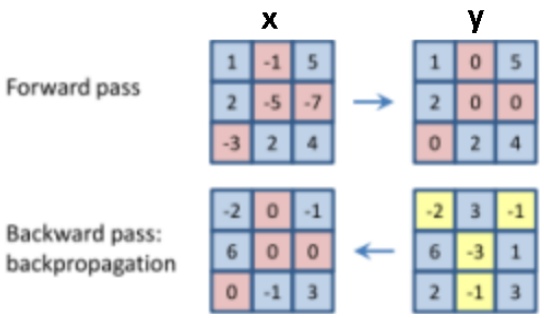


Figura 4.4. Ejemplo ReLu backpropagation

En la propagación hacia atrás, ReLu recuerda los valores de entrada iniciales y por lo tanto pone a 0 los valores negativos, no en función de los valores en *'y',* sino en función de los valores *'x'* original. Esto provoca que la imagen resultante, aunque otorgue información del área y características de la imagen en las que la red se ha fijado, aparezca demasiado ruido:

Figura 4.5: Resultado de aplicar Gradient Backpropagation(izq.) e imagen original(der.)

Entonces, la técnica propuesta por Springenberg et al., sugiere modificar artificialmente la derivada de la función ReLu para evitar la propagación hacia atrás de valores negativos:

$$\frac{\partial ReLu(x, \partial y)}{\partial x} = \begin{Bmatrix} 0, & if \quad x < 0 \quad or \quad \partial y < 0 \\ 1, & otherwise \end{Bmatrix}$$

De forma que en la propagación hacia atrás, los valores negativos dejan de propagarse como puede verse en la figura 4.6:

Figura 4.6: Ejemplo de propagación hacia atrás con derivada ReLu modificada

Con lo cual, se obtienen visualizaciones con mucho menos ruido y "de grano fino" como puede verse en la figura 4.7:

Figura 4.7: Resultado de aplicar Guided Backpropagation(izq.) e imagen original(der.)

Al final, ambas técnicas permiten conocer el área de píxeles de la imagen original que

logran viajar por la red convolucional y llegar hasta la capa estudiada y, por lo tanto, en capas profundas, conocer qué píxeles son los que influyen en la clasificación final. También permite visualizar la capacidad innata de las redes convolucionales de capturar y conservar la información espacial de las diferentes entidades presentes en una imagen.

La explotación del gradiente es la opción más popular para estas técnicas, dado el dinamismo que ofrece al poder aplicarlo a cualquier red sin importar mucho su arquitectura. Otra técnica basada en gradientes es **Integrated Gradients**.

Integrated Gradients

Propuesto por Mukund Sundararajan et al. en 2017 [47] , la idea de esta técnica consiste en ir modificando la imagen original de la siguiente forma:

$$x = x' + \alpha * (x_0 - x')$$

Donde:

- x imagen resultante que será usada como entrada del modelo.
- x' *'baseline'*, imagen en negro, o con valores aleatorios.
- x_0 imagen original.
- α constante de interpolación, valor en rango [0,1] para indicar el grado de perturbación.

Figura 4.8: Imagen inicial(izq.),baseline(centro), interpolación(der.)

Se genera un lote de tamaño *'N'*, de estas interpolaciones, variando la constante de interpolación (α) en cada una de ellas para obtener interpolaciones diferentes(x_i), posteriormente se calcula el gradiente:

$$\frac{\delta F(x_i)}{\delta x_i}$$

Donde:

- $F()$ es la función de predicción/clasificación de la red.
- x_i es la interpolación 'i' del lote de tamaño 'N'

Cada gradiente es sumado y posteriormente multiplicado por la imagen diferencia entre la imagen original y el baseline,

$$IntegratedGrad^{approx}(x_0) = (x_o - x') * \frac{1}{N}\sum_{i}^{N}(\frac{\delta F(x_i)}{\delta x_i})$$

Es la aproximación a la integral del gradiente, porque, como su nombre indica, la suma es originalmente una integración, sin embargo se realiza una aproximación a ella para reducir el elevado coste computacional que supone calcular una integral, siendo la fórmula original la siguiente:

$$IntegratedGrad(x_0) = (x_o - x') * \int_{\alpha=0}^{1}(\frac{\delta F(x' + \alpha_i * (x_0 - x'))}{\delta x})\delta\alpha$$

Figura 4.9: Mapa de atribuciones con Integrated Gradient(izq.) e imagen original(der.)

En la figura 4.9 se pueden ver los píxeles que más importan para la clasificación de la clase 24, siendo especialmente importantes los ojos y el contorno/forma de la cabeza. También se puede notar que el mapa incorpora ruido, posiblemente debido al error que se propaga en el gradiente de ReLu, como ya se ha descrito anteriormente, y que es la razón de existir de **Guided BackPropagation**.

4.4.2 Basados en CAM

Class Activation Mapping es una técnica de visualización propuesta por Zhou et al. en 2016 [43] con la intención de estudiar la importancia de los píxeles de una imagen de entrada en la predicción final de una clase de un modelo con función de activación softmax [44]. Sin embargo, **CAM** presenta el problema de no poder aplicarse para redes convolucionales que incorporen capas no convolucionales (p ej. VGG16). A raíz de esto, se desarrolla una alternativa más flexible, basada también en cálculo del gradiente,

nombrada **Grad-CAM**.

Grad-CAM

Esta técnica, propuesta por Selvaraju et al. en 2016 [46], es una generalización de **CAM** y, por lo tanto, también utiliza **GAP**(*global average pooling*) para calcular el promedio, solo que en esta ocasión, calcula el promedio de los gradientes de los *logits* [45] de la clase en estudio respecto a los valores de los mapas de activaciones con ancho *'w'* y altura *'h'* de la última capa convolucional *'k'* que aparece en la red. Expresado matemáticamente:

$$\alpha_k^c = \overbrace{\frac{1}{Z}\sum_i^{w}\sum_j^{h}}^{\text{global average pooling}} \underbrace{\frac{\partial y^c}{\partial A_{ij}^k}}_{\text{gradients via backprop}}$$

Donde:

- y^c es la salida del modelo para la clase *'c'* del modelo antes de SoftMax (*logits*).
- $A_{ij}^{\ k}$ es el valor del mapa de activación en la posición *'i','j'* para cada salida de las neuronas de la última capa convolucional *'k'*.
- $\alpha_k^{\ c}$ es el vector de pesos resultado con longitud igual al número de salidas/neuronas de la capa convolucional *'k'.*

Este vector de pesos recoge, conceptualmente, una puntuación que describe la importancia de cada mapa de activación en la predicción final de la clase *'c'* estudiada. Esto servirá para maximizar aquellos mapas con mayor puntuación y lo contrario:

$$\alpha_k^{\ c} * A^k$$

Estos mapas de activación ponderados resultantes se suman entre ellos para reducir la dimensionalidad y obtener un único mapa de activación con máximos en las regiones que de media estaban maximizados, y lo contrario.

$$cam = \sum_k (a_k^{\ c} * A^k)$$

El resultado se pasa por ReLu para eliminar aquellos valores que influyen negativamente en la predicción y que, en este contexto, representan ruido, y deben ser eliminados pues lo que se quiere visualizar son las regiones de píxeles que más influyen.

$$L^{c}_{Grad-CAM} = ReLu(\sum_{k}(a_{k}^{c} * A^{k}))$$

Obteniendo finalmente el mapa de atribuciones que se tendrá que procesar para moverlo al rango de valores admitidos por una imagen([0,255]) y redimensionar al tamaño de la misma original(*upsampling*) para visualizarla, si se quiere, como mapa de calor .

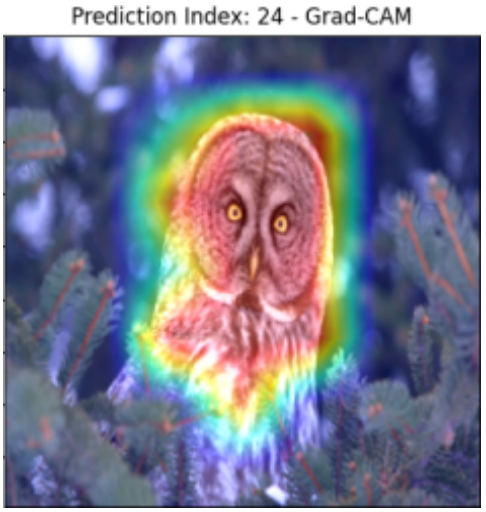


Figura 4.10: Ejemplo de Grad-CAM como mapa de calor sobre imagen original

4.4.3 Basados en perturbaciones

Una de las formas más intuitivas de evaluar un modelo y su comportamiento consiste en la perturbación de las características de una misma imagen fija, pasarla al modelo y evaluar los cambios en los resultados. Las técnicas basadas en perturbaciones explotan precisamente eso. Un ejemplo de técnica de este grupo es la **Oclusión.**

Técnica de oclusión

Esta técnica, más conocida como *Occlusion Sensitivity*, fue propuesta por Matthew D Zeiler y Rob Fergus en 2013 [49]. La idea del algoritmo es bastante intuitiva, se definen cuadrados negros (o variable) de tamaño *NxN* y se superponen sobre una imagen original para ir desplazandose por la imagen de izquierda a derecha y de arriba hacia abajo, moviendose *m* pixeles iterativamente:

Figura 4.11: Resumen proceso de oclusión con cuadrado tamaño 'N' y desplazamiento 'N'

Cada iteración en el proceso de oclusión genera una imagen ligeramente alterada respecto a la original que es pasada a la red para recuperar la predicción respecto a la clase estudiada y que, guardada en forma de matriz, devuelve una puntuación que describe la importancia del área ocluida por el cuadrado respecto a la clasificación.

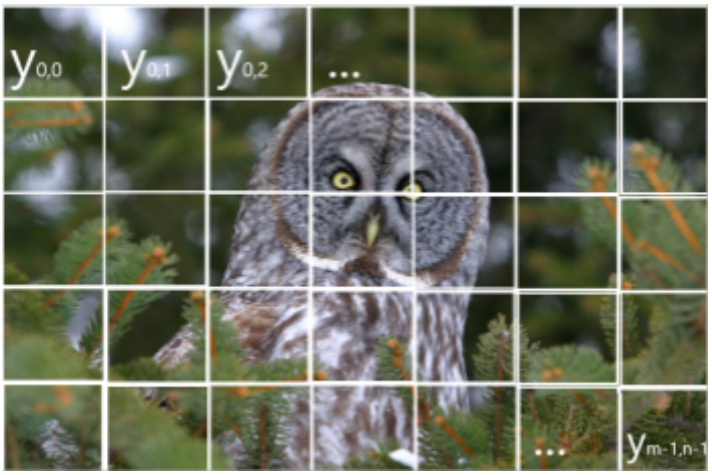


Figura 4.12: Ejemplo de matriz de puntuaciones.

De forma que la matriz redimensionada al tamaño de la imagen original y superpuesta sobre ella en forma de mapa de calor, describe el área de píxeles más determinantes para la clasificación de la clase en la red, como se puede observar en la figura 4.13.

Figura 4.13: Ejemplo de mapa de calor (usando función sigmoide para enfatizar) resultante de aplicar el método de oclusión a esta imagen

Capítulo 5 Conclusiones y líneas futuras

5.1 Conclusiones

En este trabajo se ha desarrollado un módulo para la visualización de filtros, pesos, activaciones y mapa de atribuciones de una red neuronal convolucional entrenada, integrando esto en la herramienta desarrollada por *David Afonso*, Dial-App. Esta aplicación tiene un enorme potencial para el desarrollo y uso de redes neuronales convolucionales, permitiendo integrar nuevos módulos como el desarrollado en este trabajo: Dial-Visualization. Un módulo con un alto nivel de complejidad, no solo por las técnicas de visualización implementadas, sino por la modificación de las clases de QT para lograr una interfaz usable.

Durante el desarrollo de Dial-Visualization, se encontraron algunos problemas, más bien por la inexperiencia propia en lo que respecta a trabajar con QT, a un grado donde se tienen que modificar funcionalidades propias de las clases implementadas por defecto. Sin embargo, conviene destacar la abundante documentación que los desarrolladores de QT ofrecen, siendo útil para solventar cualquier detalle o problema surgido durante la elaboración de este proyecto.

5.2 Desarrollo futuro

Existen muchas líneas de trabajo a futuro abiertas, algunas de ellas son:

- Desarrollar e implementar nuevas técnicas de visualización. Como se comentó anteriormente, existen muchas y cada día la lista se expande.
- Soporte desde interfaz para incluir nuevas técnicas de visualización. Aunque a nivel de código la aplicación está preparada, sería conveniente añadir un proceso para cargarlos desde la interfaz de Dial-App directamente.
- Expandir el soporte a otro tipo de redes neuronales. Actualmente la aplicación solo está preparada para funcionar con redes neuronales secuenciales convolucionales.
- Desarrollar soporte para visualizar secuencias de imágenes (gifs). Existen técnicas que aplican procesos iterativos, como la maximización de activaciones, en las que conviene visualizar los resultados de cada iteración en tiempo real, pues aportan mayor información cuando se muestran de esa forma.
- Implementar la traducción del proceso a cuadernos Jupyter. Dial-App incorpora una utilidad para que cada acción realizada sobre el nodo se traduzca a celdas con código Python típicas de un cuaderno Jupyter.
- Dividir Dial-Visualization en nodos más simples. Por la forma de diseñarlo, se

rompe la filosofía de la programación por nodos inicial, esto es, operaciones descompuestas en nodos que representan sus partes más sencillas.

Capítulo 6 Summary and Conclusions

6.1 Conclusions

In this work, a module has been developed for the visualization of filters, weights, activations and attributions maps of a trained convolutional neural network, integrating this in the tool developed by David Afonso, Dial-App. This application has enormous potential for the development and use of convolutional neural networks, allowing the integration of new modules such as the one developed in this work: Dial-Visualization. A module with a high level of complexity, not only due to the visualization techniques implemented, but also due to the modification of the QT classes to achieve a usable interface.

During the development of Dial-Visualization, some problems were encountered, more due to inexperience when it comes to working with QT, to a degree where functionalities of the classes implemented by default have to be modified. However, it is worth highlighting the abundant documentation that the QT developers offer, being useful to solve any detail or problem that has arisen during the development of this project.

6.2 Future development

There are many lines of future work open, some of them are:

- Develop and implement new visualization techniques. As mentioned above, there are many and every day the list expands.
- Support from the interface to include new visualization techniques. Although at the code level the application is prepared, it would be convenient to add a process to load them from the Dial-App interface directly.
- Expand support to other types of neural networks. Currently the application is only prepared to work with convolutional sequential neural networks.
- Develop support to view image sequences (gifs). There are techniques that apply iterative processes, such as maximization of activations, in which it is convenient to view the results of each iteration in real time, since they provide more information when they are shown in this way.
- Implement the translation of the process to Jupyter notebooks. Dial-App incorporates a utility so that each action performed on the node is translated into cells with Python code typical of a Jupyter notebook.
- Divide Dial-Visualization into simpler nodes. Due to the way it is designed, the philosophy of initial node programming is broken, that is, operations decomposed into nodes that represent its simplest parts.

Capítulo 7 Presupuesto

En este capítulo se presenta el presupuesto en función de las horas dedicadas.

7.1 Coste Material

Objeto	Valor (€)
Ordenador	1059,03 €

7.2 Coste Humano

En este capítulo se incluye un presupuesto con las horas trabajadas y su precio equivalente, teniendo en cuenta que la duración de este proyecto oscila entre las 300 y 360 horas, equivalente a los 12 créditos que supone la asignatura Trabajo Fin de Grado

Tarea	Horas	Valor por unidad	Valor(€)
Horas de desarrollo	230	20	4600
Horas de Investigación	60	15	900
Redacción de Memoria	45	15	675
Total	335	-	6175

Tabla 7.1: Resumen de tipos

Bibliografía

[1] Fukushima, Kunihiko . URL Kunihiko Fukushima - Wikipedia

[2] Neocognitron (1980). URL https://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf

[3] LeCun, Yann; Léon Bottou; Yoshua Bengio; Patrick Haffner (1998). URL lecun-01a.pdf

[4] Dan Ciresan. URL Dan Ciresan - Google Académico

[5] Deep learning market size, share and trends analysis report by solution. URL Deep Learning Market Size, Share | Industry Growth Report, 2025 (grandviewresearch.com)

[6] TensorFlow. URL https://www.tensorflow.org/

[7] Keras. URL https://keras.io/.

[8] Our World In Data. Moore's Law: Transistors per microprocessor. URL Moore's Law: Transistors per microprocessor (ourworldindata.org)

[9] Ley de Moore. URL Ley de Moore - Wikipedia, la enciclopedia libre

[10] TensorWatch. URL microsoft/tensorwatch: Debugging, monitoring and visualization for Python Machine Learning and Data Science (github.com)

[11] Microsoft Research. URL Microsoft Research – Emerging Technology, Computer, and Software Research

[12] Jupyter Notebook. URL Project Jupyter | Home

[13] Lime. URL marcotcr/lime: Lime: Explaining the predictions of any machine learning classifier (github.com)

[14] CNN Explainer. URL CNN Explainer (poloclub.github.io)

[15] Dial-App. URL dial (github.com)

[16] Python . URL Welcome to Python.org

[17] Qt . URL Qt | Cross-platform software development for embedded & desktop

[18] Git. URL Git (git-scm.com)

[19] GitHub. URL GitHub

[20] David Afonso Dorta. URL David Afonso Dorta | LinkedIn

[21] Framework para agilizar la aplicación de técnicas basadas en Deep Learning . URL Framework para agilizar la aplicación de técnicas basadas en Deep Learning (ull.es)

[22] Dial-APP. URL dial (github.com)

[23] Implementación de Red Neuronal Convolucional RAW TF2. URL Image_Classification_TF2.ipynb - Colaboratory (google.com)

[24] Dependency Injector. URL Dependency Injector — Dependency injection framework for Python — Dependency Injector 4.32.3 documentation (ets-labs.org)

[25] Patrón diseño: Inyección de dependencias. URL Inyección de dependencias - Wikipedia, la enciclopedia libre

[26] QML . URL QML Applications | Qt 5.15

[27] PySide. URL Qt for Python | The official Python bindings for Qt

[28] PyQT. URL Riverbank Computing | Introduction

[29] QT Company. URL The Qt Company

[30] Modelo-Vista-Controlador . URL Modelo–vista–controlador - Wikipedia, la enciclopedia libre

[31] Patrón MVC . URL Modelo–vista–controlador - Wikipedia, la enciclopedia libre

[32] Documentación QT . URL Qt Documentation | Home

[33] Qt Designer . URL Qt Designer Manual

[34] Repositorio Dial-Visualization . URL https://github.com/JDM-ULL-93/Dial-Visualization

[35] Backpropagation in a convolutional layer by Pierre JAUMIER. URL Backpropagation in a convolutional layer | by Pierre JAUMIER | Towards Data Science

[36] Operación de Convolución. URL Kernel (image processing)-Convolution- Wikipedia

[37] Operación de Pooling. URL Max-pooling / Pooling - Computer Science Wiki

[38] Funciones de activación. URL Activation function - Wikipedia

[39] Regla de la cadena en derivadas parciales. URL Chain rule - Wikipedia

[40] Visualizing Higher-Layer Features of a Deep Network p3.URL [PDF] Visualizing Higher-Layer Features of a Deep Network | Semantic Scholar

[41] Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps . URL [1312.6034] Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps (arxiv.org)

[42] Striving for Simplicity: The All Convolutional Net . URL [1412.6806] Striving for Simplicity: The All Convolutional Net (arxiv.org)

[43] Learning Deep Features for Discriminative Localization . URL [1512.04150] Learning Deep Features for Discriminative Localization (arxiv.org)

[44] Función SoftMax . URL Función SoftMax - Wikipedia, la enciclopedia libre

[45] Logit definition in CNN context. URL machine learning - What is the meaning of the word logits in TensorFlow? - Stack Overflow

[46] Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. URL [1610.02391] Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization (arxiv.org)

[47] Axiomatic Attribution for Deep Networks . URL [1703.01365] Axiomatic Attribution for Deep Networks (arxiv.org)

[48] Dial-Visualization . URL https://github.com/JDM-ULL-93/Dial-App

[49] Visualizing and Understanding Convolutional Networks . URL https://arxiv.org/abs/1311.2901