Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Framework para agilizar la aplicación de técnicas basadas en Deep Learning

Framework for streamlining the application of Deep-Learning techniques

David Afonso Dorta

La Laguna, 11 de septiembre de 2020

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

D. **Iván Castilla Rodríguez**, con N.I.F. 78.565.451-G profesor Contratado Doctor 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:

”*Framework para agilizar la aplicación de técnicas basadas en Deep Learning*”

ha sido realizada bajo su dirección por D. **David Afonso Dorta**, con N.I.F. 79.074.017-V.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de septiembre de 2020

Agradecimientos

A mi tutor Rafael Arnay del Arco y cotutor Iván Castilla Rodríguez, por el entusiasmo y dedicación que han demostrado al 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 Deep Learning está en auge. Es una de las áreas con más relevancia hoy en día dentro del ámbito de la Inteligencia Artificial, y al igual que muchos investigadores trabajan en crear nuevos modelos con los que lograr mejores resultados, también hay una gran cantidad de personas que se esfuerzan en hacer que estas nuevas tecnologías sean accesibles para el público. Por ejemplo, creando frameworks como Tensorflow, Keras, o PyTorch.

Contribuyendo con este esfuerzo, en este proyecto se pretende desarrollar un framework basado en Keras/Tensorflow para agilizar la aplicación de técnicas de Deep Learning a problemas de diversa índole. Usando este framework como base, se ha creado una herramienta de interfaz gráfica que el usuario podrá utilizar, sin necesidad de programar ni de conocer ningún lenguaje/librería específico. Eliminando estos requisitos previos, se podría acercar este campo a un gran número de usuarios menos expertos o sin un currículum orientado a la informática, pero donde el Deep Learning afecta transversalmente, como las Matemáticas o las Ingenierías.

Palabras clave: Deep Learning, Framework, Keras, Tensorflow, Python

Abstract

The Deep Learning field is booming. It is one of the most relevant fields on Artificial Intelligence these days, and with many researchers working on creating new models for archiving better and better results, there is also a huge effort made by people who work on making these new technologies accessible by everyone. For example, creating frameworks like Tensorflow, Keras, or PyTorch.

Contributing to this effort, in this thesis we will present a new framework based on Keras/Tensorflow for streamlining the application of Deep Learning techniques to several problems. Using this framework as a base, we have developed a graphic interface tool for the user to interact with, without the need to know or use any specific programming language or library. By removing these requirements, we could close the gap between Deep Learning and amateur users, or people without a background on computer science, on fields where it affects transversely like Mathematics or Engineering.

Keywords: Deep Learning, Framework, Keras, Tensorflow, Python.

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 productos resultantes de este trabajo (framework y herramienta gráfica), justificando la necesidad de disponer de las herramientas desarrolladas, y comparándolos con proyectos previos utilizados como referencias.

1.1. Antecedentes

El Deep Learning es un campo cuyo crecimiento está en auge [8]. Sin embargo, un área que cambia tan rápido también conlleva que su barrera de entrada sea mayor que en otras tecnologías más asentadas.

Debido a esto, durante los últimos años la comunidad de desarrolladores e investigadores ha estado haciendo esfuerzos en democratizar el acceso a los últimos descubrimientos dentro del Deep Learning. Por ejemplo, muchos investigadores han liberando los pesos de las redes neuronales que entrenan, permitiendo que otros investigadores puedan replicar los resultados de la investigación con exactitud, y dando a toda la comunidad la oportunidad de crear proyectos con potentes modelos entrenados en supercomputadores.

Contribuyendo a esto, como esfuerzo de grandes compañías y desarrolladores independientes han surgido librerías para facilitar la programación de nuevos modelos y redes neuronales. Por ejemplo, Google liberó Tensorflow [26] en 2015, y en la actualidad (2020) es uno de los frameworks con más presencia, abstrayendo e implementando la matemática subyacente a los algoritmos de Deep Learning. También ha contribuido a la aparición de Keras [16], o PyTorch [25], que se basan en Tensorflow y abstraen su funcionamiento para ofrecer APIs mucho más programáticas y fáciles de utilizar para el usuario.

Sin embargo, estos frameworks aún requieren que el usuario tenga conocimientos previos en programación (concretamente Python) y en alguna de estas librerías. Si bien esta barrera de entrada es mucho más baja, sigue siendo un impedimento para un público potencial: usuarios e investigadores ajenos a la informática pero donde el Deep Learning afecta de manera transversal: Matemáticas [9], Medicina [10], Industria Automotriz [11],

Traducción [12], etc.

1.2. Objetivos

El Deep Learning es un campo muy matemático y considerablemente complejo. Disponer de visualizaciones, como ver la estructura de una red neuronal, o gráficas sobre el proceso de entrenamiento ayudan mucho a la comprensión, a tomar decisiones más informadas, y a agilizar el ciclo de desarrollo.

Por tanto, la idea principal del proyecto era la de crear una herramienta de interfaz gráfica que cumpliera, como mínimo, con las siguientes características:

- **Fácil de utilizar**: La interfaz gráfica tendría que ser intuitiva de utilizar, evitando en todo lo posible afectar negativamente a la experiencia de usuario, e intentando aprovechar al máximo el apartado gráfico: Componentes sencillos de controlar, un diseño de botones e iconos coherente, colocar textos de ayuda y diálogos para guiar al usuario durante el proceso...
- **Extensible mediante plugins**: A diferencia de otras interfaces gráficas, que solo ofrecen una cantidad limitada de opciones en la interfaz, uno de los principales puntos de esta herramienta consistiría en que cualquier usuario pueda crear y añadir nuevas ventanas y widgets gráficos al programa. Este acercamiento ofrece mucha más flexibilidad para explorar nuevas interfaces gráficas, y se le da la oportunidad a cualquier usuario de que desarrolle sus propios widgets si fuera necesario.
- **Open Source**: Es un proyecto de una escala considerable, por lo que es de esperar que aunque el desarrollo comience con una sola persona, con el tiempo se busque ayuda de usuarios externos para escalar la aplicación. Además, como ya se comentó en el requisito anterior, si se quiere que más usuarios puedan contribuir desarrollando plugins para el programa, es esencial que dispongan de ejemplos en el código base, o que puedan acabar incorporando sus plugins al programa principal.
- **Potente**: Idealmente, la aplicación gráfica debería tener el mismo potencial que sus librerías equivalentes, como Keras o Tensorflow. Estas librerías disponen de una comunidad mucho más grande contribuyendo en ellas, y por eso este proyecto las utiliza como base para implementar las operaciones y algoritmos de Deep Learning. Por otra parte, volviendo al caso de que es extensible por los usuarios, cualquier persona sería capaz de implementar nuevos widgets gráficos, tan específicos como fuera necesario, dando a la aplicación mucho más potencial que otras herramientas gráficas que solo disponen de varias ventanas predeterminadas.

1.3. Referencias

Previo al comienzo del desarrollo, se llevó a cabo una fase de investigación para conocer el estado del arte y las alternativas ya existentes al programa a realizar. Durante esta fase se identificaron varios proyectos que han servido como inspiración y referencia para el producto final.

1.3.1. Deep Learning Studio

Figura 1.1: Logo de Deep Learning Studio

Existen varios proyectos de herramientas gráficas enfocadas al Deep Learning. Sin embargo, la gran mayoría están discontinuadas (DIGITS [5]), o son software privativo y de pago (Neural Network Console - Sony [19]). Uno de los pocos proyectos mantenidos, y que además goza de cierta popularidad es Deep Learning Studio [7] (Figura 1.2).

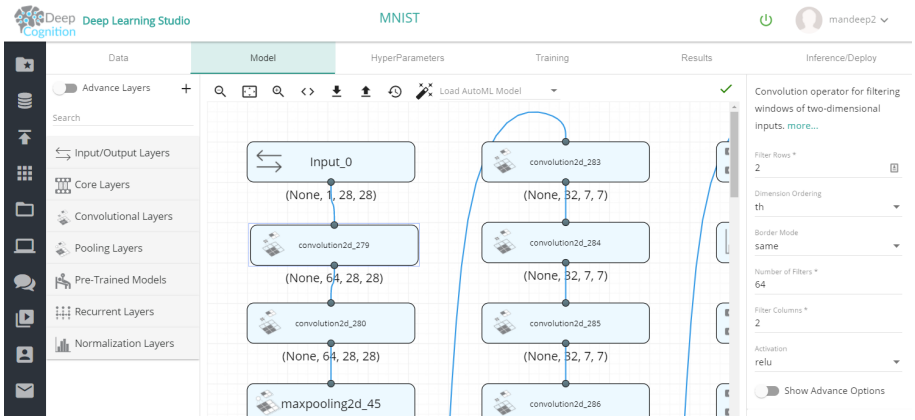

Figura 1.2: Captura del menú principal de Deep Learning Studio

Este programa se vende principalmente como una herramienta gráfica sencilla de utilizar, permitiendo componer redes neuronales arrastrando capas visualmente, cargar y visualizar diferentes datasets, visualizar gráficas del entrenamiento, y exportar los modelos entrenados. Además, cuenta con varios algoritmos para asistir al usuario durante el proceso de composición de las redes neuronales, sugiriendo qué capas o hiperparámetros utilizar.

Las características que implementa pueden ser suficiente para una gran parte de los usuarios, pero la principal diferencia con el programa a realizar es que esta alternativa es de código privativo, y no permite que otros usuarios extiendan su funcionamiento, por lo que está limitada en cuanto a posibles aplicaciones.

1.3.2. Weka

Figura 1.3: Logo de Weka

Desarrollado por la Universidad de Waikato, Nueva Zelanda, Weka [29] es una interfaz gráfica potente, extensible mediante plugins, y además completamente gratuita y de software libre (publicado bajo la licencia GPL-3.0). Teniendo en cuenta que es bastante popular, es un buen marco de referencia para demostrar el potencial que podría llegar a tener una aplicación de este estilo (Figura 1.4).

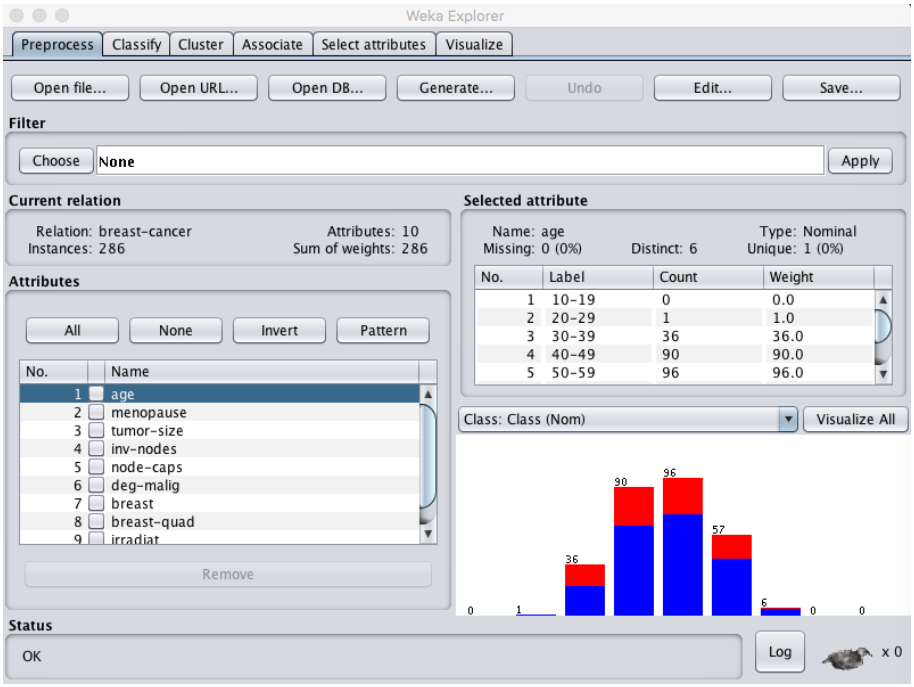

Figura 1.4: Captura del menú principal de Weka

De hecho, esta misma herramienta se utiliza en una de las asignaturas del grado de Ingeniería Informática en la Universidad de La Laguna, Tratamiento Inteligente de Datos, para llevar a cabo todo un proyecto de Machine Learning por parte de los alumnos, permitiéndoles centrarse solo en los algoritmos y aplicaciones sin necesidad de que tengan que aprender Python o R previamente.

1.3.3. Blender Node Editor

Figura 1.5: Logo de Blender

Blender [1] es una herramienta de modelado 3D gratuita y de software libre. Si bien no tiene relación con las áreas de Deep/Machine Learning, de una de sus interfaces se ha sacado la característica más destacable de la herramienta gráfica implementada: el editor de nodos (Figura 1.6).

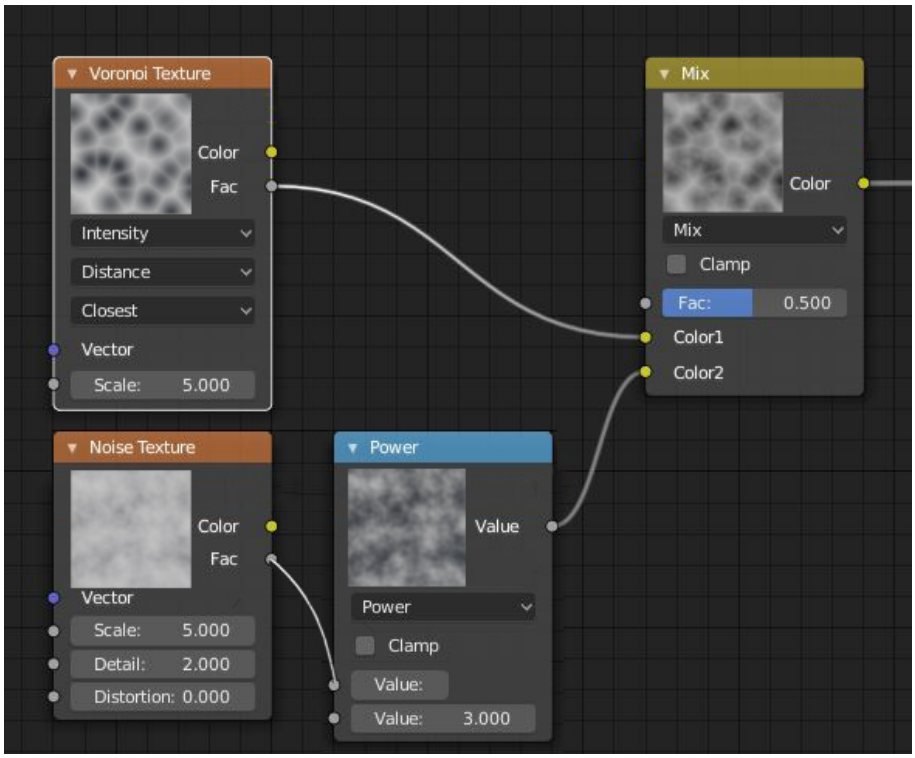

Figura 1.6: Captura del editor de nodos de Blender

Un editor de nodos se basa principalmente en dos componentes: Nodos y Puertos. Los nodos cumplen una función específica (en el caso de Blender, este editor de nodos se utiliza para componer materiales y texturas, por lo que hay nodos para cambiar y combinar colores, aplicar patrones geométricos, o aplicar diversas transformaciones matemáticas). Cada nodo ofrece una serie de puertos de entrada y de salida. Consume los datos a través de los puertos de entrada, los transforma, y envía los nuevos datos a través de los puertos de salida al resto de nodos a los que esté conectado. A medida que se conectan más nodos se va formando una red (o grafo) más compleja, pero en la que se puede ver en todo momento como fluyen los datos y quién se está encargando de transformarlos, lo que es especialmente útil cuando se quiere descomponer el grafo para entender su funcionamiento parte por parte.

1.4. Resultados

Como resultados de este TFG se han desarrollado dos productos: *dial-core* y *dial-gui*.

1.4.1. dial-core

dial-core es un framework desarrollado en Python. Se podría ver como una capa de abstracción por encima de Keras/Tensorflow. Basa su funcionamiento en definir componentes llamados nodos, cada uno especializado en una función diferente. Estos nodos disponen de puertos, que sirven como puntos de conexión entre diferentes nodos (siempre que los puertos sean compatibles) y mediante los cuales los datos van fluyendo a través del grafo que se forma.

En un posible ejemplo, un nodo podría encargarse de importar la arquitectura de una red neuronal. Otro nodo se encargaría de definir los hiperparámetros de entrenamiento. A través de sus puertos, tanto la red neuronal como los hiperparámetros se podrían propagar hasta otro nodo, esta vez uno encargado de ejecutar el entrenamiento. Tras entrenar la red, el resultado se podría conectar al puerto de otro nodo encargado de comprobar y mostrar la precisión de la red neuronal.

Al final, si viéramos todos estos nodos en conjunto tendríamos como resultado un grafo que se encarga de instanciar, entrenar y testear una red neuronal.

1.4.2. dial-gui

dial-gui es la herramienta de interfaz gráfica desarrollada utilizando *dial-core* como base. Ofrece el mismo funcionamiento lógico que *dial-core* (conectar nodos entre si mediante sus puertos), pero con la ventaja de que ahora cada nodo puede configurarse mediante widgets gráficos (diálogos para importar archivos, cajas y selectores de texto, tablas, gráficas, ayudas para explicar el funcionamiento de partes más intrincadas, etc)

Además de visualizar estos componentes como un editor de nodos (Figura 1.7), también se pueden agrupar en ventanas para ofrecer una interfaz más tradicional pero igual de modular (Figura 1.8).

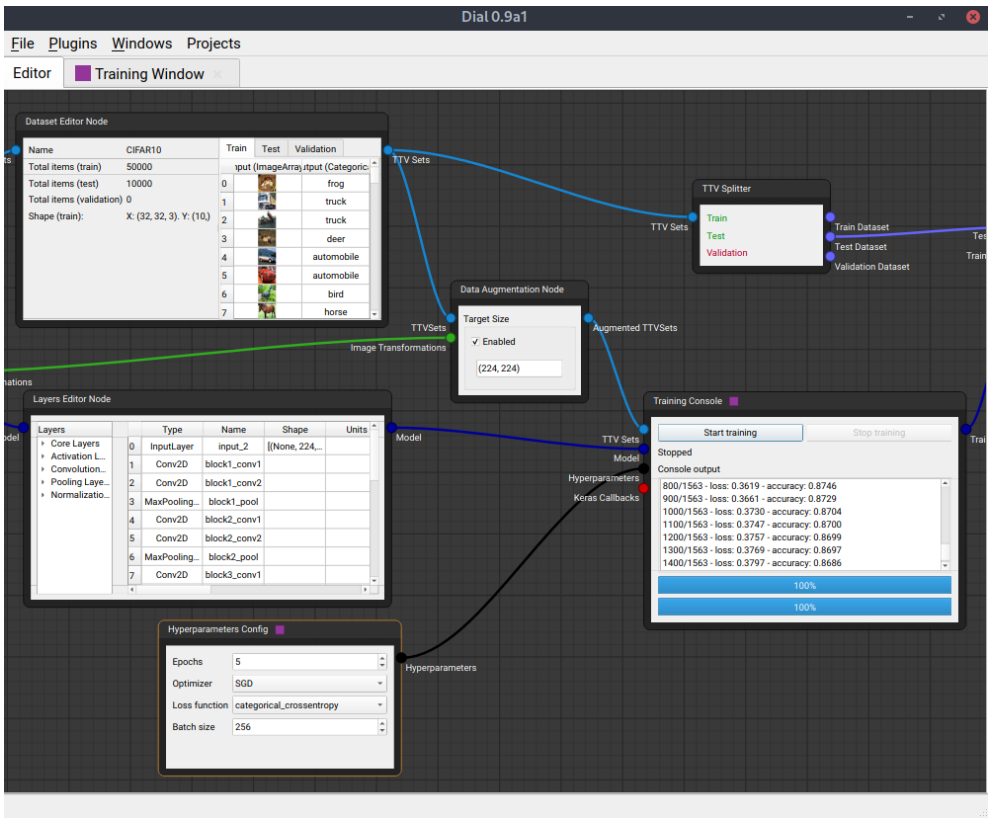

Figura 1.7: Ejemplo de la interfaz gráfica de dial-gui

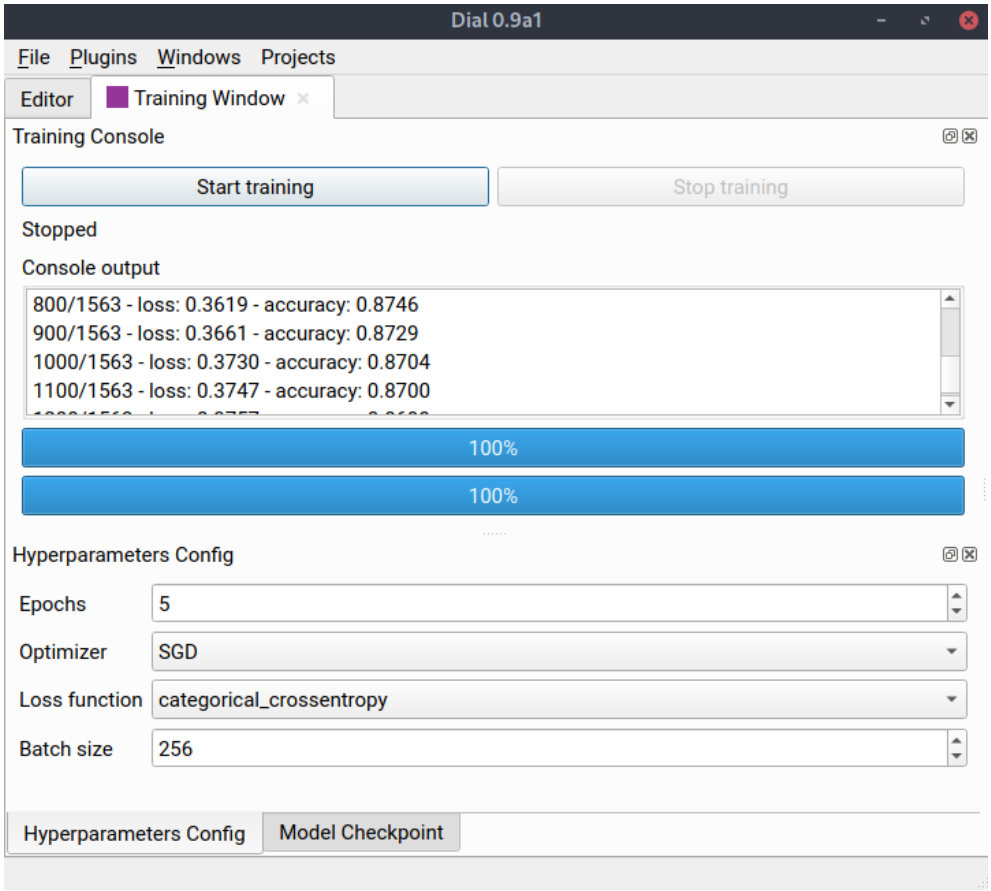

Figura 1.8: Interfaz gráfica de dial-gui mostrando los widgets en una ventana tradicional

Capítulo 2

Tecnologías

2.1. Tecnologías utilizadas durante el desarrollo

En este apartado se recogerán las tecnologías (librerías, lenguajes de programación y dependencias) utilizadas por el proyecto realizado, así como las ventajas que aportan.

2.1.1. Python

Figura 2.1: Logo de Python

Python [23] es el lenguaje de programación en el que se ha implementado el proyecto, tanto para *dial-core* como para *dial-gui*.

El principal aliciente de usar este lenguaje ha sido para poder utilizar las librerías de Deep Learning de Tensorflow y Keras, implementadas también en Python. Así mismo, es uno de los lenguajes de programación más populares hoy en día [18].

Respecto a la versión, se ha utilizado como base la versión 3.6, siendo también compatible con las versiones 3.7 y 3.8. La idea de utilizar esta versión es que a partir de esta el lenguaje incluye soporte para ”type hinting”.

El ”type hinting” es una característica que permite, opcionalmente, asignar un tipo a una variable o un argumentos de una función, haciendo que Python se comporte como si fuera un lenguaje fuertemente tipado con todas sus ventajas: errores detectables en ”tiempo de compilación” cuando los tipos no sean compatibles, mayor facilidad para entender qué datos se espera que reciba y devuelva una función, capacidad de navegar el

código con más facilidad en los IDEs compatibles, permitiendo saltar a las declaraciones de los tipos...

Ya que se ha abandonado el soporte desde principios de 2020 [24], esta aplicación no es compatible con Python 2.

2.1.2. Git, Github, PyPI

Figura 2.2: Logos de Git, Github, y PyPI

Es esencial llevar un control de versiones en el proyecto para poder mantener un historial de cambios, volver a versiones anteriores del código, o trabajar de manera más ordenada utilizando ramas.

Con este objetivo se ha utilizado Git [13], junto a Github [14] para almacenar el código en un repositorio online público. Esto va en línea con el propósito de que el proyecto sea Open Source.

Además, no solo se ha utilizado Github para alojar el código. Algunas de sus características, como los Issues (tickets donde registrar tareas a realizar, nuevas características, bugs encontrados...), o las Pull Requests (combinar una serie de commits realizados en una rama secundaria a la rama principal, solo tras haber comprobado que pasa los tests necesarios) han sido de utilidad a la hora de organizar el desarrollo en el proyecto.

En cuanto a PyPI [21], esta plataforma permite alojar código de Python en paquetes, que luego se podrán instalar mediante el gestor de paquetes pip [20]. Gracias a esta característica, es muy sencillo distribuir el programa realizado con diferentes usuarios: basta que lo instalen en su ordenador mediante el comando pip. PyPI también avisará a los usuarios de cuando hayan versiones nuevas del programa, para que se puedan mantener en la última versión del desarrollo.

2.1.3. Pytest, Travis y Codecov

Figura 2.3: Logos de PyTest, Travis y Codecov

Para mantener unas buenas practicas de programación, se han utilizado las aplicaciones Pytest [22], Travis CI [27] y Codecov [3] para las tareas de testeo e integración continua (CI).

En primer lugar, Pytest es la librería utilizada para definir y ejecutar los tests en Python. Facilita la creación de test unitarios, test parametrizados, y además permite inyectar en los tests objetos definidos previamente (concepto que denominan como "fixtures"en la librería). También tiene una librería opcional, pytest-mock, para facilitar el inyectar objetos como Mocks (un mock es un objeto que se comporta como si fuera de una clase específica, pero cuya implementación está totalmente simulada. Por ejemplo, en lugar de ejecutar el contenido de una función, simplemente devuelve un valor que le hemos especificado, o lanza una excepción para determinados argumentos).

Seguidamente, TravisCI es la solución de integración continua utilizada. Se mezcla perfectamente con Github, ya que cada vez que se añade un nuevo commit, TravisCI lanza una nueva tarea donde ejecuta todos los tests incluidos en el commit y devuelve el resultado. En caso de que los tests no hayan pasado, el commit se marcará en rojo en Github para indicar que la última versión tiene algún problema que debe ser investigado. Además, Travis es gratuito para proyectos Open Source, por lo que nos podemos aprovechar de esta ventaja.

Finalmente, Codecov permite conocer qué porcentaje del código se ha cubierto en los tests. Esto nos da una información muy importante: conocer que clases y métodos se han dejado sin comprobar que funcionen correctamente en los tests. Un código sin comprobar podría introducir un error en nuestro programa sin darnos cuenta.

Codecov recoge las estadísticas de cubrimiento de código directamente desde Travis, por lo que podemos estar seguro de que siempre va a reflejar el cubrimiento de la última versión publicada en Github.

Ya que dial-core es el framework en el que se basan la interfaz y el resto de nodos, era especialmente importante mantener un cubrimiento lo más alto posible en este (Figura 2.4).

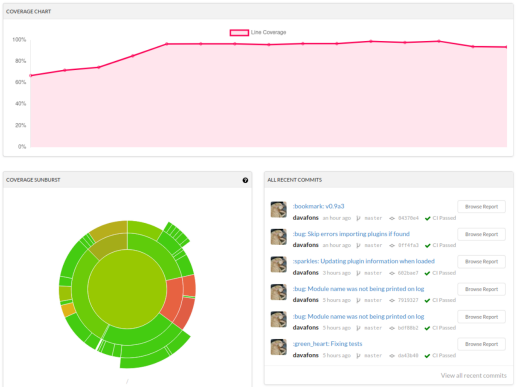

Figura 2.4: Coverage del paquete dial-core al 94 %

2.2. Dependencias de dial-core

Aunque se ha programado desde cero todo el sistema de nodos, conexiones y ejecución de la librería principal del proyecto (*dial-core*), también se han utilizado otras librerías populares para implementar partes importantes del funcionamiento.

2.2.1. Tensorflow y Keras

Figura 2.5: Logos de Tensorflow y Keras

Tensorflow [26] es una librería Open Source creada por Google, centrada en la computación numérica mediante grafos de flujo de datos. En esencia, en este tipo de grafo un nodo representaría una operación, y los arcos que lo conectan serían los datos que recibe para operar (o los que produce como resultado de la operación). A medida que se van conectando estos nodos se forma un grafo que representa una operación más y más compleja.

La principal ventaja de este tipo de grafos es que es relativamente sencillo conocer las dependencias: que datos se pueden calcular y cuales dependen de los resultados de operaciones anteriores antes de poder ejecutarse.

Además, separar una operación compleja en varios nodos de operación más pequeños permitiría ejecutarla de forma paralela o distribuida. Teniendo en cuenta que las redes neuronales normalmente tratan con operaciones entre matrices de grandes dimensiones, realizar estos cálculos de forma paralela en, por ejemplo, una GPU, aceleraría en gran medida las fases de entrenamiento.

Sin embargo, la computación numérica también tiene otras aplicaciones más allá del Deep Learning, por lo que Tensorflow se mantiene como una librería relativamente genérica.

Es para cubrir este hueco que aparece Keras. También Open Source, Keras [16] es una API: una capa de abstracción por encima de Tensorflow que presenta nuevas funciones, tipos de datos y clases centradas en el Deep Learning. Usando Keras podríamos instanciar una red neuronal en unas pocas líneas de código, tal como muestra el Listado 2.1.

Listado 2.1: Ejemplo de como utilizar Keras

```
# Define Sequential model with 3 layers
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
```

```
        layers.Dense(4, name="layer3"),
    ]
)
# Call model on a test input
x = tf.ones((3, 3))
y = model(x)
```

Se podría considerar que *dial-core* (y *dial-gui*) son una capa de abstracción por encima de Keras, agrupando estas funciones y clases de Deep Learning en widgets más concretos (Figura 2.6).

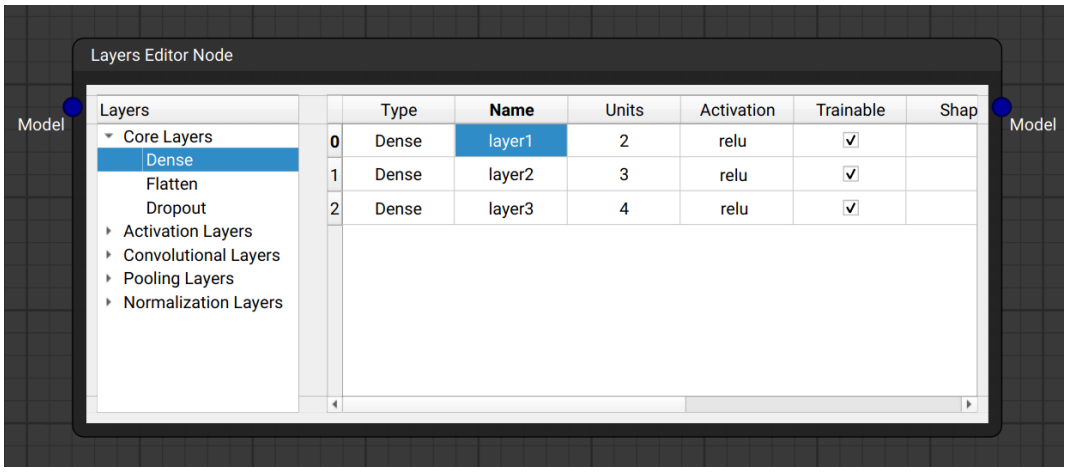

Figura 2.6: Equivalente en dial-gui del código en Keras para instanciar la red neuronal

2.2.2. dependency-injector

Figura 2.7: Logo de Dependency Injector

Esta dependencia es algo peculiar ya que no implementa ninguna funcionalidad que se utilice en el programa, sino que extiende el lenguaje Python para introducir un patrón importante a la hora de diseñar el código: la Inyección de Dependencias.

Este patrón consiste en un concepto introducido por Martin Fowler en 2004 [17]. Mediante la inyección de dependencias, se pretende que si una clase A depende de otra clase B, reciba una instancia de esta que pueda utilizar, en vez de crearla ella misma. El Listado 2.2 muestra un ejemplo de su uso.

Listado 2.2: Ejemplo de injección de dependencias en dial-gui

```
class GraphicsNode(QGraphicsObject):
    def __init__(
        self,
        node: "Node",
```

```
        painter_factory: "providers.Factory",
        parent: "QGraphicsItem" = None,
    ):
        super().__init__(parent)

        ...

GraphicsNodeFactory = providers.Factory(
    GraphicsNode, painter_factory=GraphicsNodePainterFactory.delegate()
)
DisabledGraphicsNodeFactory = providers.Factory(
    GraphicsNode, painter_factory=DisabledGraphicsNodePainterFactory.delegate()
)
```

En el ejemplo anterior, sacado de *dial-gui*, podemos ver este concepto en acción: la clase `GraphicsNode` (que representa un nodo en el editor) recibe como uno de los argumentos un `painter_factory`, la clase encargada de pintar el propio nodo en la intefaz gráfica.

Por tanto, inyectando esta dependencia podemos crear nuevos tipos de datos que representen diferentes formas de pintar un nodo, o incluso reemplazar esta dependencia de forma dinámica (para cambiar la forma en la que se pinta el nodo según las acciones del usuario)

Relacionado con los tests, utilizar este patrón de diseño facilita mucho el introducir esta dependencia como un mock, testeando solo el comportamiento interno de la clase.

2.3. Dependencias de dial-gui

2.3.1. PySide2 (Qt5)

Figura 2.8: Logos de Pyside2 y Qt

PySide2 es la librería oficial que ofrece soporte en Python para Qt5, un framework multiplataforma orientado a objetos utilizado para diseñar programas que utilicen interfaces gráficas. Originalmente, Qt está programado en C++, de ahí que necesitemos utilizar este ”binding” (PySide2) para poder ejecutarlo desde Python.

Es también una librería Open Source, y una de las principales alternativas a la hora de programar interfaces de usuario ”tradicionales” (que se ejecutan en el propio ordenador como una aplicación, a diferencia de las interfaces de usuario basadas en navegadores web que se han popularizado en los últimos años).

El principal motivo por el que se había elegido esta librería era porque ya se había

utilizado previamente en otras asignaturas de la carrera y se tenía cierto conocimiento previo, además de que encajaba bien con el proyecto.

Independientemente de que sea una librería para el diseño de interfaces gráficas, también introduce varios patrones de diseño únicos que facilitan la interacción con esta, como el sistema de signals y slots.

Un slot es, básicamente, una función. Recibe una serie de argumentos y los procesa. Por otra parte, una signal también es una función, pero se comporta de manera diferente. Primero hay que conectar la signal a uno o varios slots. Tras realizar esto, en cualquier momento en el que se ”emita” la signal (llamando a la función `emit()` de la signal con los argumentos apropiados), se invocarán las funciones slot a las que esté conectada. En el Listado 2.3 se muestra el código para realizar estas conexiones entre signals y slots.

Listado 2.3: Ejemplo del patrón signal/slot en Qt

```
@Slot(str)
def say(name: str):
    print(f"Hello␣{name}!")

greetings = Signal(str)
greetings.connect(say)

greetings.emit("foo")  // Will print "Hello␣foo!"
```

Capítulo 3

Arquitectura y diseño

En este capítulo hablaremos del diseño general de la aplicación, como interaccionan los diferentes módulos y clases que componen tanto *dial-core* como *dial-gui*, y repasaremos los nodos que se han implementado para la interfaz gráfica mediante un ejemplo.

3.1. dial-core

Podemos comenzar introduciendo la estructura de *dial-core* con un diagrama (Figura 3.1) de las principales clases que lo componen.

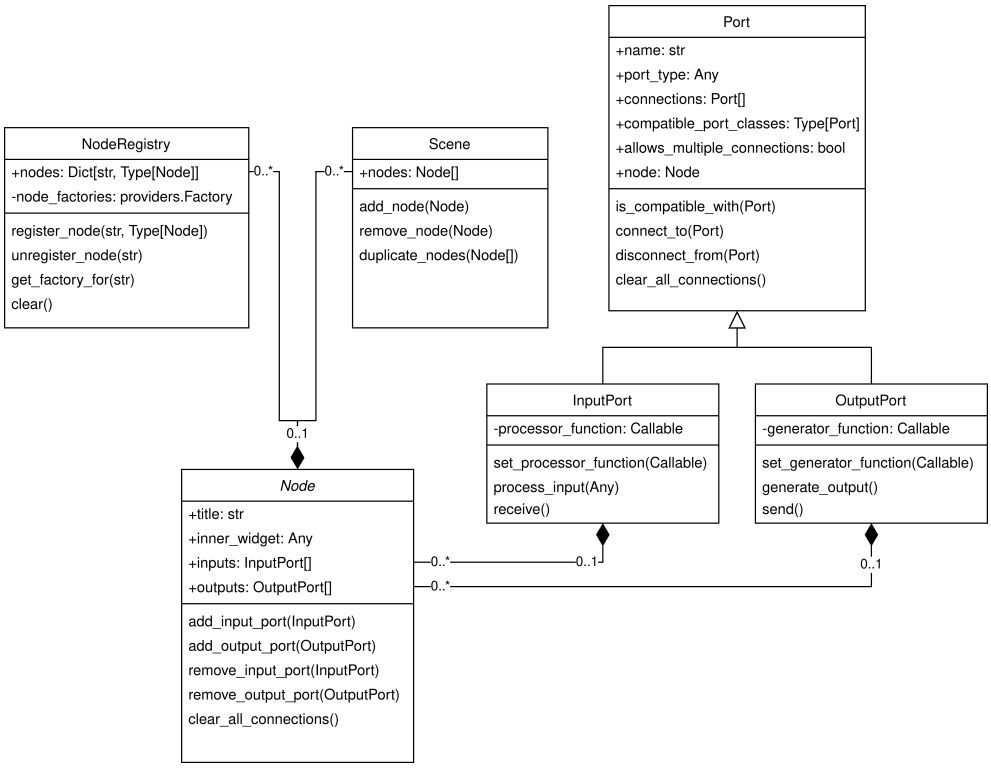

Figura 3.1: Diagrama de clases de dial-core

En esencia, la clase Scene actuaría como el ”contenedor” del grafo, almacenando todos los nodos que lo componen. Por cada nodo, que son las unidades principales que se encargarán de almacenar y procesar los datos, definiríamos una serie de puertos de entrada (InputPort) y de salida (OutputPort). Los puertos gestionan las conexiones con otros puertos (y sus nodos asociados), y transmiten los datos entre ellos. Por último, la clase NodeRegistry nos permite registrar ”fábricas” de nodos, para luego poder instanciarlos más fácilmente.

Esta clase será especialmente útil cuando estemos trabajando con nodos más complejos, pudiendo definir diferentes ”configuraciones iniciales” de un mismo nodo, como se muestra en el Listado 3.1.

Listado 3.1: Ejemplo de NodeRegistry

```
class TalkingNode(Node):
    def __init__(self, message):
        super().__init__("Talking Node")

        self.message = message

    def say(self):
        print(f"I say {self.message}")

node_registry.register_node("SayFooNode", TalkingNode, message: "foo!")
node_registry.register_node("SayBarNode", TalkingNode, message: "bar!")

foo_node = node_registry.instance_node("SayFooNode")
bar_node = node_registry.instance_node("SayBarNode")

foo_node.say()   // Prints "I say foo!"
bar_node.say()   // Prints "I say bar!"
```

Pasemos a ver como utilizar estas clases, centrándonos primero en la clase Port. Para instanciar esta clase (Listado 3.2) solo necesitamos dos valores, un nombre (identificador) y un tipo. Este tipo puede ser cualquier clase de Python que sea comparable (una clase comparable es aquella que se puede utilizar con el operador ==), es decir, primitivas como int, string, boolean, clases creadas por el usuario, enums...

Introducimos el concepto de tipo para poder definir que puertos son compatibles, y por tanto, pueden conectarse entre ellos.

Listado 3.2: Código del constructor la clase Port

```
class Port:
    def __init__(self, name: str, port_type: Any, allows_multiple_connections: bool = True):
        self._name = name
        self._port_type = port_type
        self._connected_to: Set["Port"] = set()  # Avoid repeated ports
        self.compatible_port_classes: Set[Type["Port"]] = set([Port])

        self.node = None

        self.allows_multiple_connections = allows_multiple_connections
```

El constructor también define otros argumentos interesantes:

- allow_multiple_connections nos permite definir si un puerto puede estar conectado a varios puertos (relación uno-muchos), o si solo se puede conectarse a un solo otro puerto (relación uno-uno).
- _connected_to es un set que contendrá todos los puertos a los que estemos conectados. Lo usaremos para poder desplazarnos por el grafo, saltando entre los diferentes puertos (y nodos) conectados.
- Con la variable compatible_port_classes podemos especificar qué nodos se podrán conectar a este (útil cuando empecemos a hacer subclases más específicas de Port).
- node almacenará una referencia al objeto Node al que pertenezca (en caso de que esté incluido en uno).

En el listado 3.3 se muestra el método que define todas las comprobaciones que se realizan para saber si dos puertos son compatibles:

Listado 3.3: Método is_compatible_with de la clase Port

```
def is_compatible_with(self, port: "Port") -> bool:
    return (
        self is not port
        and self._port_type == port.port_type
        and (not self.node or self.node != port.node)
        and type(port) in self.compatible_port_classes
    )
```

Principalmente, podemos conectar dos puertos si no son el mismo objeto, comparten el mismo tipo, no pertenecen al mismo nodo (para evitar que se produzcan ciclos al conectar los puertos), y el puerto a conectar pertenezca a la lista de puertos compatibles.

Respecto a cómo podríamos conectar dos puertos, solo haría falta utilizar el método connect_to, especificando los dos puertos a conectar. Este método también se encargaría de actualizar la lista de puertos conectados (_connected_to). Podemos ver un ejemplo en el listado 3.4.

Listado 3.4: Mostrando cómo conectar tres puertos entre ellos

```
port_a = Port(name: "a", port_type: str, allow_multiple_connections: True)
port_b = Port(name: "b", port_type: str, allow_multiple_connections: False)
port_c = Port(name: "c", port_type: str, allow_multiple_connections: False)

port_a.connect_to(port_b)
port_a.connect_to(port_c)

print(a.connections)  // [port_b, port_c]
print(b.connections)  // [port_a]
print(c.connections)  // [port_a]
```

Con la clase Port hemos introducido la lógica para conectar diferentes puertos, pero por si sola esta clase no tiene ninguna capacidad para procesar ningún dato.

Para realizar operaciones con los puertos, debemos introducir dos subclases de Port: InputPort y OutputPort

Estas clases se instancian de la misma manera que Port, pero introducen un atributo nuevo: processor_function para los InputPort, y generation_function para los OutputPort.

La función processor_function, propia de los InputPort, es invocada cada vez que recibimos un valor desde el puerto al que estemos conectados. Esta función la define el usuario, y sirve normalmente para hacer alguna tarea de preprocesamiento con el dato recibido, almacenarlo para más tarde, mostrar el valor en algún log y/o terminal...

Por otra parte, la función generator_function, propia de los OutputPort, se invoca cada vez que queremos generar un valor que propagar al resto de puertos conectados. Esta función también está definida por el usuario. El Listado 3.5 proporciona un ejemplo.

Listado 3.5: Procesando datos con InputPort y OutputPort

```
def generate_random_number() -> int:
    return random.randint(0, 9)

def print_received_number(number: int) -> int:
    print(f"I␣received␣the␣␣number␣{number}")]
    return number

output_port = OutputPort(name: "output_port", port_type: int)
input_port = InputPort(name: "input_port", port_type: int)

output_port.set_generator_function(generate_random_number)
input_port.set_processor_function(print_received_number)

output_port.connect_to(input_port)

output_port.send()              // Prints "I␣received␣the␣number␣2"
number = input_port.receive() // Prints "I␣received␣the␣number␣9"
print(number)                   // "9"
output_port.send()              // Prints "I␣received␣the␣number␣3"
number = input_port.receive() // Prints "I␣received␣the␣number␣4"
print(number)                   // "4"
...
```

Varias cosas están sucediendo en este trozo de código:

1. Definimos dos funciones para generar números aleatorios e imprimirlos.
2. Instanciamos los objetos *InputPort* y *OutputPort*, y les asignamos a cada uno una de las funciones que creamos anteriormente. Además, conectamos los puertos entre ellos.
3. Cuando llamamos a output_port.send(), lo que ocurrirá internamente es que la clase llamara a su generator_function asociado. Tras haber generado el valor, lo siguiente que hará será llamar a la processor_function de cada puerto al que esté conectado, propagando el valor a todas sus conexiones.

4. Por otra parte, cuando llamamos a input_port.receive() pasará algo similar. Internamente, esta función llamará al generator_function de su puerto asociado. Cuando se haya generado el valor, llamará a su propia processor_function para procesarlo y devolverlo.

5. Un detalle a remarcar es que como los dos puertos son de tipo int, ambas funciones (generator y processor) deben devolver y recibir valores del mismo tipo (int).

Utilizando este mecanismo de send y receive podemos propagar valores fácilmente entre dos puertos. Sin embargo, si queremos realizar conexiones con más puertos (y acabar formando un grafo), necesitaremos agruparlos bajo la clase Node, tal como muestra el Listado 3.6.

Listado 3.6: Código del constructor de la clase node

```
class Node:
    def __init__(self, title: str, inner_widget: Any = None, parent: Any = None):
        self.parent = parent

        self._title = title
        self._inner_widget: Optional[Any] = inner_widget

        self._inputs: Dict[str, "InputPort"] = {}
        self._outputs: Dict[str, "OutputPort"] = {}
```

Analicemos el constructor (Listado 3.6) de la clase Node, al igual que hicimos con la clase Port:

- title es el nombre o identificador del nodo.
- inner_widget se instanciará con un objeto que encapsulará el funcionamiento del nodo. Lo veremos más adelante cuando tratemos con nodos más complicados, como aquellos que forman parte de la interfaz de dial-gui.
- inputs y outputs definen un diccionario con los diferentes puertos de salida y de entrada que dispone este nodo. Al crear un nuevo tipo de nodo, definimos en este diccionario que puertos van a tener para definir la "interfaz"de nuestro nodo.

Con solo estos dos componentes (Node y Port) ya tendríamos todas las herramientas necesarias para definir grafos que realicen cálculos más complejos. Por ejemplo, en el código incluido en el Listado 3.7 se muestra como crear un grafo que sea capaz de sumar dos números e imprimir el resultado.

Listado 3.7: Grafo que define una suma entre dos números e imprime el resultado

```
class ValueNode(Node):
    def __init__(self, value=0):
        super().__init__("Value Node")

        # Port configuration
        self.add_output_port(name="value", port_type=int)

        self.outputs["value"].set_generator_function(self._generate_value)

        # Attributes
        self._value = value

    @property
    def value(self) -> int:
        return self._value

    @value.setter
    def value(self, new_value: int):
        self._value = new_value

        self.outputs["value"].send()

    def _generate_value(self) -> int:
        return self.value


class AddNode(Node):
    def __init__(self):
        super().__init__("Addition Node")

        # Port configuration
        self.add_input_port(name="op1", port_type=int)
        self.add_input_port(name="op2", port_type=int)

        self.add_output_port(name="result", port_type=int)

        self.inputs["op1"].set_processor_function(self.save_op1)
        self.inputs["op2"].set_processor_function(self.save_op2)

        self.outputs["result"].set_generator_function(self.get_result)

        # Attributes
        self._op1 = 0
        self._op2 = 0
        self._result = 0

    def save_op1(self, op1: int) -> int:
        self._op1 = op1
        self._update_result()

        return op1

    def save_op2(self, op2: int) -> int:
        self._op2 = op2
        self._update_result()
```

```
        return op2

    def get_result(self) -> int:
        return self._result

    def _update_result(self):
        self._result = self._op1 + self._op2

        self.outputs["result"].send()


class PrintIntNode(Node):
    def __init__(self):
        super().__init__("Print Int Node")

        # Port configuration
        self.add_input_port(name="value", port_type=int)
        self.inputs["value"].set_processor_function(self._print_value)

    def get_result_and_print(self):
        self._print_value(self.inputs["value"].receive())

    def _print_value(self, value):
        print(f"Result is {value}")


# ——— Graph creation ———
node_op1 = ValueNode(4)
node_op2 = ValueNode(3)

add_node = AddNode()

print_node = PrintIntNode()

# ——— Connections ———
node_op1.outputs["value"].connect_to(add_node.inputs["op1"])
node_op2.outputs["value"].connect_to(add_node.inputs["op2"])

add_node.outputs["result"].connect_to(print_node.inputs["value"]) // Prints "Result is 7"

node_op1.value = 10                 // Prints "Result is 13"
node_op2.value = 40                 // Prints "Result is 50"

print_node.get_result_and_print()   // Prints "Result is 50"
```

Entraremos más en detalle con el código.

Primero hemos definido una serie de nodos personalizados: ValueNode se encarga de almacenar un número (int), que puede propagarse a partir del puerto de salida *value* que define. AddNode tiene dos puertos de entrada, *op1* y *op2*, que reciben los valores de los ValueNode conectados, realiza la suma, y expone el resultado en el puerto de salida *result*. Por último, el nodo PrintIntNode imprimirá el valor que reciba en su puerto de entrada *value*.

Tras instanciar los nodos en el código, procedemos a realizar las conexiones entre ellos. Analicemos la lista de funciones que se ejecutan al realizar la conexión entre add_node y node_op1 en la línea node_op1.outputs["value"].connect_to(add_node.inputs["op1"]).

1. El puerto add_node.inputs["op1"] va llamar al generator_function de node_op1.outputs["value"] (que es _generate_value()).
2. La función _generate_value() nos devuelve el valor almacenado cuando se instanció el nodo (4).
3. Volviendo a add_node, este llama a su processor_function con el valor recibido.
4. La processor_function de add_node, save_op1(), almacenará el valor de op1 en el nodo y actualizará el resultado de la suma. En este caso, ya que aún no hemos conectado op2, el resultado de la operación será 4 + 0 = 4.
5. Ya que tampoco tenemos ningún puerto conectado al puerto de salida, no se propaga el resultado y termina la ejecución.

Este mismo proceso ocurrirá cuando ejecutemos la siguiente línea, para conectar los nodos node_op2 y add_node. El valor de op2 se actualizará, al igual que el del resultado, que ahora será 4 + 3 = 7.

Finalmente, en la siguiente línea (add_node.outputs["result"].connect_to(print_node.inputs["value"])) realizaremos la última conexión entre add_node y print_node.

1. El puerto print_node.inputs["value"] va a llamar a la generator_function (get_result()) del puerto add_node.outputs["result"]
2. Un detalle a tener en cuenta es que la función get_result() nos devuelve el cálculo que hemos precalculado y almacenado en add_node._result. Por lo tanto, no es necesario seguir volviendo atrás en el grafo hasta los nodos op1 y op2. Poder ahorrarnos recorrer partes del grafo ”cacheando” estos valores es una característica que será de mucha utilidad al trabajar con nodos más complejos.
3. Tras obtener el resultado de get_result(), se invoca la processor_function (_print_value()) del puerto print_node.inputs["value"]. Esta función finalmente imprimirá el resultado de la operación.

En este punto ya tendríamos todo el grafo conectado. Ahora, si modificáramos el valor de uno de los nodos, este se propagaría por todo el grafo. Por ejemplo, si cambiamos el valor de op1 a 10, se propagaría a add_node, donde se calculará el resultado (13), y se propagaría también a print_node, donde se imprimiría el mensaje ”Result is 13”.

3.2. dial-gui

Este paquete que actúa como interfaz gráfica de *dial-core* nos permite realizar lo mismo que con el framework base, pero mediante el uso de widgets con los que visualizar los diferentes nodos.

En cuanto a funcionamiento general, el editor de nodos incluye las operaciones más básicas que se podrían esperar: desplazar, eliminar y duplicar nodos, así como añadir nodos nuevos haciendo click derecho en el editor y eligiendo el nodo a instanciar en el menú desplegable. La única operación no implementada por el momento es la capacidad de deshacer y rehacer los cambios en los nodos.

Igual que con *dial-core*, podemos introducir la estructura de *dial-gui* utilizando el diagrama de clases de la Figura 3.2.

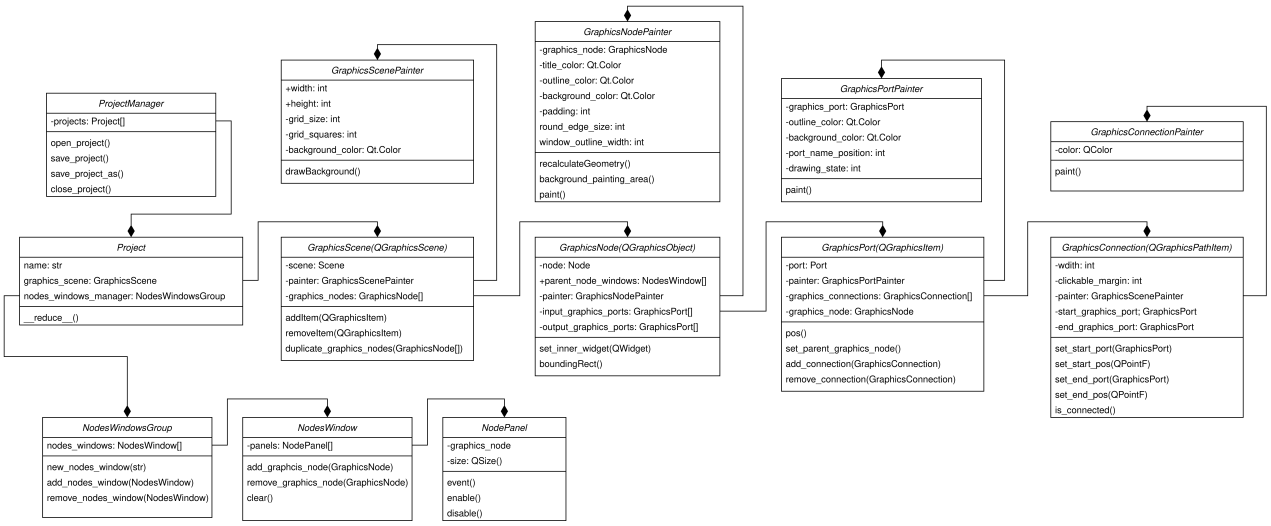

Figura 3.2: Diagrama de clases de dial-gui

Podemos fijarnos en que sigue un diseño muy parecido al de dial-core. Las clases GraphicsScene, GraphicsNode y GraphicsPort son ”wrappers” (clases que engloban otro objeto y le añaden ciertas funcionalidades, sin por ello dejar de poder acceder a los objetos que engloban) de Scene, Node y Port, y se encargan tanto de definir como se van a dibujar en pantalla (utilizando sus clases ∗Painter), como de manejar y responder a los eventos realizados por el usuario (doble click en un nodo, click en un puerto y arrastrar, click derecho...).

Sin embargo, tenemos otras clases que no estaban presentes en *dial-core*, como la clase Project. Un proyecto es una clase contenedor para almacenar la escena (objeto de clase Scene) actual y algunos datos extra como el nombre de proyecto. Además, esta clase es serializable, es decir, se puede almacenar la instancia de la clase como un objeto binario en un archivo .dial.

Por tanto, podríamos decir que la clase Project define el estado de la interfaz y de la aplicación en un momento determinado: qué nodos hay presentes en el grafo, cuál es su posición, qué conexiones tienen, cuál es la configuración interna de cada nodo... Y se vale

en esta capacidad de serializarse para almacenar una copia en memoria que pueda ser cargada más tarde, incluso tras haber cerrado el programa.

Junto a esta clase aparece ProjectManager que es la encargada de gestionar los varios proyectos que puedan haber abiertos al mismo tiempo en la aplicación.

Por otra parte, las clases NodesWindowsGroup, NodesWindow y NodePanel nos permiten mapear los diferentes nodos que hemos definido en ventanas, pudiendo reorganizarse y definir una interfaz más tradicional para trabajar con ciertos nodos si fuera necesario. Las interacciones y el estado del widget se comparten entre las dos visualizaciones (ventana y nodo), tal como se ve en la Figura 3.3.

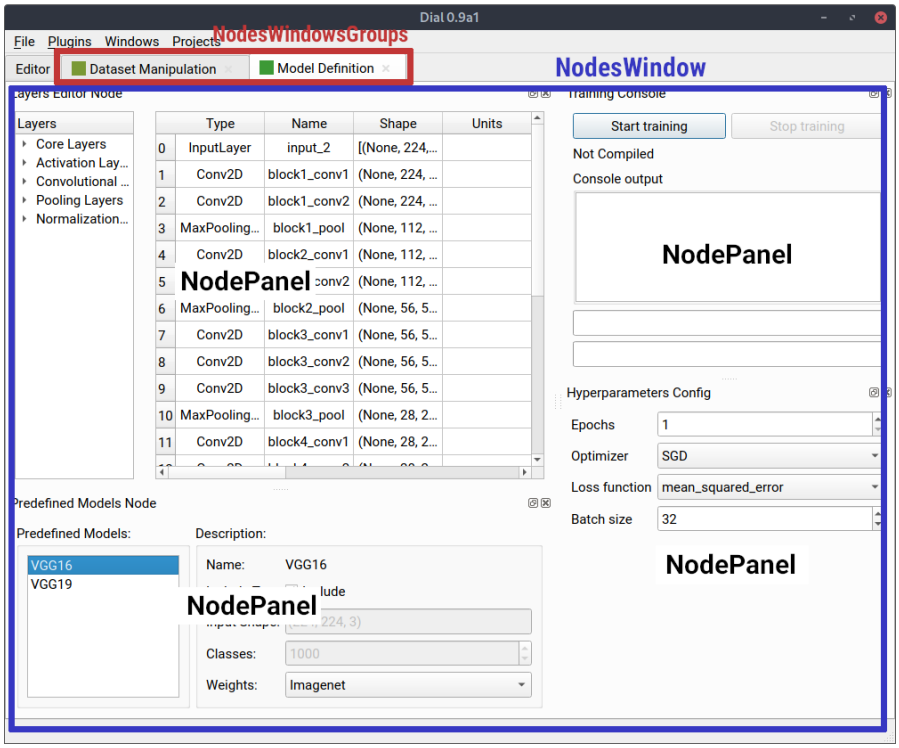

Figura 3.3: Descomposición de una ventana en dial-gui

Para terminar, podemos ver el código que define un nodo utilizado para tareas de Deep Learning. En este caso, el nodo se encargaría de importar datasets predefinidos en Keras (Listado 3.8).

Listado 3.8: Código de PredefinedTTVSetsNode

```
class PredefinedTTVSetsNode(Node):
    def __init__(self, predefined_ttv_sets_widget: "PredefinedTTVSetsWidget"):
        super().__init__(
            title="Predefined TTVSets Node", inner_widget=predefined_ttv_sets_widget
        )

        self.add_output_port(name="TTVSets", port_type=TTVSets)
        self.outputs["TTVSets"].set_generator_function(self.inner_widget.get_ttv)

        self.inner_widget.selected_ttv_loader_changed.connect(
            lambda: self.outputs["TTVSets"].send()
        )
```

En la clase PredefinedTTVSetsNode definimos la ”interfaz”, los puertos de entrada y salida que tendrá nuestro nodo. Además, podemos aprovecharnos del patrón de signal/slots en Qt para propagar datos a través de los puertos como respuestas a acciones del usuario en la interfaz. En el ejemplo anterior, la signal selected_ttv_loader_changed se emitiría cada vez que el usuario hace click a un dataset diferente de la lista. La respuesta sería propagar el dataset seleccionado al resto de nodos conectados para que se actualicen.

Uno de los argumentos es PredefinedTTVSetsWidget, clase que define los componentes que forman la interfaz mostrada dentro del widget y las signals a interceptar por el nodo (Listado 3.9).

Listado 3.9: Código de PredefinedTTVSetsWidget

```
class PredefinedTTVSetsWidget(QWidget):
    selected_ttv_loader_changed = Signal(TTVSetsLoader)

    def __init__(self, predefined_ttv_sets_window, parent: "QWidget" = None):
        super().__init__(parent)

        self._predefined_ttv_sets_window = predefined_ttv_sets_window

        self._main_layout = QVBoxLayout()
        self._main_layout.setContentsMargins(0, 0, 0, 0)
        self._main_layout.addWidget(predefined_ttv_sets_window)

        self._predefined_ttv_sets_window.selected_ttv_loader_changed.connect(
            lambda ttv_loader: self.selected_ttv_loader_changed.emit(ttv_loader)
        )

        self.setLayout(self._main_layout)

    def get_ttv(self) -> Optional["TTVSets"]:
        return self._predefined_ttv_sets_window.selected_ttv()
```

3.3. dial-basic-plugins

Si bien *dial-gui* es el paquete que define la interfaz gráfica, este no contiene ninguna funcionalidad en sí mismo. La herramienta sigue un sistema de plugins, por lo que todos los nodos se han implementado en un paquete separado (dial-basic-plugins) que dial-gui carga al iniciar el programa.

Para presentar y explicar los nodos que se han desarrollado seguiremos el siguiente ejemplo: Vamos a importar la arquitectura de la red neuronal VGG16 [28], y entrenarla para clasificar las imágenes del dataset CIFAR10 [2]. Además, exportaremos el modelo entrenado en un archivo .h5 (para poder importarlo en otras aplicaciones), y visualizaremos la precisión de nuestro modelo clasificando el dataset de testeo.

Todo este ejemplo queda contenido en el grafo de la Figura 3.4, que iremos viendo en detalle.

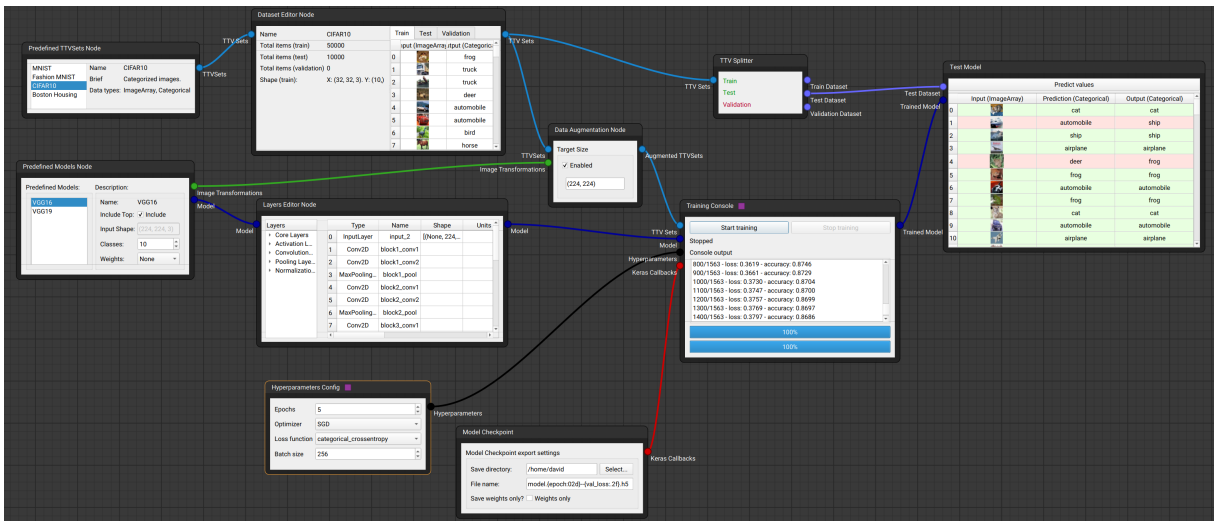

Figura 3.4: Grafo completo representando el ejemplo de VGG16 y CIFAR10

3.3.1. Predefined TTVSets Node

Este nodo (Figura 3.5) se encarga de importar TTVSets predefinidos por Keras (un TTVSet es como se denomina en el programa al conjunto de tres datasets: Train, Test y Validation). Son especialmente útiles para hacer pruebas sin necesidad de estar buscando e importando modelos de otras fuentes, como Kaggle [15].

Este nodo utiliza una clase llamada PredefinedTTVSetsContainer, que contiene todos los TTVSets que se hayan registrado. Por tanto, si otros plugins añadieran nuevos datasets predefinidos, se seguirían mostrando aquí.

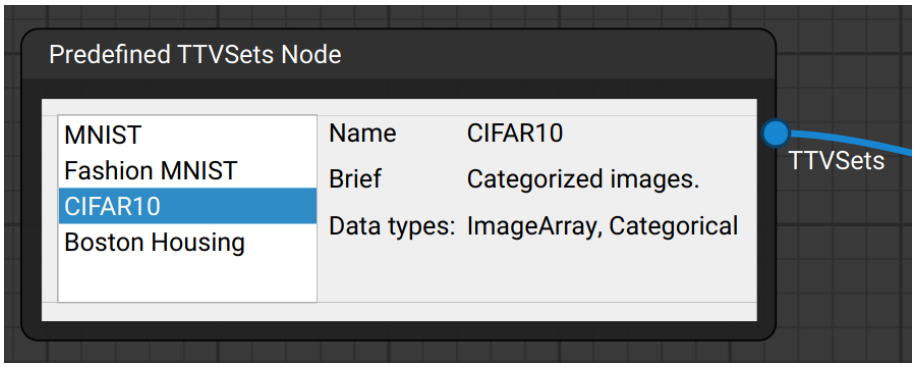

Figura 3.5: Captura del nodo "Predefined TTVSets"

3.3.2. Dataset Editor Node

El nodo ”Dataset Editor” permite visualizar el TTVSet recibido en el puerto de entrada (Figura 3.6). Separa el TTVSet en sus diferentes datasets, cada uno en una pestaña diferente (Train, Test y Validation), mostrando también varias estadísticas generales como el número de objetos o la ”shape” (estructura de los vectores) de los datos de entrada y salida.

También permite realizar algunas operaciones sencillas, como eliminar objetos del dataset, añadir objetos nuevos, o modificar los existentes. En nuestro ejemplo, podríamos corregir la categoría de alguna imagen que hayamos etiquetado mal para luego exportar el TTVSet corregido.

El puerto de salida para este nodo es el TTVSet modificado (o el mismo TTVSet si no se ha hecho ningún cambio).

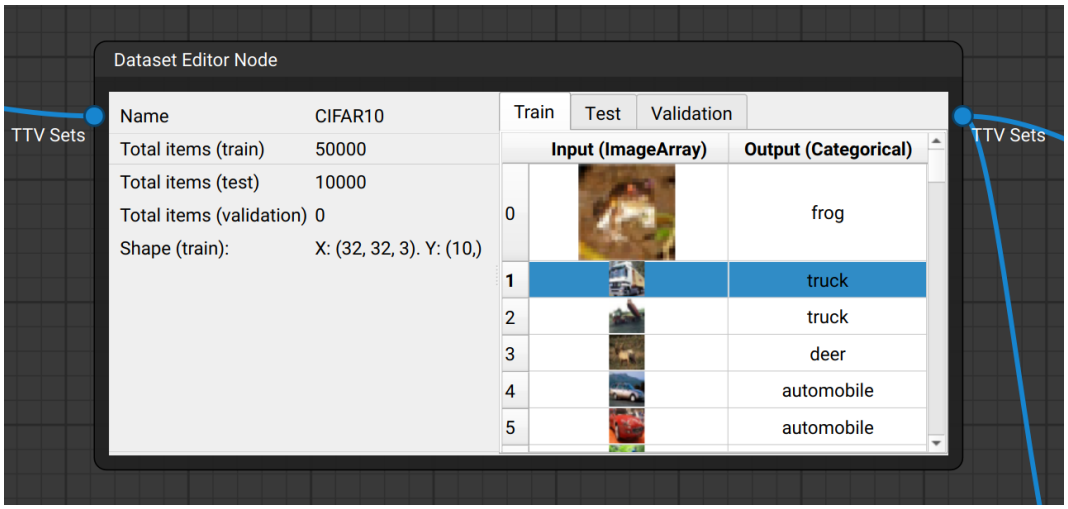

Figura 3.6: Captura del nodo "Dataset Editor"

3.3.3. Predefined Models Node

Al igual que el nodo ”Predefined TTVSets”, este nodo nos permite cargar redes neuronales predefinidas por Keras (Figura 3.7). Junto a cada red neuronal hay una serie de parámetros que se pueden configurar, como elegir si cargar la red preentrenada, o el tamaño de la primera capa (para ajustarla al tamaño de los objetos del dataset).

Este nodo también utiliza una clase (PredefinedModelsContainer) para mostrar los modelos que se hayan registrado en este. Por tanto, otros plugins podrían añadir más modelos predefinidos para que se muestren aquí.

Un detalle importante de este nodo es que uno de sus puertos de salida es del tipo ”Image Transformation”. Una ”Image Transformation” es una función que recibe una imagen como entrada, y realiza una serie de transformaciones (normalización, cambiar posición de los canales de color, representar cada píxel en la escala del 0.0 al 1.0 en lugar del 0 al 255...). Junto con la arquitectura del VGG16, podemos exportar también la función de transformación utilizada al entrenarla para transformar nuestras imágenes.

Si estamos utilizando la red neuronal preentrenada, es imprescindible asegurarnos de que las nuevas imágenes que utilicemos hayan recibido la misma transformación que las imágenes que se utilizaron durante el entrenamiento. En nuestro ejemplo, no es esencial ya que estamos utilizando la red neuronal sin preentrenar, pero de igual manera transformar las imágenes suele mejorar la precisión y resultados del modelo.

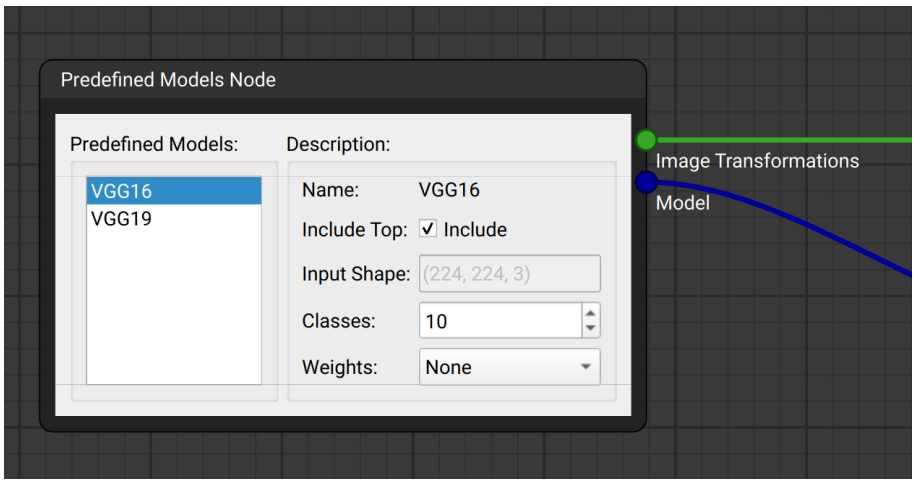

Figura 3.7: Captura del nodo "Predefined Models"

3.3.4. Layers Editor Node

Este nodo es un editor de las capas de una Red Neuronal. Permite tanto crear un nuevo modelo desde cero, arrastrando las capas que hay en el menú de la izquierda en la tabla, como modificar un modelo existente (para ello se debe especificar el modelo en el puerto de entrada). En nuestro ejemplo, podemos visualizar las diferentes capas que componen la red neuronal VGG16 (Figura 3.8).

Las diferentes columnas de la capa muestran los varios atributos de cada capa de la red neuronal, pudiéndose modificar en caso de que dicho atributo lo permita (por ejemplo, se pueden marcar que capas entrenar, útil para las técnicas de fine-tunning, o cambiar la función de activación para una determinada capa, el número de nodos en una capa densa, el tamaño del filtro para capas convolucionales...)

En el puerto de salida de este nodo obtendríamos el modelo compilado.

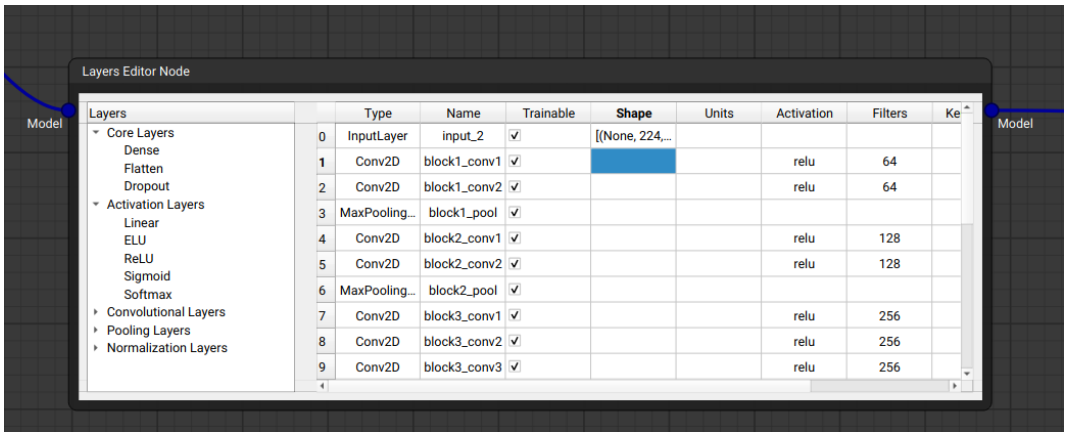

Figura 3.8: Captura del nodo "Layers Editor"

3.3.5. Hyperparameters Config Node

El nodo ”Hyperparameters Config” permite definir los hiperparámetros más comunes utilizados por Keras durante el entrenamiento de la red neuronal. Incluye todas las funciones de optimización y funciones de pérdida soportadas por Keras (Figura 3.9).

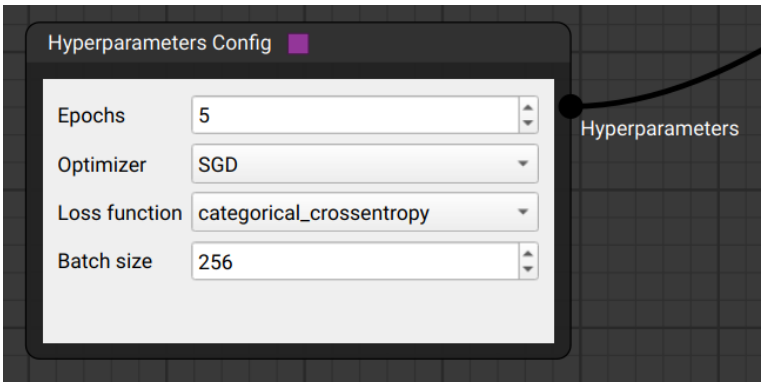

Figura 3.9: Captura del nodo "Hyperparameters Config"

3.3.6. Model Checkpoint Node

Este nodo es algo especial, pues su resultado es una ”callback” de Keras (Figura 3.10). Estas callbacks son, en sí mismas, funciones que se acoplan al entrenamiento de Keras y se ejecutan en un momento específico del mismo (antes/después de un epoch, antes/despues de un batch, justo al empezar el entrenamiento, justo al terminar...)

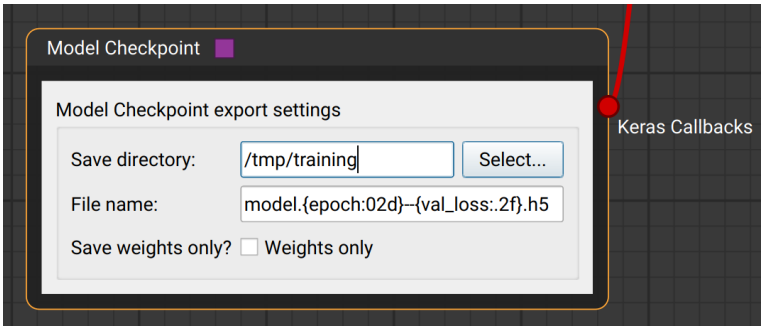

Figura 3.10: Captura del nodo "Model Checkpoint"

Por tanto, este nodo lo que nos permite es definir una callback que irá guardando el modelo entrenado tras cada epoch. De esta forma, podríamos cortar el entrenamiento a la mitad y quedarnos con el último modelo generado, sin necesidad de esperar a que el entrenamiento se complete.

Estos modelos se guardan en archivos con extensión .h5, que luego pueden ser importados fácilmente por Keras en otras aplicaciones (Listado 3.10). Además, se puede elegir si guardar solo los pesos o guardar también la arquitectura del modelo para poder cargarlo completamente más adelante.

Listado 3.10: Código para cargar un modelo .h5 en Keras

```
new_model = keras.models.load_model('path_to_my_model.h5')
```

3.3.7. Data Augmentation Node

Para entender este nodo hay que explicar primero qué se entiende por Data Augmentation. Esta práctica consiste en aplicar varias funciones de transformación a las imágenes del dataset. Normalmente, estas funciones se utilizan tanto para dar mas variedad al dataset (aplicar operaciones de espejo, zoom, o rotación, para que la red neuronal entrene con entradas más diversas), como para ajustar las imágenes al vector de entrada de la red neuronal (redimensionándolas, por ejemplo).

Este nodo aún está en desarrollo, por lo que por ahora solo permite redimensionar las imágenes. En nuestro ejemplo (Figura 3.11), lo utilizamos para agrandar las imágenes de CIFAR10 (32x32) al tamaño que espera VGG16 (224x224).

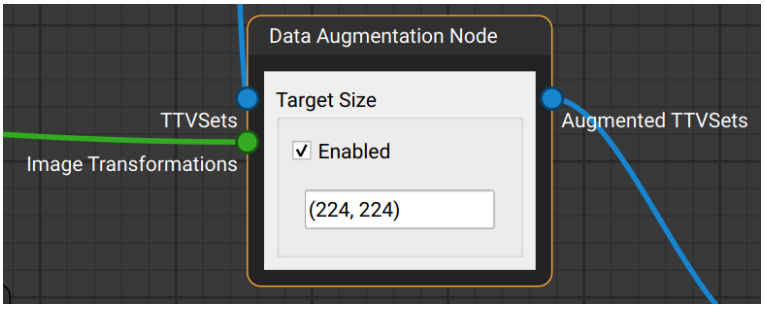

Figura 3.11: Captura del nodo "Data Augmentation"

3.3.8. Training Console Node

Este nodo es bastante sencillo: Recibe los datos necesarios para el entrenamiento mediante sus puertos de entrada (TTVSets, Model, Hyperparameters y, opcionalmente, Keras Callbacks) y muestra una barra con el progreso del entrenamiento (Figura 3.12). Cada varios batches, registra en la consola el epoch, valores de pérdida y precisión actuales. En caso de que hubiera algún error con los valores pasados mediante los puertos de entrada (algún dataset o modelo inválido, hiperparámetros no compatibles...) se mostraría el mensaje de error en la consola.

En el futuro se añadirá la opción de elegir qué valores incluir en la consola (aparte de las funciones de pérdida y precisión).

Una vez se haya completado el entrenamiento (o si se ha parado antes de terminarlo), el modelo entrenado estará disponible para usarse en el puerto de salida.

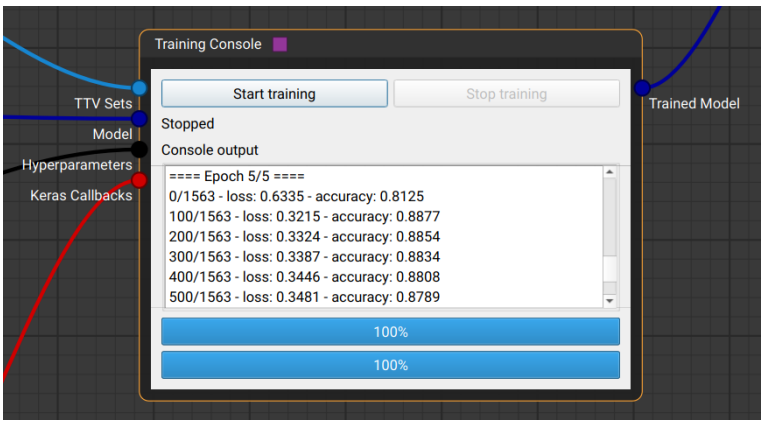

Figura 3.12: Captura del nodo "Training Console"

3.3.9. TTV Splitter Node

El único objetivo de este simple nodo es el de separar un TTVSet en sus tres datasets correspondientes: Train, Test y Validation. Marcará en rojo los datasets que sean nulos (y por tanto no se puedan exportar), y en verde los que sí (Figura 3.13).

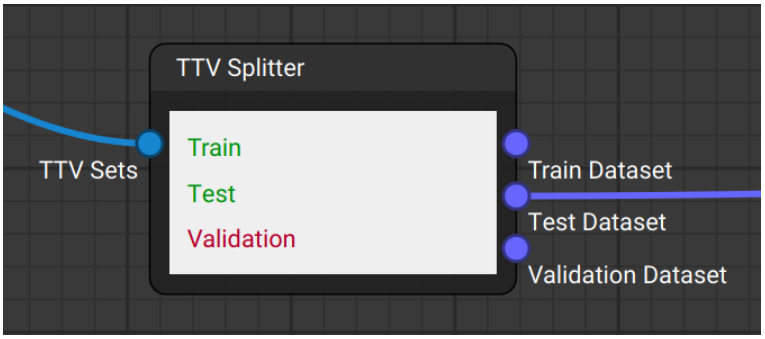

Figura 3.13: Captura del nodo "TTV Splitter"

Este nodo es necesario para otros nodos que tengan como entrada solo un dataset específico, y no todo el TTVSet.

Hay un equivalente a este nodo, TTV Merger, con el funcionamiento inverso: crear un TTVSet a partir de datasets individuales.

3.3.10. Test Model Node

Este nodo nos permite coger un modelo entrenado y un dataset de testeo, y visualizar qué categorías está clasificando bien o mal para los diferentes objetos del dataset. Las filas bien clasificadas aparecerán en verde, y las mal clasificadas, en rojo (Figura 3.14).

Este nodo aún está en desarrollo, en un futuro se añadirá la posibilidad de filtrar y solo mostrar las clases bien o mal clasificadas, así como estadísticas como el número o porcentaje de aciertos, o como la matriz de confusión de las clases.

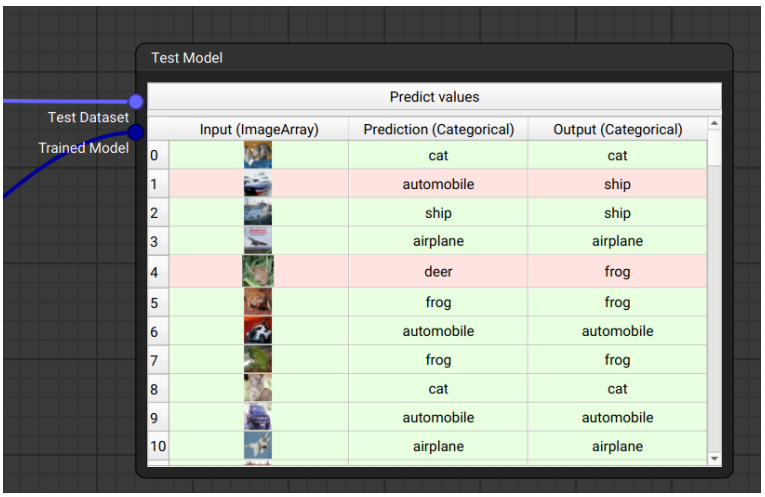

Figura 3.14: Captura del nodo "Test Model"

3.3.11. TTV Importer Node

Aunque este nodo no se ha incluido en el ejemplo, también merece la pena explicarlo, pues nos permitiría importar cualquier dataset que hayamos descargado y no solo los predefinidos de Keras (Figura 3.15).

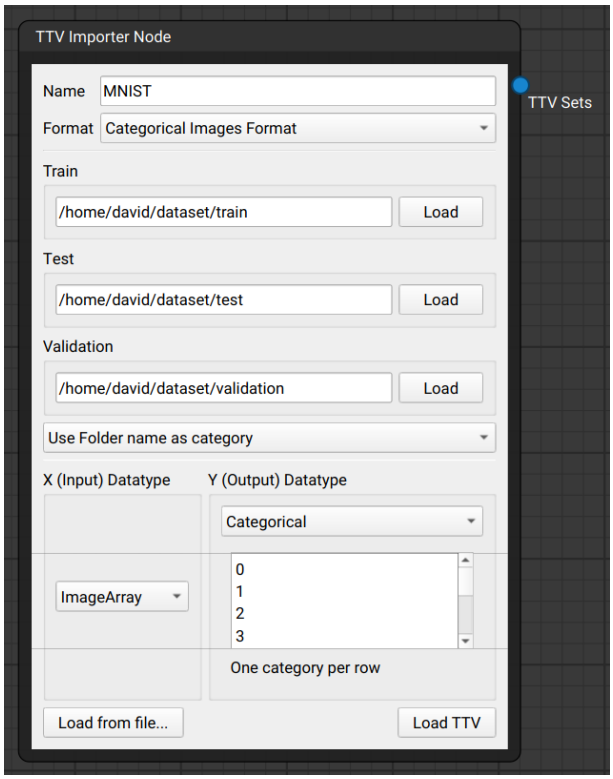

Figura 3.15: Captura del nodo "TTV Importer"

Por ahora, los datasets a cargar deben tener un formato y organización específico (que normalmente coincide con el formato en el que se distribuyen). Lo primero es que debemos tener una carpeta por cada dataset (train, test, validation). Luego, en el caso de un dataset de imágenes categóricas, podríamos:

1. Tener una carpeta por cada clase, y añadir las imágenes dentro de la carpeta. En este caso, se usaría el nombre de la carpeta como nombre de la clase.

2. Añadir todas las imágenes en la misma carpeta, y especificar la clase en el nombre de la imagen. Luego se definiría una expresión regular para extraer la parte correspondiente a la clase (por ejemplo, para una imagen llamada dog.000.jpg, la expresión regular (\w*).* nos permitiría extraer "dog". Este caso es más complejo, por lo que se recomienda seguir la organización de 1. Sin embargo, puede ser útil para importar datasets que ya tengan esta estructura.

Tras especificar en qué carpeta se encuentra cada dataset, y como están organizados en memoria, quedaría asociarlo con un tipo de dato. Para las imágenes categóricas, la entrada (X) del dataset es un array multidimensional (donde cada celda corresponde con el valor de un píxel de la imagen tras ser cargada en memoria), y la salida (Y) es una variable del tipo categórica que podrá tener el valor de cualquiera de las clases especificadas en la lista de debajo. Esta lista se rellenará automáticamente con las clases detectadas, pero también se puede modificar (para, por ejemplo, no cargar clases que no nos interesen)

Al final, todas estas opciones se representan internamente como un diccionario (JSON en archivos). Si se dispone de uno, se puede cargar directamente el TTVSet mediante el botón "Load from file..."(Listado 3.11).

Si estamos tratando con datasets que no están formados por imágenes, podemos importarlos si están guardados en archivos con formato .npz (numpy arrays)

Listado 3.11: Ejemplo de un json para cargar un TTVSet de ejemplo

```
{
    "name": "MNIST",
    "format": "Categorical␣Images␣Format",
    "train": {
        "x_type": {
            "class": "ImageArray"
        },
        "y_type": {
            "class": "Categorical",
            "categories": [
                "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
            ]
        },
        "organization": "CategoryOnFolders",
        "filename_category_regex": ""
    },
    "test": {
        "x_type": {
            "class": "ImageArray"
        },
        "y_type": {
            "class": "Categorical",
            "categories": [
                "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
            ]
```

```
        },
        "organization": "CategoryOnFolders",
        "filename_category_regex": ""
    },
    "validation": {}
}
```

De igual manera, existe también un nodo TTVSet Exporter que nos permite exportar un TTVSet a alguno de los formatos permitidos (Categorical Images o Npz). Junto a los datasets, nos creará el JSON para que podamos importarlo en el programa sin rellenar los datos manualmente (Figura 3.16).

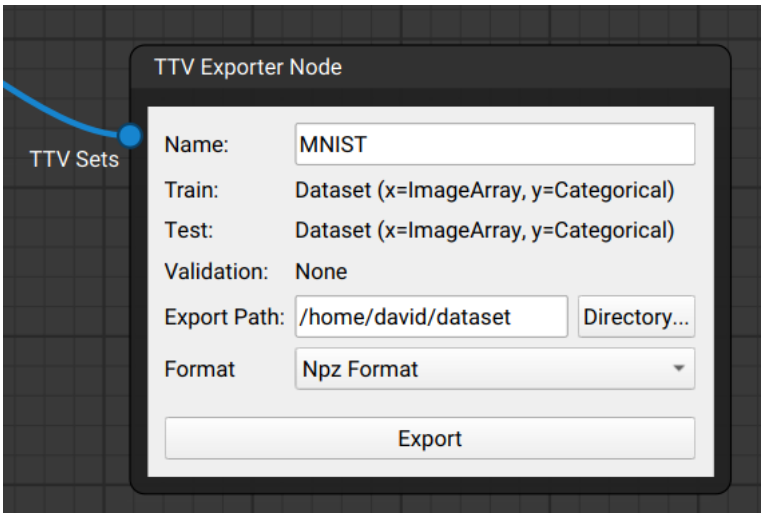

Figura 3.16: Captura del nodo "TTV Exporter"

Capítulo 4

Conclusiones y desarrollo futuro

4.1. Conclusiones

Tras haber finalizado la etapa de desarrollo correspondiente a este Trabajo de Fin de Grado, se puede concluir que el proyecto ha asentado una base suficiente para continuar el desarrollo en líneas de desarrollo futuras.

Surgieron algunas dificultades durante el desarrollo del proyecto, como el diseño de la arquitectura del código, tanto para dial-core como para dial-gui. Antes de llegar a decidir el diseño basado en nodos y puertos hubieron varias iteraciones del programa con una interfaz más tradicional. Cuando se decidió cambiar el diseño a uno basado en nodos, algunos componentes se pudieron reciclar, pero en gran parte se tuvo que empezar de nuevo por las considerables diferencias entre los dos acercamientos. Con este nuevo acercamiento también hubieron bastantes iteraciones hasta conseguir definir una estructura lo suficientemente buena. Por ejemplo, características inherentes a los grafos, como un ciclo, podían provocar un bucle infinito y congelar la interfaz gráfica. Este tipo de problemas se tenían que detectar en el código para evitar que el usuario tuviera una experiencia negativa.

4.2. Desarrollo futuro

Debido a la magnitud del proyecto, se han dejado varias líneas de desarrollo abiertas que serían especialmente interesantes de implementar en un futuro.

En primer lugar, para *dial-core*:

- **Completar el desarrollo de dial-core como API**: Inicialmente, *dial-core* se concibió como una librería que se utilizara internamente en *dial-gui*, pero no como un producto independiente. En las últimas etapas del desarrollo se vió el potencial que

tendría convertirlo en algo más parecido a una API: poder instanciar y utilizar los nodos directamente en código Python, además de en la interfaz gráfica. El principal potencial de esta característica sería el de traducir un grafo a código Python para, por ejemplo, incluirlo en un Jupyter Notebook.

- **Mejorar el procesamiento de datos en texto**: Si bien el framework actualmente soporta redes neuronales para problemas con datos textuales de entrada, se ha testeado más con imágenes de entrada y en problemas de clasificación. Sería beneficioso realizar pruebas resolviendo, por ejemplo, problemas de regresión con datos textuales para detectar posibles errores y oportunidades de plugins a desarrollar.

Por otra parte, para *dial-gui*:

- **Generación de Jupyter Notebooks a partir del grafo del proyecto**: Ya que *dial-gui* es una interfaz gráfica que se ejecuta en un ordenador local, puede ser que no se disponga en este de una GPU preparada para el entrenamiento de redes neuronales grandes. Una de las posibilidades para suplir esta desventaja podría ser implementar *Estrategias de Entrenamiento Distribuido en Tensorflow* [6], pero sería incluso más interesante poder convertir el grafo generado en dial-gui en un Jupyter Notebook. Este Notebook se podría luego subir a plataformas como Google Colab, que disponen de GPUs y TPUs preparadas para el entrenamiento de redes neuronales, y además hace mucho más sencillo el poder compartir los resultados del proyecto con otros usuarios que no utilicen *dial-gui*.

Y por último, se presentan varias ideas de plugins que se podrían desarrollar:

- **Detección de objetos en tiempo real con YOLO**: YOLO [30] es una de las redes neuronales con mejores resultados en clasificación y detección de ”bounding-box” (lugar de la imagen en el que aparece un objeto). Aunque originalmente depende de un framework en C llamado ”darknet” [4], hay implementaciones de este en Keras que se podrían utilizar para crear nodos de YOLO en la aplicación. Por ejemplo, un nodo para clasificar y definir el bounding box en las imágenes mediante el uso del ratón (más cómodo que especificar las coordenadas en texto), u otro nodo capaz de mostrar una entrada de vídeo (grabada, o en directo, como una cámara web) y realizar detección en tiempo real.
- **Redes neuronales generativas adversarias (GANs)**: Los modelos generativos son un tipo de red neuronal capaz de generar imágenes (por ejemplo, crear imágenes de satélite a partir de un mapa, o crear imágenes en color a partir de imágenes en blanco y negro). Crear una serie de nodos con redes generativas preentrenadas ofrecería una buena herramienta para prototipar rápidamente posibles aplicaciones para estas redes.

Capítulo 5

Summary and future lines

5.1. Summary

After finishing the development stage corresponding to this thesis, we can conclude that the project has established a firm ground for future development lines.

There were several complications while developing the project, like the code architecture design for *dial-core* and *dial-gui*. Before committing to the node-based design, we developed several iterations of the program with a traditional interface. When the design changed, only a few components could be recycled, and most of the interface was redone from scratch. But even with this new approach, we still had to do several iterations before reaching a solid design. For example, inherent characteristics of graphs, like cycles, could provoke an infinite loop and freeze the graphic interface. This kind of problems had to be detected and addressed on the code to avoid a bad user experience of the interface.

5.2. Future development

Due to the project magnitude, we left several development lines still open. Implementing them would be quite interesting and beneficial for the project.

On one hand, for *dial-core*:

- **Complete the development of dial-core as an API**: Initially, *dial-core* was created as an internal library used inside *dial-gui*, but not as an independent product. On the last stages of the development we saw the potential of transforming it into a proper API: Allowing nodes and graphs to be defined and used directly from Python code, and not only on the graphic interface. This would allow us to translate a dial graph into Python code that could be, for example, included on a Jupyter Notebook.
- **Improve the processing of text data**: The framework currently supports using

neural networks with textual input data, but most of the tests have been done with image-based neural networks. Some proper testing would be needed to detect possible bugs or plugin ideas.

On the other hand, for *dial-gui*:

- **Jupyter Notebook generation from node graphs**: Given that *dial-gui* is a graphical interface running on a local computer, it may be the case that we don't have access to a good GPU for training big models on that computer. A way to solve this could be implementing Tensorflow's Distributed Learning Strategies [6], but it would be even more interesting to transform the *dial-gui* graph into a Jupyer Notebooks. Notebooks can be uploaded to cloud training platforms like Google Colab, which have GPUs and TPUs ready for training neural networks, and they are also easier to share and show to other people.

Finally, we have some ideas for plugin opportunities:

- **Real time object detection with YOLO**: YOLO [30] is one of the neural networks with better results for "bounding-box"detection and object classification. Originally based on a C framework called Darknet [4], we could use one of the implementations based in Keras for new nodes. For example, a node for defining de bounding box on images using the mouse (which is easier than specifying the coordinades on text), or another node for displaying a video input (recorded, or live, like a webcam) and perform real-time detection.
- **Generative Adversarial Networks (GANs)**: Generative models are a new type of neural networks that can generate images (for example, create satellite images from a map, or create color images from black and white ones). Having nodes with pretrained neural networks would be a good tool for prototyping possible applications for these models.

Capítulo 6

Presupuesto

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.**

Actividad	Horas
Búsqueda de información	35h
Diseño de la arquitectura	40h
Desarrollo del proyecto	250h
Composición de la memoria con resultados	20h

Tabla 6.1: Resumen de tipos

Suponiendo un precio medio de **25€** la hora, el coste total del proyecto ascendería a los **8625€**

Bibliografía

[1] Blender. URL https://www.blender.org/.

[2] Cifar 10. URL https://www.cs.toronto.edu/~kriz/cifar.html.

[3] Codecov. URL https://codecov.io/.

[4] Darknet. URL https://github.com/AlexeyAB/darknet.

[5] Digits. URL https://developer.nvidia.com/digits.

[6] Distributed training with tensorflow. URL https://www.tensorflow.org/guide/distributed_training.

[7] Deep leaning studio. URL https://deepcognition.ai/.

[8] Deep learning market size, share and trends analysis report by solution, . URL https://www.grandviewresearch.com/industry-analysis/deep-learning-market.

[9] Deep learning for symbolic mathematics, . URL https://arxiv.org/pdf/1912.01412.pdf.

[10] Deep learning to improve breast cancer detection on screening mammography, . URL https://www.nature.com/articles/s41598-019-48995-4.pdf.

[11] Autonomous cars: Past, present and future, . URL https://www.researchgate.net/publication/283757446_Autonomous_Cars_Past_Present_and_Future_-_A_Review_of_the_Developments_in_the_Last_Century_the_Present_Scenario_and_the_Expected_Future_of_Autonomous_Vehicle_Technology.

[12] Neural machine translation: A review, . URL https://arxiv.org/pdf/1912.02047.pdf.

[13] Git, . URL https://git-scm.com/.

[14] Github, . URL https://github.com/.

[15] Kaggle. URL https://www.kaggle.com/.

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

[17] Inversion of control containers and the dependency injection pattern. URL https://martinfowler.com/articles/injection.html.

[18] Tiobe index for september 2020. URL https://www.tiobe.com/tiobe-index/.

[19] Neural network console - sony. URL https://dl.sony.com/.

[20] Pip. URL https://pypi.org/project/pip/.

[21] Pypi. URL https://pypi.org/.

[22] pytest, . URL https://docs.pytest.org/en/stable/.

[23] Python, . URL https://es.python.org/.

[24] Python end of life, . URL https://pythonclock.org/.

[25] Pytorch, . URL https://pytorch.org/.

[26] Tensorflow. URL https://www.tensorflow.org/.

[27] Travis ci. URL https://travis-ci.com/.

[28] Vgg16. URL https://neurohive.io/en/popular-networks/vgg16/.

[29] Weka. URL https://www.cs.waikato.ac.nz/ml/weka/.

[30] Yolo. URL https://pjreddie.com/darknet/yolo/.