**Curso profesional de Redes Neuronales con TensorFlow**

Ya conoces cómo funcionan las redes neuronales. Incrementa tus habilidades usando TensorFlow y todo su ecosistema de herramientas. Crea modelos de deep learning que podrás poner a funcionar en ambientes profesionales.

* Almacena modelos y reutilízalos.
* Utiliza modelos pre-entrenados con transfer learning.
* Optimiza la precisión de tus modelos de deep learning.
* Pre-procesa datos con Keras datasets y datasets generators.

**NOTA:**

Antes de continuar te invito a que revises los cursos anteriores:

* [1: Curso profesional de Redes Neuronales con TensorFlow](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/1%20Curso%20de%20fundamentos%20de%20redes%20neuronales)
* [2: Curso de Redes Neuronales Convolucionales con Python y keras](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/2%20Curso%20de%20Redes%20Neuronales%20Convolucionales)

Este Curso es el Número 3 de una ruta de Deep Learning, quizá algunos conceptos no vuelvan a ser definidos en este repositorio, por eso es indispensable que antes de empezar a leer esta guía hayas comprendido los temas vistos anteriormente.

Sin más por agregar disfruta de este curso

**Índice:**

* [1 Cómo utilizar TensorFlow 2.0 con Python](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#1-c%C3%B3mo-utilizar-tensorflow-20-con-python)
  + [1.1 Redes Neuronales con TensorFlow](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#11-redes-neuronales-con-tensorflow)
  + [1.2 Introducción a TensorFlow 2.0](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#12-introducci%C3%B3n-a-tensorflow-20)
* [2 Manejo y preprocesamiento de datos para redes neuronales](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#2-manejo-y-preprocesamiento-de-datos-para-redes-neuronales)
  + [2.1 Uso de data pipelines](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#21-uso-de-data-pipelines)
  + [2.2 Cómo cargar bases de datos JSON](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#22-c%C3%B3mo-cargar-bases-de-datos-json)
  + [2.3 Cargar bases de datos CSV y BASE 64](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#23-cargar-bases-de-datos-csv-y-base-64)
  + [2.4 Preprocesamiento y limpieza de datos](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#24-preprocesamiento-y-limpieza-de-datos)
  + [2.5 Keras datasets](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#25-keras-datasets)
  + [2.6 Datasets generators](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#26-datasets-generators)
  + [2.7 Aprende a buscar bases de datos para deep learning](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#27-aprende-a-buscar-bases-de-datos-para-deep-learning)
  + [2.8 Cómo distribuir los datos](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#28-c%C3%B3mo-distribuir-los-datos)
  + [2.9 Crear la red neuronal, definir capas, compilar, entrenar, evaluar y predicciones](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#29-crear-la-red-neuronal-definir-capas-compilar-entrenar-evaluar-y-predicciones)
* [3 Optimización de precisión de modelos](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#3-optimizaci%C3%B3n-de-precisi%C3%B3n-de-modelos)
  + [3.1 Métodos de regularización: overfitting y underfitting](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#31-m%C3%A9todos-de-regularizaci%C3%B3n--overfitting-y-underfitting)
  + [3.2 Recomendaciones prácticas para ajustar un modelo](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#32-recomendaciones-pr%C3%A1cticas-para-ajustar-un-modelo)
  + [3.3 Métricas para medir la eficiencia de un modelo: Callback](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#33-m%C3%A9tricas-para-medir-la-eficiencia-de-un-modelo--callback)
  + [3.4 Monitoreo del entrenamiento en tiempo real: early stopping y patience](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#34-monitoreo-del-entrenamiento-en-tiempo-real--early-stopping-y-patience)
  + [3.5 kerasTuner: Construyendo el modelo](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#35-kerastuner--construyendo-el-modelo)
  + [3.6 KerasTuner: Buscando la mejor configuración para tu modelo](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#36-kerastuner--buscando-la-mejor-configuraci%C3%B3n-para-tu-modelo)
* [4 Almacenamiento y carga de modelos](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#4-almacenamiento-y-carga-de-modelos)
  + [4.1 Almacenamiento y carga de modelos: pesos y arquitectura](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#41-almacenamiento-y-carga-de-modelos--pesos-y-arquitectura)
  + [4.2 Criterios para almacenar los modelos](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#42-criterios-para-almacenar-los-modelos)
* [5 Fundamentos de aprendizaje por transferencia](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#5-fundamentos-de-aprendizaje-por-transferencia)
  + [5.1 Introducción al aprendizaje por transferencia](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#51-introducci%C3%B3n-al-aprendizaje-por-transferencia)
  + [5.2 Cuándo utilizar aprendizaje por transferencia](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#52-cu%C3%A1ndo-utilizar-aprendizaje-por-transferencia)
  + [5.3 Carga de sistemas pre-entrenados en Keras](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#53-carga-de-sistemas-pre-entrenados-en-keras)
  + [5.4 API funcional de Keras](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#54-api-funcional-de-keras)
  + [5.5 Uso de sistemas pre-entrenados de TensorFlow Hub](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#55-uso-de-sistemas-pre-entrenados-de-tensorflow-hub)
* [6 Resultados de entrenamiento](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#6-resultados-de-entrenamiento)
  + [6.1 Introducción a variables relevantes del TensorBoard](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#61-introducci%C3%B3n-a-variables-relevantes-del-tensorboard)
  + [6.2 Análisis y publicación de resultados del entrenamiento](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#62-an%C3%A1lisis-y-publicaci%C3%B3n-de-resultados-del-entrenamiento)
  + [6.3 Introducción al despliegue de modelos en producción](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#63-introducci%C3%B3n-al-despliegue-de-modelos-en-producci%C3%B3n)
  + [6.4 Siguientes pasos con deep learning](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#64-siguientes-pasos-con-deep-learning)

**1 Cómo utilizar TensorFlow 2.0 con Python**

**1.1 Redes Neuronales con TensorFlow**

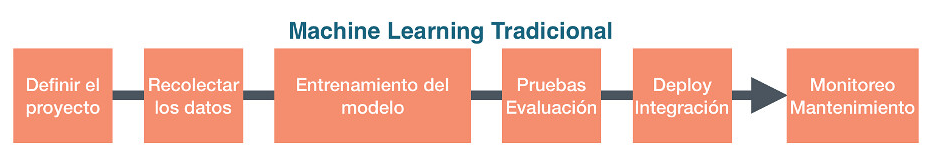
Los requisitos para poder tomar este curso (repositorio de github):

* Fundamentos de redes neuronales
* Uso de Entornos Virtuales en Python
* Creación de proyectos de ciencia de datos e inteligencia Artificial

**Ciclo de vida de un proyecto de Inteligencia Artificial**

El ciclo de vida de una IA (Inteligencia Artificial) es un proceso iterativo que consta de varias etapas y que tiene como objetivo crear y mejorar un sistema de IA a lo largo del tiempo. A continuación se describen las principales etapas del ciclo de vida de una IA:

1. Identificación del problema: se identifica el problema a resolver y se define el objetivo de la IA. Esta etapa implica entender el problema, definir las metas y los requisitos para el sistema de IA y determinar si la IA es la mejor solución para el problema.
2. Adquisición de datos: se recopila y prepara el conjunto de datos necesario para entrenar y validar el modelo de IA. Esta etapa implica identificar las fuentes de datos, recopilar los datos necesarios, limpiar y preprocesar los datos.
3. Preprocesamiento de datos: se realiza una exploración de los datos y se aplican técnicas de preprocesamiento para preparar los datos para su uso en el modelo de IA. Esto incluye la normalización, la reducción de dimensiones y la selección de características.
4. Desarrollo del modelo: se desarrolla el modelo de IA utilizando una arquitectura específica (por ejemplo, redes neuronales convolucionales para clasificación de imágenes). Esta etapa implica el entrenamiento, la validación y la optimización del modelo de IA.
5. Evaluación del modelo: se evalúa la precisión y el rendimiento del modelo de IA utilizando conjuntos de datos de prueba. Esta etapa implica la comparación de los resultados del modelo con los resultados esperados y la identificación de los posibles errores.
6. Implementación del modelo: se implementa el modelo de IA en un entorno de producción y se realiza un seguimiento continuo del rendimiento del modelo en tiempo real. Esto implica la integración con otros sistemas, la monitorización y la actualización del modelo.
7. Mantenimiento del modelo: se realiza un mantenimiento continuo del modelo de IA para garantizar su precisión y rendimiento en el tiempo. Esto incluye la actualización del modelo con nuevos datos, la optimización de la arquitectura y la resolución de problemas de calidad de datos.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F1%2F1.png)

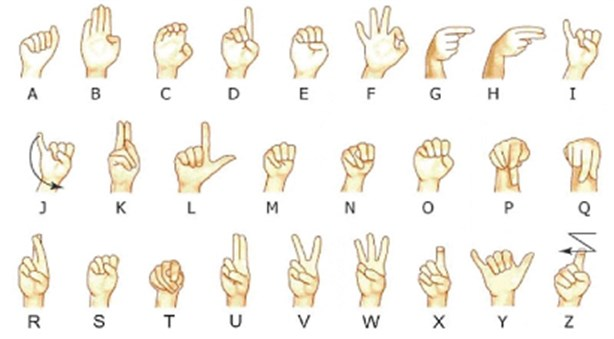
En resumen, el ciclo de vida de una IA es un proceso iterativo que involucra la identificación del problema, la adquisición y preparación de datos, el desarrollo y entrenamiento del modelo, la evaluación del modelo, la implementación del modelo en un entorno de producción y el mantenimiento continuo del modelo. Este ciclo permite crear y mejorar sistemas de IA de manera efectiva y eficiente.

**¿Qué vamos a hacer en este curso?**

1. Redes neuronales: Llevar redes neuronales a la práctica
2. Cargar bases de datos: conocer diferentes formatos de bases de datos
3. Optimizar modelos: Aumentar el accuracy y reducir la perdida
4. Evitar overfitting y underfittig
5. Transferencia de aprendizaje (transfer learning)
6. Almacenar y cargar modelos

**Proyecto principal del curso:**

El objetivo del curso será entrenar una CNN capas de detectar entre 24 letras del vocabulario de lenguaje de señas. (se evitará el reconocimiento de la J y la Z, pues son letras que necesitan del movimiento de la mano)

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F1%2F2.png)

La base de datos que se usará tiene las siguientes características:

* 27455 imágenes
* Escala de grises
* 28 x 28 píxeles
* 24 clases
* JPEG
* TecPerson - kaggle

**Objetivos del curso**

* Cómo cargar tus propias bases de datos
* Cargar bases de datos en formatos como CSV, JSON, BASE64, imágenes
* Aplicar técnicas para optimizar tus modelos
* Agregar métricas en el entrenamiento de tus modelos
* Cargar y guardar modelos
* Auto-tuner de Keras para encontrar mejores variables
* Bases de aprendizaje por transferencia
* Uso de TensorBoard y cómo mostrar tu proyecto al mundo entero
* Tener tu modelo listo para utilizarlo como inferencia

**Consiguiendo datos del proyecto de lenguaje de señas**

mkdir data

cd data

wget --no-check-certificate https://storage.googleapis.com/platzi-tf2/sign-language-img.zip -O sign-language-img.zip

unzip sign-language-img.zip

rm sign-language-img.zip

**Nota:**

Por temas de almacenamiento con GitHub la carpeta data ha sido añadida en el gitignore

**1.2 Introducción a TensorFlow 2.0**

Hay varias librerías de Deep Learning en Python. A continuación se mencionan algunas de las más populares:

1. TensorFlow: es una librería de Deep Learning desarrollada por Google. Es muy utilizada en aplicaciones de visión por computadora, procesamiento de lenguaje natural y otros campos del aprendizaje automático. TensorFlow es conocida por su escalabilidad y su capacidad para trabajar con grandes conjuntos de datos.
2. Keras: es una librería de alto nivel para el desarrollo de modelos de Deep Learning. Keras se enfoca en la simplicidad y facilidad de uso, permitiendo a los desarrolladores crear modelos de IA con pocas líneas de código. Keras también es compatible con TensorFlow y otras librerías de Deep Learning.
3. PyTorch: es una librería de aprendizaje profundo desarrollada por Facebook. PyTorch es conocida por su facilidad de uso y su capacidad para crear modelos de IA en tiempo real. Es muy popular en el ámbito de la investigación y se utiliza en aplicaciones de visión por computadora, procesamiento de lenguaje natural y otros campos.
4. Theano: es una librería de aprendizaje profundo que permite la definición, optimización y evaluación de expresiones matemáticas que involucran matrices multidimensionales. Theano es conocida por su rapidez y eficiencia en la realización de cálculos matemáticos.
5. Caffe: es una librería de aprendizaje profundo que se utiliza principalmente en aplicaciones de visión por computadora. Es conocida por su velocidad y eficiencia, lo que la hace muy útil en aplicaciones en tiempo real.
6. MXNet: es una librería de aprendizaje profundo desarrollada por Amazon. MXNet es conocida por su capacidad de escalabilidad y su eficiencia en la utilización de múltiples procesadores. Es muy popular en aplicaciones de visión por computadora, procesamiento de lenguaje natural y otros campos.

Para este curso nos enfocaremos principalmente en Tensorflow 2.0 + Keras

Sin embargo, Tensorflow No es solo una librería de python es un ecosistema general:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F1%2F3.png)

Tensorflow nos apoya a llevar el deploy de nuestros modelos no solo en computadoras, sino también en otras plataformas como celulares, páginas web, crear API, utilizar Arduinos, o cloud computing.

Para más información puedes visitar: <https://www.tensorflow.org/api_docs>

**2 Manejo y preprocesamiento de datos para redes neuronales**

En este módulo aprenderemos:

* Cómo cargar bases de datos en formatos CSV, JSON, BASE64, etc
* Pre-procesar los datos
* Cómo cargar datasets de keras
* Dataset Generators
* Cómo cargar tus propios datasets con TF.data
* Cómo distribuir los datos

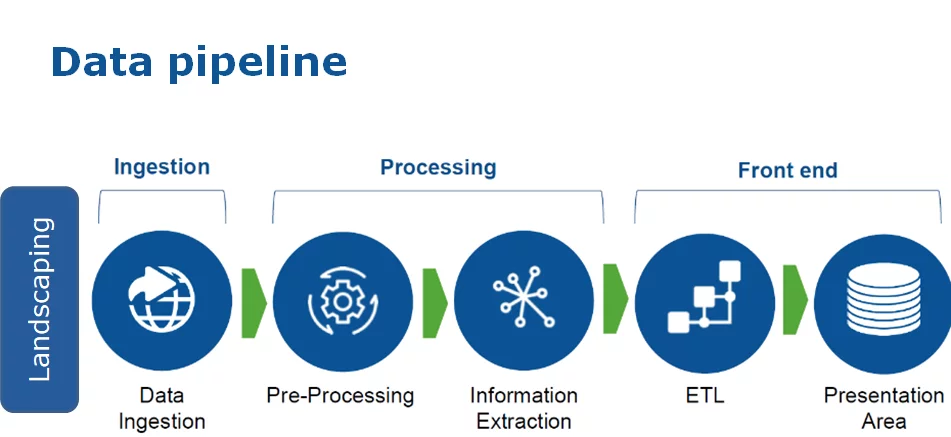
**2.1 Uso de data pipelines**

Un Data Pipeline en Python es un proceso automatizado que permite la ingestión, procesamiento, transformación y almacenamiento de datos en una secuencia ordenada de pasos. El Data Pipeline en Python se utiliza para automatizar el flujo de datos de una fuente a otra, como por ejemplo, desde una base de datos hasta un modelo de aprendizaje automático o un sistema de visualización de datos.

El Data Pipeline en Python generalmente se compone de varios pasos. Estos pasos incluyen la lectura de los datos de una fuente de datos, la limpieza y preprocesamiento de los datos, la transformación de los datos en un formato adecuado para su uso en el modelo de aprendizaje automático, el entrenamiento del modelo, la evaluación del modelo y la visualización de los resultados.

Para implementar un Data Pipeline en Python, se pueden utilizar varias herramientas y librerías, tales como:

* Pandas: es una librería de Python que se utiliza para el análisis de datos y el procesamiento de datos en memoria. Pandas ofrece funciones para la limpieza, transformación y filtrado de datos.
* NumPy: es una librería de Python que se utiliza para el procesamiento numérico. NumPy ofrece una gran cantidad de funciones matemáticas para el procesamiento de datos.
* Scikit-Learn: es una librería de Python que se utiliza para el aprendizaje automático. Scikit-Learn ofrece una gran cantidad de algoritmos de aprendizaje automático para la clasificación, regresión y clustering.
* TensorFlow: es una librería de Python que se utiliza para el aprendizaje profundo. TensorFlow ofrece una gran cantidad de herramientas para el desarrollo de modelos de aprendizaje profundo.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F1.png)

**Basura que entra basura que sale**

El rendimiento de que tan bueno puede ser cualquier modelo de machine learning o deep learning para clasificar un problema se ve determinado mayoritariamente por la calidad de la base de datos con la que fue entrenado. Si la base, no es buena, entonces es mejor buscar una base diferente o intentar limpiar lo mejor posible dicha base de datos.

Es muy importante contar con etiquetas de calidad que realmente reflejen el comportamiento deseado por el modelo de clasificación. Es relevante lidiar con los problemas que conlleva crear una buena base de datos como: lidiar con los datos perdidos, con los valores atípicos o con datos corruptos. Construir un conjunto de datos adecuado y de alta calidad es fundamental para resolver cualquier problema de ML o DL a continuación enlisto algunos consejos a tener en cuenta:

1. Definir claramente el objetivo del modelo: Es importante tener claro cuál es el objetivo del modelo de machine learning, ya que esto permitirá determinar los datos necesarios para el entrenamiento. Por ejemplo, si el objetivo es predecir el riesgo de fraude de una transacción, los datos necesarios podrían incluir información de transacciones previas, información financiera y datos de perfil de los usuarios.
2. Recolectar una cantidad suficiente de datos: Es importante tener suficientes datos para el entrenamiento del modelo. La cantidad de datos requeridos depende de la complejidad del problema y la cantidad de características necesarias para el entrenamiento. En general, se recomienda tener al menos unas miles de muestras para entrenar un modelo de machine learning.
3. Seleccionar características relevantes: Es importante elegir características relevantes para el modelo. Las características irrelevantes pueden agregar ruido y afectar el rendimiento del modelo. Se debe tener cuidado en seleccionar las características que sean más relevantes para el modelo y descartar aquellas que no sean útiles.
4. Limpiar y preprocesar los datos: Es importante limpiar y preprocesar los datos para eliminar datos incompletos, inconsistentes y ruidosos. Además, el preprocesamiento de datos puede incluir normalización, escalamiento y codificación de variables categóricas.
5. Verificar la calidad de los datos: Es importante verificar la calidad de los datos antes de entrenar un modelo de machine learning. Esto incluye comprobar la consistencia de los datos, la distribución de las características y la presencia de valores atípicos o datos faltantes.
6. Considerar el desequilibrio de clases: Si el conjunto de datos está desequilibrado (es decir, hay muchas más instancias de una clase que de otra), es importante considerar técnicas para manejar el desequilibrio. Esto puede incluir técnicas de submuestreo, sobremuestreo o ajuste de pesos.
7. Realizar una validación cruzada: Es importante realizar una validación cruzada para evaluar el rendimiento del modelo en un conjunto de datos que no se utilizó en el entrenamiento. Esto ayuda a evitar el sobreajuste del modelo.

Si tienes algunas dudas con los puntos anteriormente mencionados, también te invito a que le eches un vistazo a mi repositorio de [introducción a machine learning con scikit-learn](https://github.com/ichcanziho/cursos_platzi/tree/master/machine_learning_scikit_learn)

**2.2 Cómo cargar bases de datos JSON**

En este escenario vamos a utilizar el formato json que contiene urls almacenadas en la nube de gcp. Es importante conocer la mayor cantidad de herramientas que nos permitan cargar y manipular datos de todas las fuentes posibles. Muchas empresas cargan sus imágenes en nubes como AMAZON, GCP, AZURE, entre otras y luego estas son consumidas.

**Nota:**

El código de esta sección lo puedes encontrar [aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos/2%20C%C3%B3mo%20cargar%20datasets%20json/read_json.py)

Vamos a descargar las bases de datos que ocuparemos para esta clase y la siguiente:

cd 2\ Manejo\ y\ preprocesamiento\ de\ datos/

mkdir datasets

cd datasets

wget --no-check-certificate https://storage.googleapis.com/platzi-tf2/sign-language-img.zip -O sign-language-img.zip --no-check-certificate https://storage.googleapis.com/platzi-tf2/databasesLoadData.zip -O databasesLoadData.zip

unzip databasesLoadData.zip

rm databasesLoadData.zip

La respuesta esperada es la siguiente estructura de carpetas:

datasets

--------|sign\_mnist\_base64

--------|----------------|data.json

--------|sign\_mnist\_json

--------|----------------|data.json

--------|sign\_mnist\_test

--------|----------------|sign\_mnist\_test.csv

--------|sign\_mnist\_train

--------|----------------|sing\_mnist\_train.csv

--------|----------------|sing\_mnist\_train\_clean.csv

--------|----------------|sing\_mnist\_train\_no\_clean.csv

pixeles.png

Analicemos brevemente el archivo [data.json](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2Fdatasets%2Fsign_mnist_json%2Fdata.json)

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/29\_B.jpg","label":"b"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/30\_B.jpg","label":"b"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/95\_B.jpg","label":"b"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/58\_A.jpg","label":"a"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/50\_A.jpg","label":"a"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/46\_A.jpg","label":"a"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/3\_C.jpg","label":"c"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/32\_C.jpg","label":"c"}

{"content": "https://storage.googleapis.com/platzi-tf2/img\_mnist/2\_C.jpg","label":"c"}

Básicamente, es una secuencia de diccionarios, que tiene como llaves: content para mostrar el URL de la imagen y label el cual contiene la clasificación de dicha imagen. Observamos que como tal NO es el formato más correcto de json, puesto que para que puediera ser cargado por json.load el archivo debería indicar que es una lista con [] y cada elemento debería ir separado por ,, entonces debemos leer este archivo de una forma ligeramente diferente.

**Ejemplo en código:**

El flujo más simple de trabajo para este ejemplo es:

1. Leer el archivo [data.json](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2Fdatasets%2Fsign_mnist_json%2Fdata.json)
2. Para cada url de content:
   1. hacer un request para recibir la imagen
   2. transformar el response en un numpy array
3. Mostrar un ejemplo de la imagen y el label recibido

**1: Importamos bibliotecas necesarias**

import requests

from json import loads

from io import BytesIO

from PIL import Image

import numpy as np

import matplotlib.pyplot as plt

**2: Leemos el archivo data.json**

Lo podemos leer como un texto plano, y por cada línea dentro del archivo decodificar dicha linea como json ocupando su método loads

with open("../datasets/sign\_mnist\_json/data.json", "r", encoding='utf-8') as d:

data = [loads(line) for line in d.readlines()]

**3: Acceder a cada elemento dentro de data**

Como ahora cada línea es un archivo json, puedo acceder al valor de sus llaves utilizando get

X, y = [], []

for example in data:

print(example)

url\_image = example.get("content", 0)

label = example.get("label", 0)

Respuesta esperada:

{'content': 'https://storage.googleapis.com/platzi-tf2/img\_mnist/29\_B.jpg', 'label': 'b'}

**4: Descargando cada imagen y convirtiéndola a un numpy array**

# Petición al servidor

response = requests.get(url\_image).content

print(type(response), response)

# transformado `bytes` en PIL image

pil\_image = Image.open(BytesIO(response))

print(pil\_image)

# transformando pil\_image en un numpy array

img = np.asarray(pil\_image).reshape(28, 28)

X.append(img)

y.append(label)

Aquí adicionalmente, añadimos a las listas X, y los valores decodificados de las imágenes y labels pertinentes.

**5: Mostrando un ejemplo de imagen clasificada**

plt.imshow(img, cmap="gray")

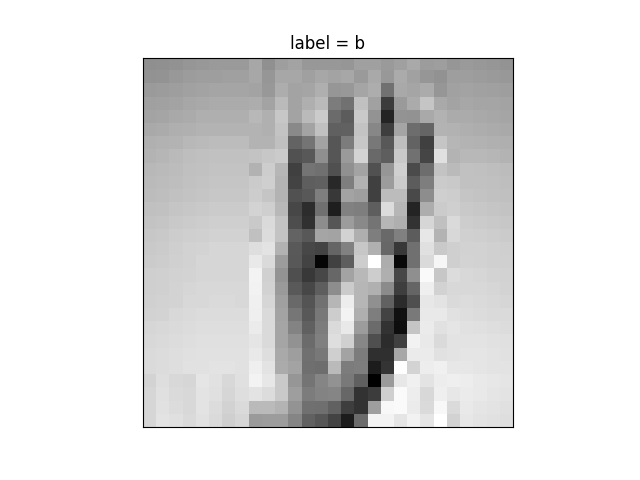
plt.title(f"label = {label}")

plt.xticks([])

plt.yticks([])

plt.savefig("test.png")

Respuesta esperada:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F2%20C%C3%B3mo%20cargar%20datasets%20json%2Ftest.png)

**Nota:**

El código de esta sección lo puedes encontrar [aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos/2%20C%C3%B3mo%20cargar%20datasets%20json/read_json.py)

**2.3 Cargar bases de datos CSV y BASE 64**

**Ejemplo de Base 64**

Primero entendamos qué es Base 64:

Base64 es una forma de codificar datos binarios en caracteres ASCII para que puedan ser transmitidos a través de canales que no admiten datos binarios, como el correo electrónico o la web. En Base64, cada conjunto de tres bytes (24 bits) se convierte en una cadena de cuatro caracteres ASCII.

Base64 se utiliza para enviar archivos adjuntos de correo electrónico, imágenes y otros tipos de datos a través de Internet. Por ejemplo, si tienes una imagen en formato binario, puedes convertirla en Base64 y luego enviarla en un correo electrónico como una cadena de texto. Cuando el destinatario recibe el correo electrónico, puede decodificar la cadena Base64 y obtener la imagen original.

Base64 es útil en situaciones donde los datos binarios no pueden ser transmitidos directamente. Sin embargo, es importante tener en cuenta que la codificación Base64 aumenta el tamaño de los datos en aproximadamente un tercio. Además, Base64 no proporciona ningún tipo de encriptación o seguridad, por lo que no se debe utilizar como una forma de proteger datos sensibles.

**Nota:**

Por experiencia laboral Base64 también es un formato muy útil para enviar imágenes a ser procesadas por un API, tiene sentido que esta codificación b64 sea utilizada en un endpoint para analizar dicha imagen.

Ahora que ya entendemos un poco más acerca de Base 64, veamos un ejemplo de nuestra base de datos [data.json](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2Fdatasets%2Fsign_mnist_base64%2Fdata.json):

{

"b": ""

}

Nuestro dataset, vuelve a ser un archivo en formato json, sin embargo, este es diferente al anterior, ahora en lugar de tener como valores de las llaves la dirección de la imagen y su label, ahora tenemos algo más simplificado, tenemos únicamente como llave la label de la imagen y como valor la propia imagen pero en formato b64.

**Ejemplo en código**

**Nota:**

El código de esta sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F2%20C%C3%B3mo%20cargar%20datasets%20json%2Fread_b64.py)

Cómo ejemplo escalable ocuparemos el siguiente dataset [data2.json](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2Fdatasets%2Fsign_mnist_base64%2Fdata2.json)

Es básicamente el mismo que el anterior, pero tiene varios datos en lugar de uno solo:

[

{"b": ""},

{"b": ""}

]

**1: Importando bibliotecas necesarias**

Vamos a utilizar OpenCv para acceder a un método muy útil imdecode que me permitirá junto con numpy convertir un texto en formato b64 a un numpy array

from json import load

import base64

import matplotlib.pyplot as plt

import numpy as np

import cv2 as cv

**2: Creamos función auxiliar de conversión b64 a numpy.array**

def b64\_to\_np(b\_string: str):

jpg\_original = base64.b64decode(b\_string)

jpg\_as\_np = np.frombuffer(jpg\_original, dtype=np.uint8)

image\_buffer = cv.imdecode(jpg\_as\_np, flags=1)

return image\_buffer

**3: Leemos el archivo** [data2.json](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2Fdatasets%2Fsign_mnist_base64%2Fdata2.json)

with open("../datasets/sign\_mnist\_base64/data2.json", "r", encoding="utf-8") as d:

data = load(d)

**4: Convertimos cada b64 en un numpy array**

X, y = [], []

for example in data:

for label, b\_image in example.items():

print(label, "-", b\_image)

img = b64\_to\_np(b\_image)

X.append(img)

y.append(label)

Respuesta esperada

b - 

**5: graficamos para observar el resultado**

plt.imshow(img, cmap="gray")

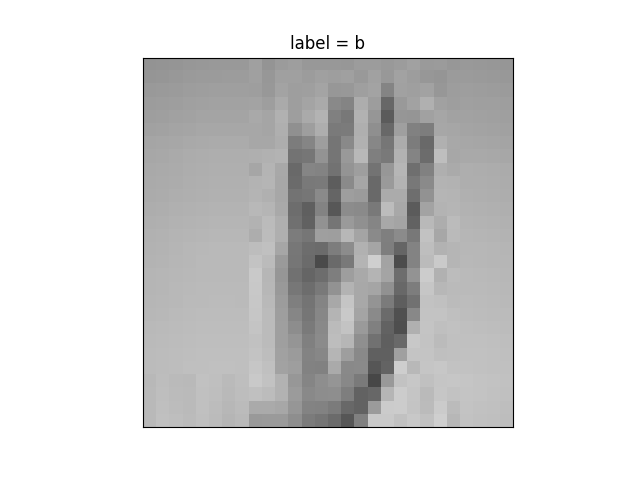
plt.title(f"label = {label}")

plt.xticks([])

plt.yticks([])

plt.savefig("test\_b64.png")

Respuesta esperada:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F2%20C%C3%B3mo%20cargar%20datasets%20json%2Ftest_b64.png)

**Ejemplo de CSV**

**Nota**

El código de esta sección lo puedes encontrar [aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F2%20C%C3%B3mo%20cargar%20datasets%20json%2Fread_csv.py)

Antes de continuar con este tema te recomiendo repasar la clase de: [Consejos para el manejo de imágenes](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/2%20Curso%20de%20Redes%20Neuronales%20Convolucionales#31-consejos-para-el-manejo-de-im%C3%A1genes) Del curso anterior. Por si note queda muy claro el código presente en este ejemplo.

Primero familiaricémonos con nuestros datos [sign\_mnist\_test.csv](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2Fdatasets%2Fsign_mnist_test%2Fsign_mnist_test.csv)

**1: Conozcamos los datos**

data = pd.read\_csv("../datasets/sign\_mnist\_train/sign\_mnist\_train.csv")

samples = len(data)

print("samples:", samples)

print(data)

Respuesta esperada:

samples: 27455

label pixel1 pixel2 pixel3 ... pixel781 pixel782 pixel783 pixel784

0 3 107 118 127 ... 206 204 203 202

1 6 155 157 156 ... 175 103 135 149

2 2 187 188 188 ... 198 195 194 195

3 2 211 211 212 ... 225 222 229 163

4 13 164 167 170 ... 157 163 164 179

... ... ... ... ... ... ... ... ... ...

27450 13 189 189 190 ... 234 200 222 225

27451 23 151 154 157 ... 195 195 195 194

27452 18 174 174 174 ... 203 202 200 200

27453 17 177 181 184 ... 47 64 87 93

27454 23 179 180 180 ... 197 205 209 215

[27455 rows x 785 columns]

Podemos observar que la primera columna contiene un label encoding de las clases. Mientras que las siguientes columnas son el flatten de la imagen (28x28) píxeles.

**2. Separamos el dataset en X & y**

y = data["label"].values

X = data.drop('label', axis=1).values.reshape((samples, 28, 28))

En realidad es todo, pandas almacena los datos como numpy arrays, solo necesitamos acceder a ellos y hacerles un reshape indicándole que tenemos n imágenes correspondientes al valor de samples y que cada imagen es de (28x28)

**3. Graficamos**

plt.imshow(img, cmap="gray")

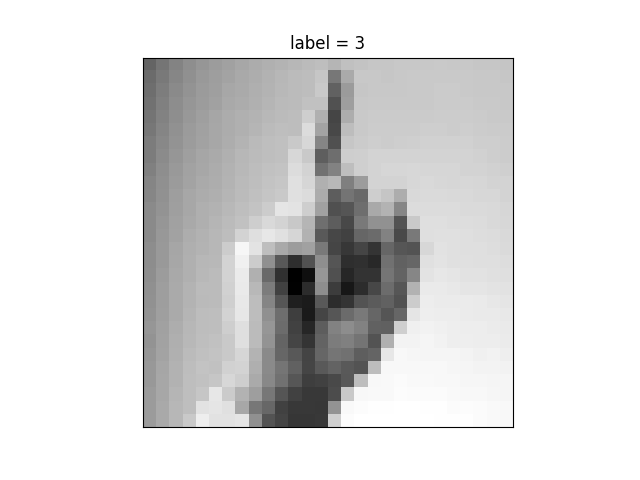
plt.title(f"label = {label}")

plt.xticks([])

plt.yticks([])

plt.savefig("test\_csv.png")

Respuesta esperada:

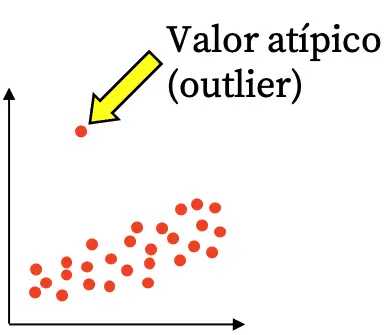
[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F2%20C%C3%B3mo%20cargar%20datasets%20json%2Ftest_csv.png)

**2.4 Preprocesamiento y limpieza de datos**

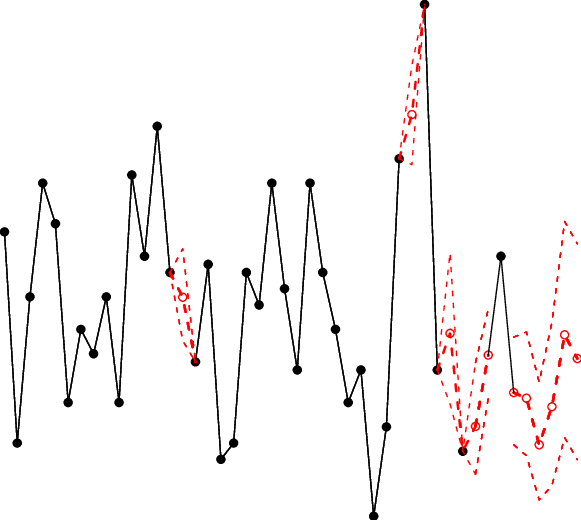
El preprocesamiento y la limpieza de datos son procesos fundamentales en el análisis de datos, en particular, cuando se trabaja con conjuntos de datos para machine learning. El preprocesamiento se refiere a la transformación de datos brutos en una forma que sea adecuada para el análisis y la modelización. La limpieza de datos es un paso específico del preprocesamiento que se enfoca en detectar y corregir errores en los datos.

A continuación, se describen algunos de los procesos comunes de preprocesamiento y limpieza de datos:

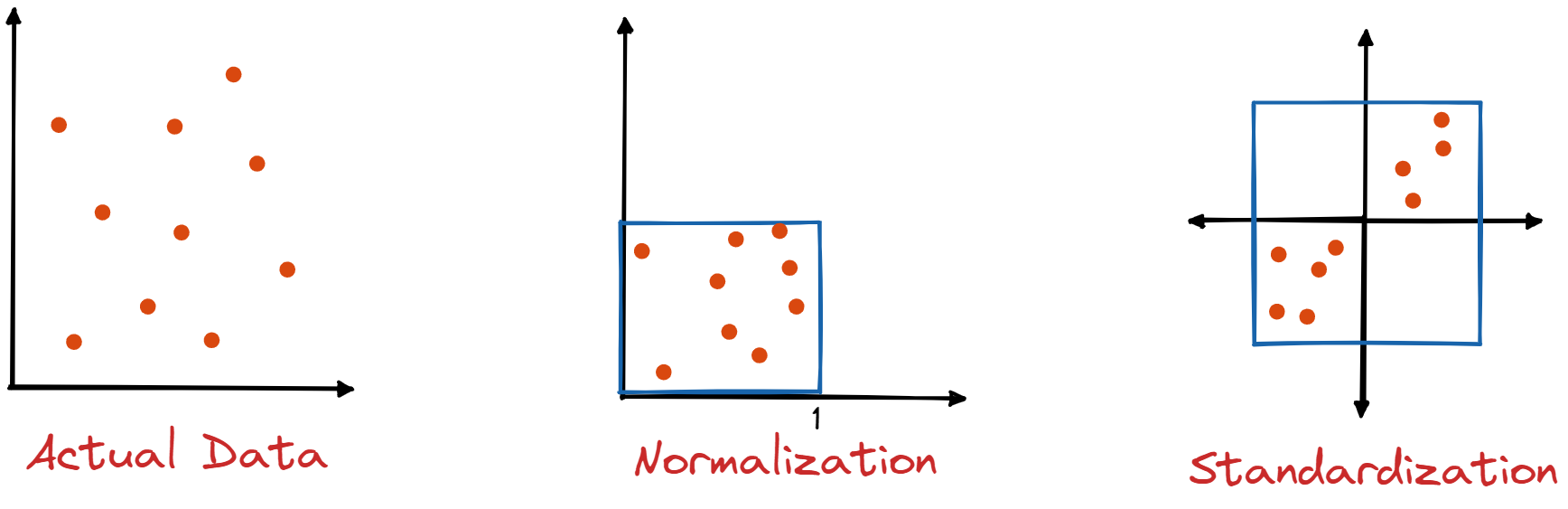
* Eliminación de valores atípicos: Los valores atípicos outliers son valores que se encuentran fuera del rango esperado de una distribución. Pueden ser causados por errores de medición o entrada de datos incorrecta. Los valores atípicos pueden distorsionar los resultados del análisis y por lo tanto, deben ser identificados y eliminados o corregidos si es posible.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F2.png)

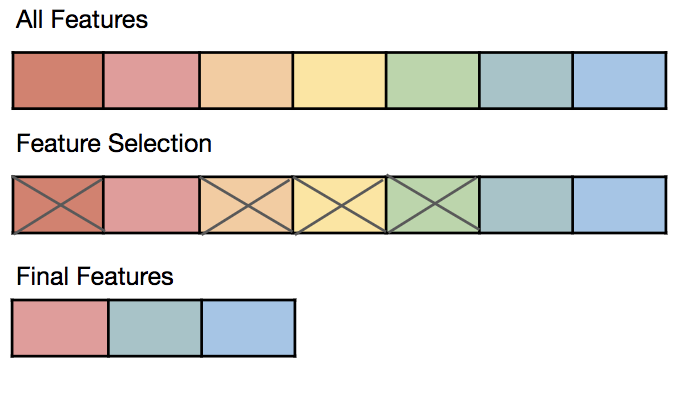
* Imputación de valores faltantes: Los valores faltantes pueden ser causados por diversas razones, como errores de medición o problemas de entrada de datos. La imputación se refiere a la estimación de valores faltantes en el conjunto de datos utilizando técnicas como la media, la mediana, la moda, la regresión u otros modelos estadísticos.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F3.png)

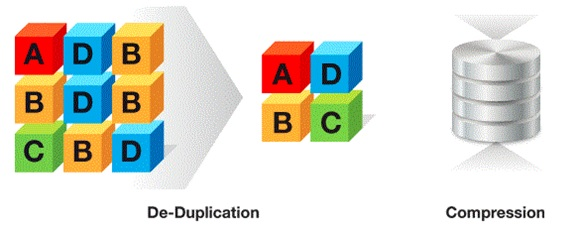
* Normalización y estandarización: Normalización se refiere al proceso de escalar los valores de una variable para que tengan una escala común. La estandarización, por otro lado, es el proceso de transformar los datos para que tengan una media cero y una desviación estándar de uno. La normalización y la estandarización se utilizan para que las variables tengan escalas similares y para mejorar el rendimiento de los modelos de machine learning.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F4.png)

* Eliminación de variables irrelevantes: Las variables irrelevantes son aquellas que no contribuyen significativamente al análisis y por lo tanto, se pueden eliminar. Eliminar variables irrelevantes puede mejorar la precisión del análisis y reducir el tiempo de procesamiento.

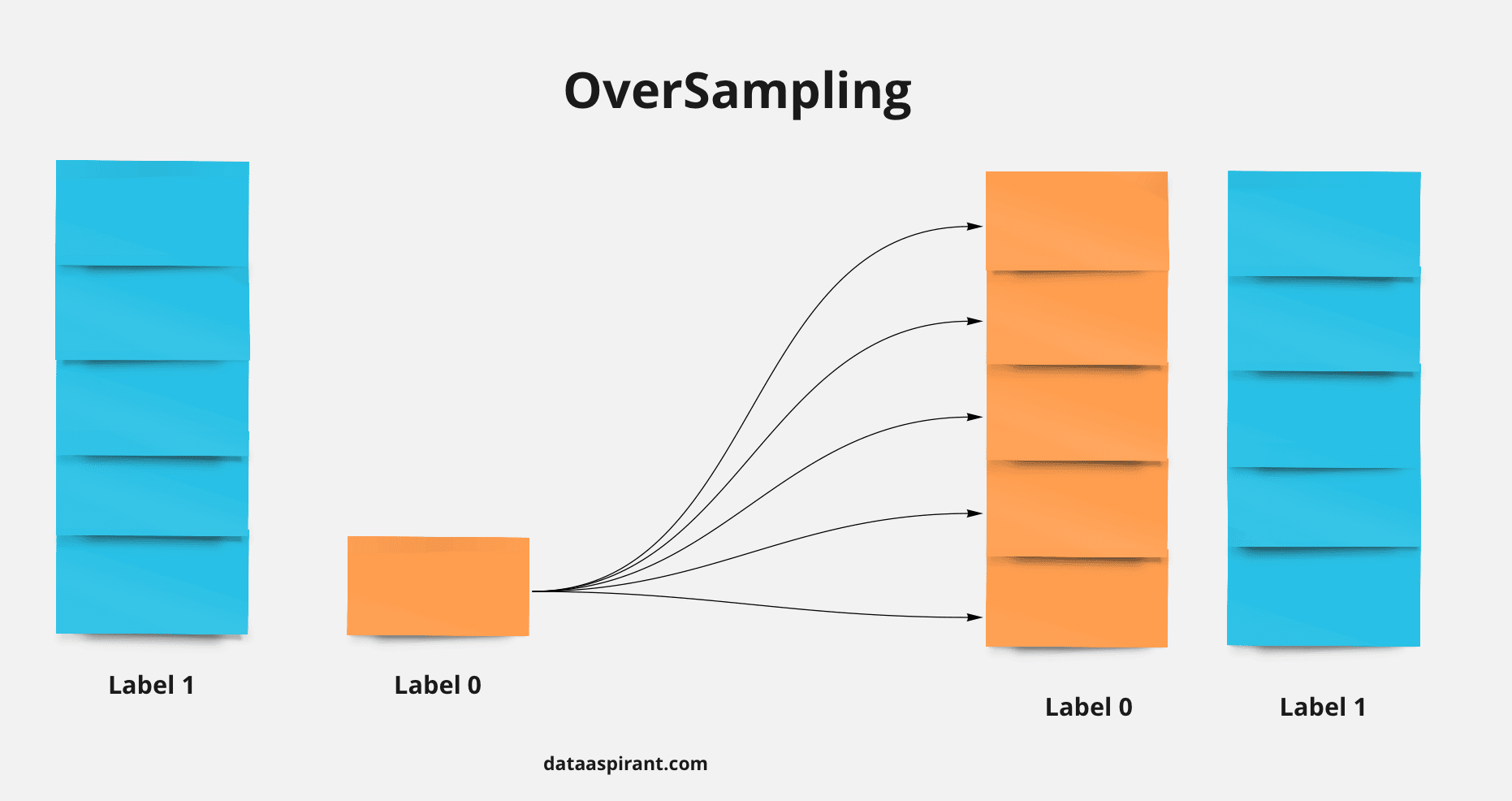
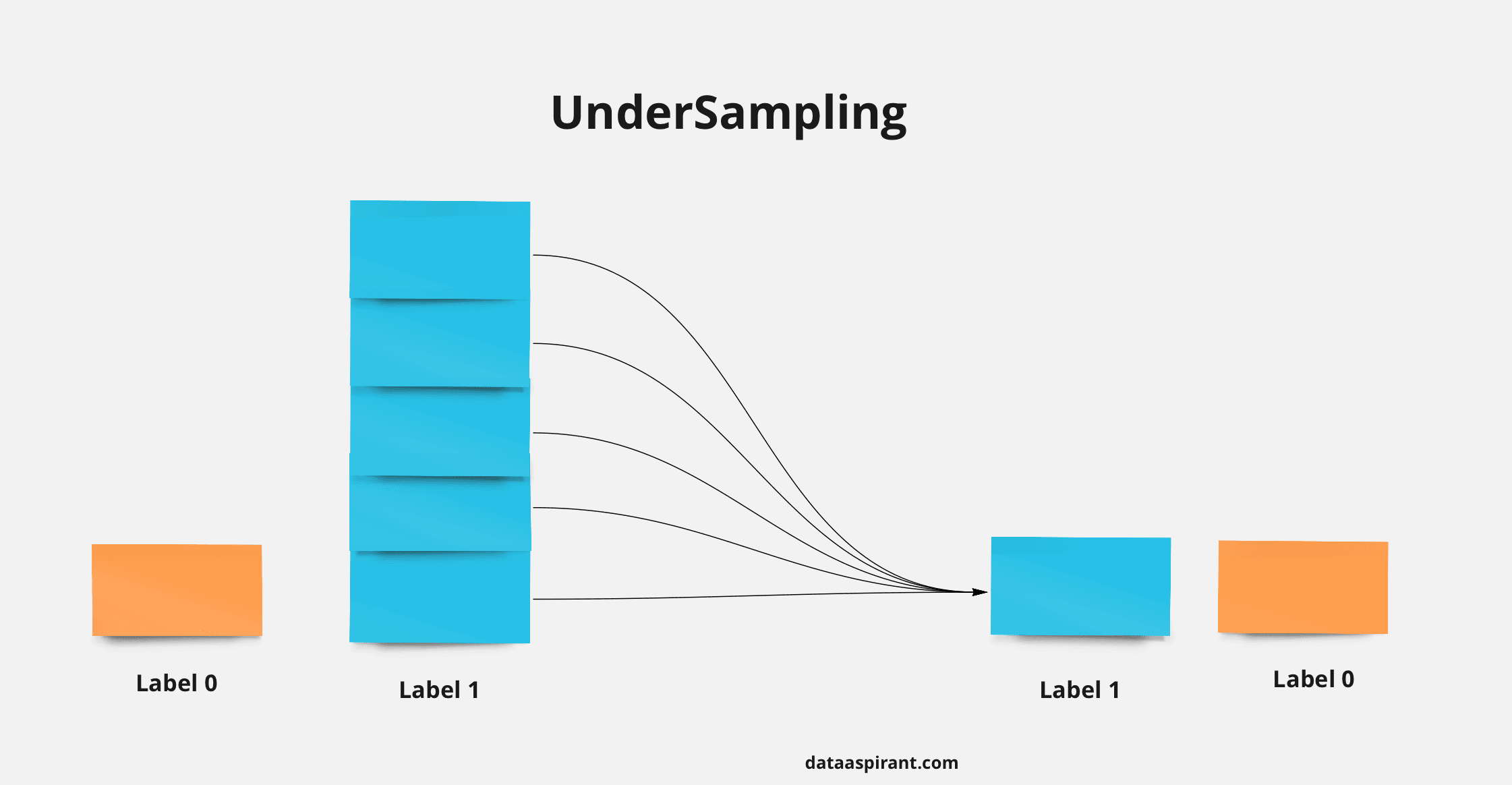
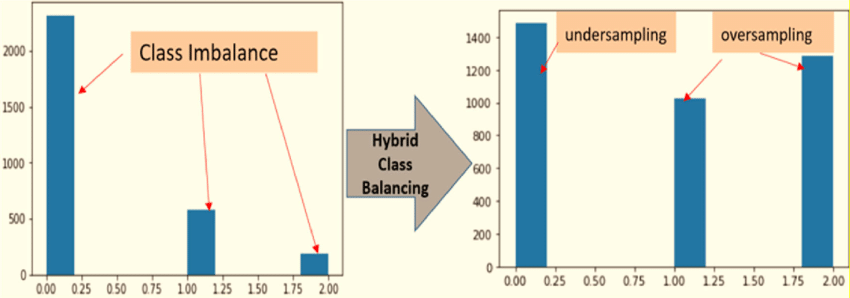
[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F5.png)

* Detección y eliminación de duplicados: Los datos duplicados pueden ser causados por errores de entrada de datos o por la duplicación de registros. La detección y eliminación de duplicados se realiza para asegurar que los datos sean precisos y no estén sesgados.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F6.png)

**Otro problema común: El desequilibrio de clases.**

El desequilibrio de clases es un problema común en machine learning, donde una clase minoritaria tiene muy pocos ejemplos en comparación con una clase mayoritaria. Si no se aborda este problema adecuadamente, los modelos pueden tener dificultades para aprender patrones en la clase minoritaria y pueden estar sesgados hacia la clase mayoritaria. A continuación, se describen algunas formas comunes de abordar el problema de desequilibrio de clases en machine learning:

* Oversampling: El oversampling implica aumentar el número de ejemplos de la clase minoritaria mediante la generación de nuevos ejemplos sintéticos. Las técnicas comunes de oversampling incluyen SMOTE (Synthetic Minority Over-sampling Technique) y ADASYN (Adaptive Synthetic Sampling). [](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F7.png)
* Undersampling: El undersampling implica reducir el número de ejemplos de la clase mayoritaria. Las técnicas comunes de undersampling incluyen la eliminación aleatoria de ejemplos de la clase mayoritaria y la selección de un subconjunto de ejemplos de la clase mayoritaria basado en algún criterio específico. [](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F8.png)
* Métodos híbridos: Los métodos híbridos combinan técnicas de oversampling y undersampling para lograr un equilibrio en el conjunto de datos. Un ejemplo común de un método híbrido es la combinación de oversampling de la clase minoritaria y undersampling de la clase mayoritaria. [](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F9.png)
* Cambio de umbral: El cambio de umbral implica ajustar el umbral de decisión de un clasificador para que favorezca la clasificación de la clase minoritaria. Por ejemplo, en lugar de clasificar una instancia como positiva solo si la probabilidad es mayor que 0,5, se puede cambiar el umbral para clasificar una instancia como positiva si la probabilidad es mayor que 0,3.
* Cost-sensitive learning: El cost-sensitive learning implica asignar un costo diferente a los errores de clasificación para cada clase. Por ejemplo, se puede asignar un mayor costo a los errores de clasificación de la clase minoritaria para que el modelo se enfoque en la clasificación correcta de la clase minoritaria.

**Ejemplo en código**

**Nota**

El código de este ejemplo lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F4%20preprocesamiento%20y%20limpieza%2Fmain.py)

Veamos de nuevo nuestro dataset csv de imágenes de lenguaje de señas: [sign\_mnist\_train\_clean.csv](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2Fdatasets%2Fsign_mnist_train%2Fsign_mnist_train_clean.csv)

Vamos a hacer un análisis de datos y limpieza del mismo:

**1: Importando bibliotecas**

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

sns.set()

**2: Leyendo nuestro dataset**

train = pd.read\_csv("../datasets/sign\_mnist\_train/sign\_mnist\_train\_clean.csv")

print(train)

Respuesta esperada:

label pixel1 pixel2 pixel3 ... pixel781 pixel782 pixel783 pixel784

0 3 107 118 127 ... 206 204 203 202

1 6 155 157 156 ... 175 103 135 149

2 2 187 188 188 ... 198 195 194 195

3 2 211 211 212 ... 225 222 229 163

4 13 164 167 170 ... 157 163 164 179

... ... ... ... ... ... ... ... ... ...

27450 13 189 189 190 ... 234 200 222 225

27451 23 151 154 157 ... 195 195 195 194

27452 18 174 174 174 ... 203 202 200 200

27453 17 177 181 184 ... 47 64 87 93

27454 23 179 180 180 ... 197 205 209 215

[27455 rows x 785 columns]

¿Cuántas clases tiene nuestro dataset?

y = train[["label"]]

n\_clases = sorted(y["label"].unique())

print(len(n\_clases), n\_clases)

Respuesta esperada:

24 [0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]

¿Cuál es la distribución de datos de nuestro dataset por clase?

fig, ax = plt.subplots(figsize=(10, 10))

sns.countplot(y, x="label")

ax.bar\_label(ax.containers[0], rotation=45, label\_type="edge", fmt=lambda x: '{:.1f}%'.format(x/len(train) \* 100))

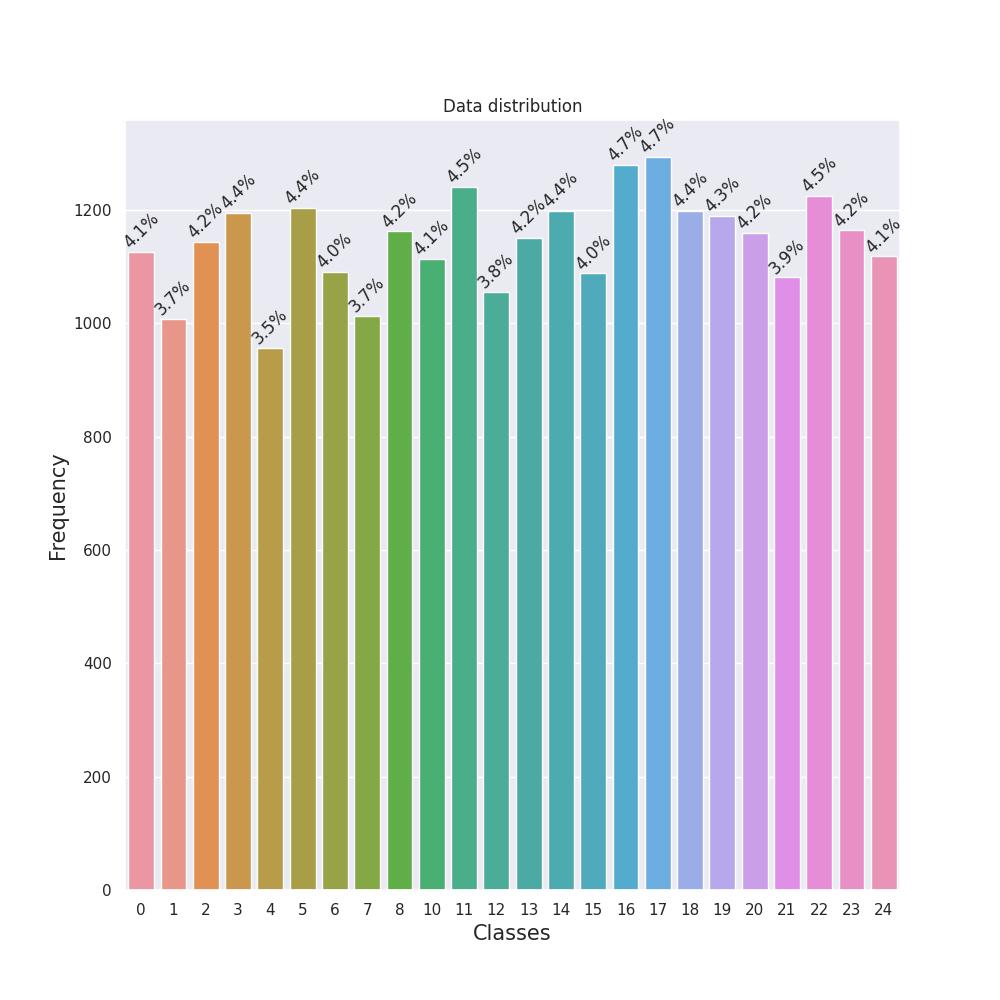
plt.xlabel("Classes", size=15)

plt.ylabel("Frequency", size=15)

plt.title("Data distribution")

plt.savefig("freq.png")

Respuesta esperada:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F4%20preprocesamiento%20y%20limpieza%2Ffreq.png)

En general podemos observar que nuestro dataset NO cuenta con un desequilibrio de claes notorio.

¿Qué tipo de dato tienen nuestras features?

print(X.dtypes)

Respuesta esperada:

pixel1 object

pixel2 object

pixel3 object

pixel4 object

pixel5 object

...

pixel780 object

pixel781 object

pixel782 object

pixel783 object

pixel784 object

Length: 784, dtype: object

Esto indica la presencia de datos de texto en nuestras columnas, lo cuál NO es bueno porque se supone que solamente estamos guardando variables numéricas.

¿Tenemos datos faltantes?

print(X.isnull().values.any())

Respuesta esperada

False

No, no tenemos missing values.

¿Existen datos duplicados?

print(X[X.duplicated()])

Respuesta esperada:

pixel1 pixel2 pixel3 pixel4 ... pixel781 pixel782 pixel783 pixel784

317 0 0 0 0 ... 0 0 0 0

487 0 0 0 0 ... 0 0 0 0

595 0 0 0 0 ... 0 0 0 0

689 0 0 0 0 ... 0 0 0 0

802 fwefew fwefew fwefew fwefew ... fwefew fwefew fwefew fwefew

861 fwefew fwefew fwefew fwefew ... fwefew fwefew fwefew fwefew

Sí, sí tenemos.

**3: Limpiando el dataset**

X = X.drop([595, 689, 727, 802, 861], axis=0)

X = X.astype(str).astype(int)

X /= 255

print(X.head())

print(X.dtypes)

Respuesta esperada:

pixel1 pixel2 pixel3 ... pixel782 pixel783 pixel784

0 0.419608 0.462745 0.498039 ... 0.800000 0.796078 0.792157

1 0.607843 0.615686 0.611765 ... 0.403922 0.529412 0.584314

2 0.733333 0.737255 0.737255 ... 0.764706 0.760784 0.764706

3 0.827451 0.827451 0.831373 ... 0.870588 0.898039 0.639216

4 0.643137 0.654902 0.666667 ... 0.639216 0.643137 0.701961

[5 rows x 784 columns]

pixel1 float64

pixel2 float64

pixel3 float64

pixel4 float64

pixel5 float64

...

pixel780 float64

pixel781 float64

pixel782 float64

pixel783 float64

pixel784 float64

Length: 784, dtype: object

Process finished with exit code 0

Perfecto, nuestros datos ya están normalizados y ahora todos son datos numéricos.

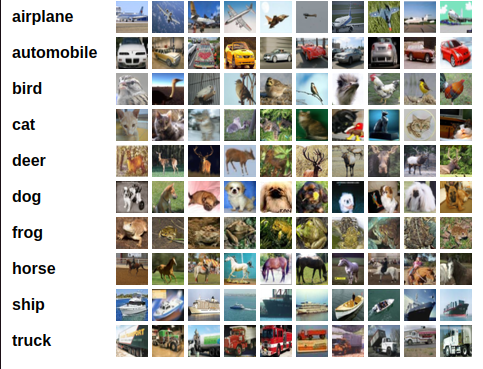
**2.5 Keras datasets**

A lo largo de los cursos anteriores, hemos utilizado varios Datasets de Keras, entre los que pudimos utilizar se encuentra:

* [MNIST digits classification dataset](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/1%20Curso%20de%20fundamentos%20de%20redes%20neuronales#14-tu-primer-red-neuronal-con-keras)
* [CIFAR10 small image classsification dataset](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/2%20Curso%20de%20Redes%20Neuronales%20Convolucionales#5-resolviendo-un-problema-de-clasificaci%C3%B3n)
* [IMDB movie review sentiment classification dataset](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/1%20Curso%20de%20fundamentos%20de%20redes%20neuronales#32-resolviendo-un-problema-de-clasificaci%C3%B3n-binaria)
* [Fashion MNIST dataset, an alternative to MNIST](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/2%20Curso%20de%20Redes%20Neuronales%20Convolucionales#2-mi-primera-red-neuronal-convolucional)
* [Boston Housing price regression dataset](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/1%20Curso%20de%20fundamentos%20de%20redes%20neuronales#39-resolviendo-un-problema-de-regresi%C3%B3n)

Para esta clase, veremos el último dataset disponible en Keras y repasaremos rápidamente como cargarlos y utilizarlos.

El dataset de repaso que veremos será: [CIFAR100 small images classification dataset](https://keras.io/api/datasets/cifar100/) una curiosidad de este dataset es que cuenta con 2 clasificaciones fine y coarse indicando si quieres clasificar por super clase o por clase. A continuación las siguientes clases disponibles:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F10.png)

| **Superclass** | **Classes** |
| --- | --- |
| aquatic mammals | beaver, dolphin, otter, seal, whale |
| fish | aquarium fish, flatfish, ray, shark, trout |
| flowers | orchids, poppies, roses, sunflowers, tulips |
| food containers | bottles, bowls, cans, cups, plates |
| fruit and vegetables | apples, mushrooms, oranges, pears, sweet peppers |
| household electrical devices | clock, computer keyboard, lamp, telephone, television |
| household furniture | bed, chair, couch, table, wardrobe |
| insects | bee, beetle, butterfly, caterpillar, cockroach |
| large carnivores | bear, leopard, lion, tiger, wolf |
| large man-made outdoor things | bridge, castle, house, road, skyscraper |
| large natural outdoor scenes | cloud, forest, mountain, plain, sea |
| large omnivores and herbivores | camel, cattle, chimpanzee, elephant, kangaroo |
| medium-sized mammals | fox, porcupine, possum, raccoon, skunk |
| non-insect invertebrates | crab, lobster, snail, spider, worm |
| people | baby, boy, girl, man, woman |
| reptiles | crocodile, dinosaur, lizard, snake, turtle |
| small mammals | hamster, mouse, rabbit, shrew, squirrel |
| trees | maple, oak, palm, pine, willow |
| vehicles 1 | bicycle, bus, motorcycle, pickup truck, train |
| vehicles 2 | lawn-mower, rocket, streetcar, tank, tractor |

**Ejemplo en código:**

**Nota:**

El código de esta sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F5%20Keras%20datasets%2Fmain.py)

**1: Importando bibliotecas**

from keras.datasets import cifar100

import matplotlib.pyplot as plt

import json

**2: Descargando el dataset**

(x\_train, y\_train), (x\_test, y\_test) = cifar100.load\_data(label\_mode="fine")

print(x\_train.shape)

print(y\_train.shape)

Respuesta esperada:

(50000, 32, 32, 3)

(50000, 1)

**3: Obteniendo el nombre de las clases**

Primero debemos descargar el nombre de las clases

mkdir labels

cd labels

wget --no-check-certificate https://storage.googleapis.com/platzi-tf2/cifar100\_labels.json \

-O cifar100\_labels.json

cd ..

Cargamos los nombres:

with open("labels/cifar100\_labels.json", "r") as fine\_labels:

cifar100\_labels = json.load(fine\_labels)

**4: Inspeccionando el dataset**

num\_image = 40

y = y\_train[num\_image][0]

x = x\_train[num\_image]

y\_name = cifar100\_labels[y]

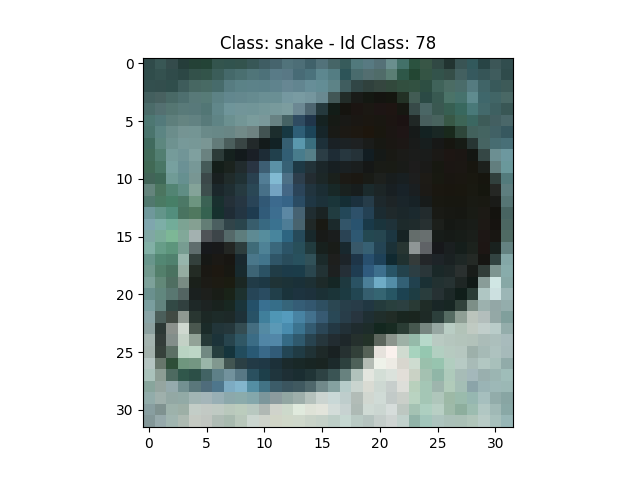
plt.imshow(x)

title = f"Class: {y\_name} - Id Class: {y}"

plt.title(title)

plt.savefig("fig.png")

Respuesta esperada:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F5%20Keras%20datasets%2Ffig.png)

**2.6 Datasets generators**

Este tema será un repaso del tema visto en: [3: Creación de Image Data Generators](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/2%20Curso%20de%20Redes%20Neuronales%20Convolucionales#71-clasificando-entre-perros-y-gatos)

Recordemos que el dataset de Lenguaje de señas MNIST ya lo habíamos descargado [Consiguiendo datos del proyecto](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#consiguiendo-datos-del-proyecto-de-lenguaje-de-se%C3%B1as)

Hay un total de 27,455 imágenes en escala de grises de tamaño 28 \* 28 píxeles cuyo valor oscila entre 0-255. Cada caso representa una etiqueta (0-25) como un mapa uno a uno para cada letra alfabética A-Z (y ningún caso para 9 = J o 25 = Z debido a movimientos gestuales).

Los datos se almacenan de forma ordenada y son compatibles para su uso con generadores de flujo de datos en la API de TensorFlow. Cada carpeta recibe un nombre de acuerdo con la clase de imágenes almacenadas en su interior, lo que facilita su carga y visualización.

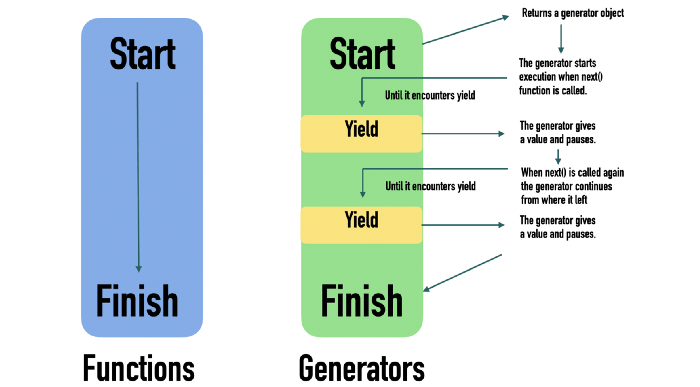
Las imágenes se almacenan en formato de archivo 'JPEG'.

Antes de continuar con el código de ejemplo, es necesario definir qué es un generador en python:

**Generadores**

En Python, un generador es una función que produce una secuencia de valores, pero en lugar de crear y retornar una lista completa de valores de una sola vez, produce cada valor uno por uno, en respuesta a las solicitudes del código que lo llama.

Los generadores son muy útiles cuando se trabaja con grandes conjuntos de datos o cuando se desea generar valores de manera eficiente en función de alguna lógica o algoritmo. En lugar de calcular todos los valores de la secuencia al mismo tiempo y almacenarlos en memoria, un generador calcula cada valor a medida que se solicita, lo que puede ahorrar tiempo y recursos.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F11.png)

Para crear un generador en Python, se utiliza la sentencia "yield" en lugar de "return". Cuando el generador se llama, devuelve un objeto de generador, que se puede iterar para obtener cada valor de la secuencia. Cada vez que se solicita el siguiente valor, la función continúa desde donde se detuvo en la última llamada, en lugar de comenzar desde el principio.

Por ejemplo, el siguiente código define una función generadora que produce los números impares menores que un número dado:

def impares(n):

i = 1

while i < n:

yield i

i += 2

Para usar esta función generadora y obtener los números impares menores que 10, se puede hacer lo siguiente:

for num in impares(10):

print(num)

**Ejemplo en código**

**Nota:**

El código de esta sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F6%20Dataset%20generators%2Fmain.py)

**1: Importando bibliotecas**

from keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt

import string

**2: Creando nuestros ImageDataGenerators**

Para este punto de la clase NO hemos hablado de Data Augmentation cosa que si has tomado este repositorio completo entonces ya debes conocer y entender. Ahora lo único que vamos a hacer es utilizar el ImageDataGenerator para normalizar las imágenes y esta clase nos permita generar un objeto con el método frow\_from\_directory

Nota: Algo interesante es que del test\_dataget también vamos a construir el validation test, por haber definido el parámetro validation\_split

train\_dir = "../../data/Train"

test\_dir = "../../data/Test"

train\_datagen = ImageDataGenerator(rescale=1 / 255)

test\_datagen = ImageDataGenerator(rescale=1 / 255, validation\_split=0.2)

**3: Creamos nuestros generadores de imágenes, para los conjuntos de train, validation y test**

train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=(28, 28),

batch\_size=128,

class\_mode="categorical",

color\_mode="grayscale",

subset="training"

)

validation\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(28, 28),

batch\_size=128,

class\_mode="categorical",

color\_mode="grayscale",

subset="validation"

)

test\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(28, 28),

batch\_size=128,

class\_mode="categorical",

color\_mode="grayscale"

)

Respuesta esperada:

Found 27455 images belonging to 24 classes.

Found 1425 images belonging to 24 classes.

Found 7172 images belonging to 24 classes.

**4: Generando el nombre de las clases**

Sabemos que las clases están en label\_encoding entonces hace falta definir el nombre de las labels de la siguiente manera

classes = [char for char in string.ascii\_uppercase if char not in ("J", "Z")]

print("Classes:", classes)

Respuesta esperada:

Classes: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y']

**4: Muestra de entrenamiento**

Definimos una función auxiliar para mostrar imágenes:

def plot\_images(images\_arr, title):

fig, axes = plt.subplots(1, 5, figsize=(10, 10))

axes = axes.flatten()

for img, ax in zip(images\_arr, axes):

ax.imshow(img[:, :, 0], cmap="gray")

ax.axis("off")

plt.tight\_layout()

plt.savefig(f"{title}.png")

Generamos una muestra de 5 imágenes de nuestro train\_generator

sample\_training\_images, \_ = next(train\_generator)

plot\_images(sample\_training\_images[:5], "Train Example")

Respuesta esperada:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F6%20Dataset%20generators%2FTrain%20Example.png)

**2.7 Aprende a buscar bases de datos para deep learning**

Deep Learning NO es solamente código. Como ya hemos mencionado anteriormente, es sumamente importante contar con una base de datos de calidad para asegurar que un proyecto tenga éxito. Sin embargo, no existen el 100% de bases de datos para el 100% de nuestros problemas. Si estamos lidiando con un problema muy específico o de un nicho muy concreto como puede ser clasificar un tipo de planta que solamente crece en nuestro jardin es 100% probable que nadie haya compartido un dataset público de los tipos de planta que crecen en nuestra casa.

A pesar de ello, existen diferentes plataformas donde podemos encontrar una gran variedad de datasets que podemos usar para nuestros proyectos de machine learning y deep learning. Entre algunas páginas donde puedes encontrar bases de datos podemos nombrar:

* Páginas de datos públicos de tu pais. En México es: [datos.gob.mx](https://datos.gob.mx/)
* Kaggle
* Dataset Search
* Data.world (tiene costo)
* GitHub [Awesome public datasets](https://github.com/awesomedata/awesome-public-datasets)

**2.8 Cómo distribuir los datos**

Antes de continuar, te recomiendo darte una vuelta por mi repositorio de [Machine Learning con Python y Scikit-learn](https://github.com/ichcanziho/cursos_platzi/tree/master/machine_learning_scikit_learn)

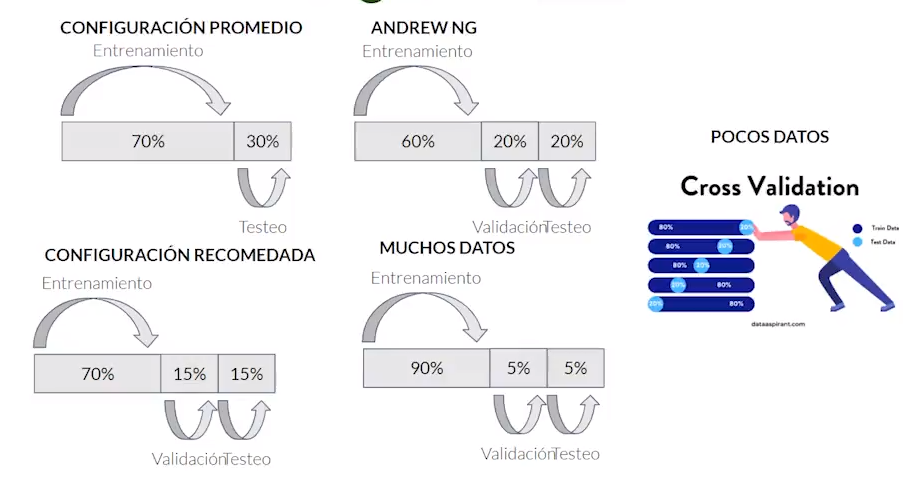
Especialmente al siguiente tema: [Optimización Paramétrica](https://github.com/ichcanziho/cursos_platzi/tree/master/machine_learning_scikit_learn#7-optimizaci%C3%B3n-param%C3%A9trica)

Sin embargo, a continuación te brindo un poco de información por si no tienes muy claro como dividir nuestros datasets.

La distribución de las particiones de los datasets en deep learning puede variar según el enfoque específico utilizado, pero aquí te presento algunas formas comunes en que se dividen los datos:

* Validación cruzada: este método implica dividir el conjunto de datos en k pliegues, y realizar k experimentos diferentes en los que se utiliza un pliegue diferente como conjunto de validación cada vez. Este enfoque puede ayudar a obtener una estimación más precisa del rendimiento del modelo.
* División de entrenamiento y validación: en este enfoque, el conjunto de datos se divide en dos conjuntos, uno para el entrenamiento y otro para la validación. El conjunto de entrenamiento se utiliza para ajustar los parámetros del modelo, mientras que el conjunto de validación se utiliza para evaluar el rendimiento del modelo y ajustar sus hiperparámetros.
* División de entrenamiento, validación y prueba: este enfoque es similar al anterior, pero se divide el conjunto de datos en tres conjuntos: entrenamiento, validación y prueba. El conjunto de prueba se utiliza para evaluar el rendimiento final del modelo después de haber ajustado los parámetros y los hiperparámetros utilizando el conjunto de entrenamiento y el conjunto de validación.
* Bootstrap: este método implica muestrear el conjunto de datos con reemplazo varias veces para generar múltiples conjuntos de entrenamiento y validación. Esto puede ayudar a reducir la varianza del modelo y mejorar su capacidad para generalizar a nuevos datos.
* División basada en el tiempo: en algunos casos, puede ser útil dividir el conjunto de datos en función de la fecha o el tiempo. Por ejemplo, se podría utilizar el conjunto de datos más antiguo para el entrenamiento, el conjunto de datos más reciente para la validación y un conjunto de datos futuro para la prueba.

Nota: Entre más pequeña es la base de datos más se recomienda usar validación cruzada, sin embargo, si la base de datos es muy larga esto puede ser contraproducente y se recomienda mejor utilizar el enfoque: train, validation, test con una distribución 90, 5, 5.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F2%2F12.png)

**Errores comunes en datasets**

* Agregar datos de entrenamiento a testeo
* Bases de datos no balanceadas en clases
* Muy pocos datos

**2.9 Crear la red neuronal, definir capas, compilar, entrenar, evaluar y predicciones**

**Nota:**

El código completo lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F7%20Crear%20la%20red%2Fmain.py)

Este será nuestra primera aproximación a resolver el problema de clasificación y de lenguaje de señas. Pero antes de continuar vamos a utilizar un par de funciones que ya hemos manejado en cursos anteriores.

Primero: utilizaremos nuestra muy querida función plot\_results que hemos usado en los cursos anteriores de deep learning

def plot\_results(history\_, metric, fname):

history\_dict = history\_.history

loss\_values = history\_dict['loss']

val\_loss\_values = history\_dict['val\_loss']

metric\_values = history\_dict[metric]

val\_metric\_values = history\_dict[f"val\_{metric}"]

epoch = range(1, len(loss\_values) + 1)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 5))

fig.suptitle("Neural Network's Result")

ax1.set\_title("Loss function over epoch")

ax2.set\_title(f"{metric} over epoch")

ax1.set(ylabel="loss", xlabel="epochs")

ax2.set(ylabel=metric, xlabel="epochs")

ax1.plot(epoch, loss\_values, 'go-', label='training')

ax1.plot(epoch, val\_loss\_values, 'ro-', label='validation')

ax2.plot(epoch, metric\_values, 'go-', label='training')

ax2.plot(epoch, val\_metric\_values, 'ro-', label='validation')

ax1.legend()

ax2.legend()

plt.savefig(f"{fname}")

plt.close()

Para efectos de continuidad, vamos a definir una función get\_data() con todo lo aprendido en la clase [Dataset generators](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#26-datasets-generators)

def get\_data():

train\_dir = "../../data/Train"

test\_dir = "../../data/Test"

\_bs = 128

train\_datagen = ImageDataGenerator(rescale=1 / 255)

test\_datagen = ImageDataGenerator(rescale=1 / 255, validation\_split=0.2)

\_train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=(28, 28),

batch\_size=\_bs,

class\_mode="categorical",

color\_mode="grayscale",

subset="training"

)

\_validation\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(28, 28),

batch\_size=\_bs,

class\_mode="categorical",

color\_mode="grayscale",

subset="validation"

)

\_test\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(28, 28),

batch\_size=\_bs,

class\_mode="categorical",

color\_mode="grayscale"

)

\_classes = [char for char in string.ascii\_uppercase if char not in ("J", "Z")]

return \_classes, \_bs, \_train\_generator, \_validation\_generator, \_test\_generator

Ahora continuemos con la clase actual.

Los pasos a seguir para esta clase serán los siguientes:

1. Cargar las particiones de datos
2. Crear la arquitectura base del modelo
3. Compilar el modelo
4. Entrenar el modelo
5. Mostrar resultados

**1: Cargar las particiones de datos**

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data()

Respuesta esperada:

Found 27455 images belonging to 24 classes.

Found 1425 images belonging to 24 classes.

Found 7172 images belonging to 24 classes.

**2: Crear la arquitectura base del modelo**

Creamos una función auxiliar:

def base\_architecture(input\_shape, n\_clases):

model = Sequential()

model.add(Flatten(input\_shape=input\_shape))

model.add(Dense(256, activation="relu"))

model.add(Dense(128, activation="relu"))

model.add(Dense(n\_clases, activation="softmax"))

print(model.summary())

return model

Llamamos a nuestra función auxiliar en nuestro ciclo principal.

base\_model = base\_architecture(input\_shape=(28, 28, 1), n\_clases=len(classes))

Respuesta esperada:

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

flatten (Flatten) (None, 784) 0

dense (Dense) (None, 256) 200960

dense\_1 (Dense) (None, 128) 32896

dense\_2 (Dense) (None, 24) 3096

=================================================================

Total params: 236,952

Trainable params: 236,952

Non-trainable params: 0

**3: Compilamos el modelo**

base\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

Recordemos que como es un problema de clasificación multiple nuestra perdida debe ser: categorical\_crossentropy y la última capa de clasificación del modelo tendrá 24 neuronas una por cada clase disponible y a su vez esto nos hace necesitar usar softmax como función de activación.

**4: Entrenamos el modelo**

history = base\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128)

Respuesta esperada:

Epoch 1/20

215/215 [==============================] - 3s 10ms/step - loss: 2.1329 - accuracy: 0.3710 - val\_loss: 1.6110 - val\_accuracy: 0.4821

Epoch 2/20

215/215 [==============================] - 2s 10ms/step - loss: 1.2105 - accuracy: 0.6255 - val\_loss: 1.3437 - val\_accuracy: 0.5923

Epoch 3/20

215/215 [==============================] - 2s 10ms/step - loss: 0.8712 - accuracy: 0.7350 - val\_loss: 1.1443 - val\_accuracy: 0.6421

Epoch 4/20

...

Epoch 18/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0271 - accuracy: 0.9955 - val\_loss: 1.2362 - val\_accuracy: 0.7474

Epoch 19/20

215/215 [==============================] - 2s 10ms/step - loss: 0.1214 - accuracy: 0.9633 - val\_loss: 1.3080 - val\_accuracy: 0.7011

Epoch 20/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0210 - accuracy: 0.9970 - val\_loss: 1.2360 - val\_accuracy: 0.7607

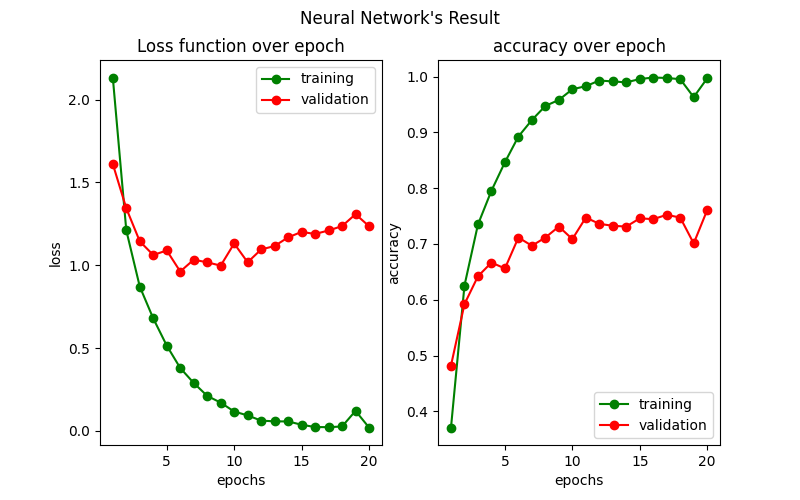
**5: Análisis de resultados**

plot\_results(history, "accuracy", "base\_results.png")

results = base\_model.evaluate(test\_generator)

Respuesta esperada:

57/57 [==============================] - 1s 9ms/step - loss: 1.1455 - accuracy: 0.7655

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/2%20Manejo%20y%20preprocesamiento%20de%20datos%2F7%20Crear%20la%20red%2Fbase_results.png)

Para resumir, la estructura en código de este ejercicio fue:

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data()

base\_model = base\_architecture(input\_shape=(28, 28, 1), n\_clases=len(classes))

base\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

history = base\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128)

plot\_results(history, "accuracy", "base\_results.png")

results = base\_model.evaluate(test\_generator)

Para este momento hemos conseguido un accuracy del 76.5% del modelo, sin embargo, nuestras gráficas de perdida y accuracy muestran como el desempeño en el conjunto de validación es bastante inferior al obtenido en entrenamiento, esto refleja un claro caso de overfitting. Nuestro modelo debe ser optimizado para mejorar los resultados de validación y con ello mejorar los resultados de testing. En próximas clases iremos mejorando estos resultados. Las propuestas de mejora son:

1. Usar regularizadores
2. Cambiar arquitectura de la red por una CNN
3. Usar DataAugmentation
4. Usar transfer learning

**3 Optimización de precisión de modelos**

A lo largo de esta sección estaremos hablando de los siguientes temas:

1. Underfitting y Overfitting
2. Recomendaciones para ajustar mi modelo
3. Métricas para medir eficacia: Callbacks
4. Monitoreo del entrenamiento de modelos con early stopping
5. Autotunning con Keras

**3.1 Métodos de regularización: overfitting y underfitting**

Antes de leer el resumen de esta sección te recomiendo visitar el siguiente enlace a [Regularizadores en deep learning](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/1%20Curso%20de%20fundamentos%20de%20redes%20neuronales#34-regularizaci%C3%B3n---dropout) Del repositorio: [Fundamentos de deep learning](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/1%20Curso%20de%20fundamentos%20de%20redes%20neuronales#curso-de-fundamentos-de-redes-neuronales-con-python-y-keras)

A continuación un pequeño resumen para refrescar nuestros conocimientos de Regularizadores:

Los métodos de regularización son técnicas utilizadas en deep learning para prevenir el sobreajuste (overfitting) del modelo, es decir, cuando el modelo se ajusta demasiado bien a los datos de entrenamiento, lo que puede afectar su capacidad de generalización a nuevos datos.

Aquí hay algunos métodos de regularización comunes en deep learning:

* Regularización L1 y L2: La regularización L1 y L2 añaden un término de penalización a la función de pérdida del modelo. L1 penaliza la suma de los valores absolutos de los pesos, mientras que L2 penaliza la suma de los cuadrados de los pesos. Estos métodos reducen la magnitud de los pesos, lo que puede reducir la complejidad del modelo y prevenir el sobreajuste.

Nota: L1 es utilizado para datos de entrada irrelevantes, L2 es para datos de entrada correlacionados entre ellos y existe ElasticNet que es L1 + L2 que se usa cuando hay un gran número de atributos

* Dropout: El dropout es una técnica que consiste en apagar de manera aleatoria algunas neuronas durante el entrenamiento. Esto hace que el modelo sea menos dependiente de un subconjunto particular de neuronas, lo que puede reducir el sobreajuste.
* Data Augmentation: La data augmentation es una técnica que implica aumentar artificialmente la cantidad de datos de entrenamiento mediante la aplicación de transformaciones como rotaciones, zooms, desplazamientos, etc. Esto puede ayudar a prevenir el sobreajuste al proporcionar al modelo más variabilidad en los datos de entrenamiento.
* Early stopping: El early stopping es una técnica que se utiliza para evitar el sobreajuste al detener el entrenamiento antes de que el modelo alcance su convergencia. El modelo se detiene cuando la función de pérdida en el conjunto de validación comienza a aumentar, lo que indica que el modelo está comenzando a sobreajustar.
* Batch Normalization: La normalización de batch es una técnica que normaliza las activaciones de cada capa del modelo. Esto ayuda a reducir la covariación de las activaciones y permite que el modelo se ajuste más fácilmente a los datos de entrenamiento.

**Ejemplo en código:**

**Nota:**

El código de esta sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F1%20M%C3%A9todos%20de%20regularizaci%C3%B3n%2Fmain.py)

La estructura de código de esta clase se mantiene igual a la de la clase anterior, el único cambio relevante es el cambio de la arquitectura base del modelo por una regularizada:

def base\_architecture\_w\_regularizes(input\_shape, n\_clases):

model = Sequential()

model.add(Flatten(input\_shape=input\_shape))

model.add(Dense(256, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(128, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(n\_clases, activation="softmax"))

print(model.summary())

return model

Respuesta esperada:

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

flatten (Flatten) (None, 784) 0

dense (Dense) (None, 256) 200960

dropout (Dropout) (None, 256) 0

dense\_1 (Dense) (None, 128) 32896

dropout\_1 (Dropout) (None, 128) 0

dense\_2 (Dense) (None, 24) 3096

=================================================================

Total params: 236,952

Trainable params: 236,952

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ejecutamos el código principal:

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data()

regularized\_model = base\_architecture\_w\_regularizes(input\_shape=(28, 28, 1), n\_clases=len(classes))

regularized\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

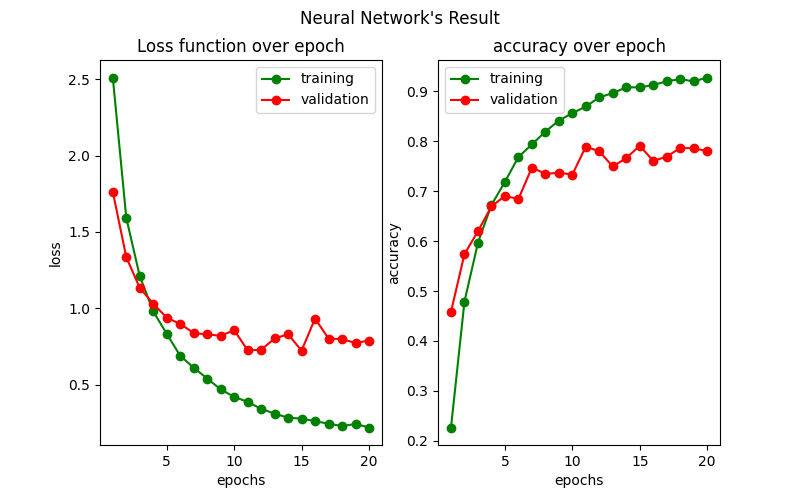
history = regularized\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128)

plot\_results(history, "accuracy", "regularized\_results.png")

results = regularized\_model.evaluate(test\_generator)

Respuesta esperada:

57/57 [==============================] - 1s 16ms/step - loss: 0.7687 - accuracy: 0.7835

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F1%20M%C3%A9todos%20de%20regularizaci%C3%B3n%2Fregularized_results.png)

En este caso podemos observar como con unas simples líneas hemos podido reducir el overfitting presente en la versión base de nuestro modelo, sin embargo, aún estamos lejos de que los resultados de validación sean tan exitosos como los resultados de entrenamiento.

**3.2 Recomendaciones prácticas para ajustar un modelo**

Cuando nos enfrentamos a un nuevo problema de Deep Learning es buena idea tener en cuantas los siguientes consejos:

* Preprocesamiento
  + Buscar datos null
  + Archivos corruptos
  + Balancea tu base de datos
  + Aplicar normalización
  + Visualiza la base de datos
  + Entiende los datos
* Recomendaciones de valores
  + Convoluciones (3x3)
  + Pooling (2x2)
  + Flatten (imágenes)
  + Neuronas (64, 128, 256, 512)
  + Learning Rate (0.001) ADAM
  + L1\_L2 (1e-5)
  + Dropout (0.2)
* Recomendaciones para regularizadores
  + L1, L2, L1+l2
  + Agregar más datos
  + Data augmentation
  + Dropout
  + Early stopping
  + Callbacks
* Recomendaciones de funciones de activación
  + MultiClass: Softmax
  + Binary: Sigmoidal
  + Regression: linear function
  + Predicciones mayor que 0: ReLU
* Recomendaciones de configuración de red
  + Aplicar capas convolucionales y poolings
  + Almacena el modelo en cada epoch
* Recomendaciones de la red
  + Más capas no significa necesariamente una mejor red
  + La solución NO siempre es redes neuronales
  + Aliado del aprendizaje por transferencia

En esta clase cambiaremos ligeramente nuestra arquitectura de la clase anterior para utilizar capas Convolucionales y ver la mejora en nuestro accuracy general.

**Ejemplo en código:**

**Nota:**

El código de esta sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F2%20CNN%20base%2Fmain.py)

Para este momento del curso ya deberías estar familiarizado con CNNs, si tienes dudas del siguiente código visita el repositorio anterior. [Curso de Redes Neuronales Convolucionales](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/2%20Curso%20de%20Redes%20Neuronales%20Convolucionales)

def conv\_architecture(input\_shape, n\_clases):

model = Sequential()

model.add(Conv2D(filters=128, kernel\_size=(3, 3), activation="relu", input\_shape=input\_shape))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(128, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(n\_clases, activation="softmax"))

print(model.summary())

return model

Respuesta esperada:

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d (Conv2D) (None, 26, 26, 128) 1280

max\_pooling2d (MaxPooling2D (None, 13, 13, 128) 0

)

flatten (Flatten) (None, 21632) 0

dense (Dense) (None, 256) 5538048

dropout (Dropout) (None, 256) 0

dense\_1 (Dense) (None, 128) 32896

dropout\_1 (Dropout) (None, 128) 0

dense\_2 (Dense) (None, 24) 3096

=================================================================

Total params: 5,575,320

Trainable params: 5,575,320

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ejecutamos el código principal:

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data()

conv\_model = conv\_architecture(input\_shape=(28, 28, 1), n\_clases=len(classes))

conv\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

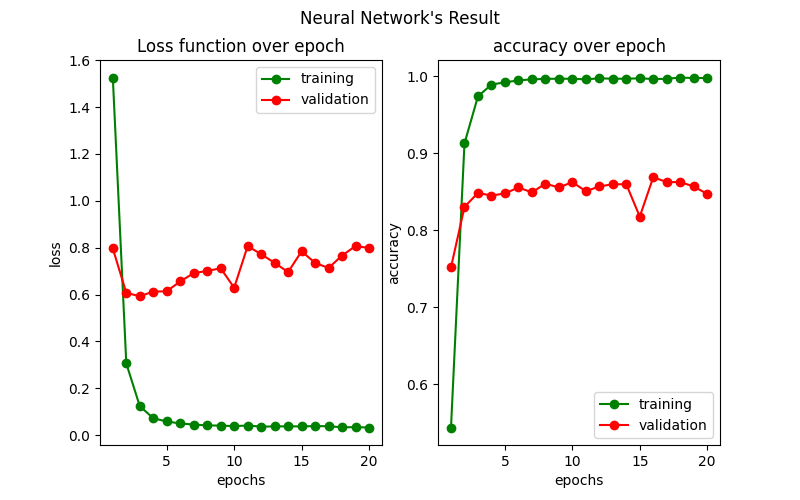
history = conv\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128)

plot\_results(history, "accuracy", "conv\_results.png")

results = conv\_model.evaluate(test\_generator)

Respuesta esperada:

57/57 [==============================] - 1s 11ms/step - loss: 0.7347 - accuracy: 0.8572

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F2%20CNN%20base%2Fconv_results.png)

En general la precisión del modelo ha mejorado notablemente respecto al modelo anterior, sin embargo, de nuevo estamos teniendo un claro ejemplo de overfitting esto es debido a que la complejidad del modelo general ha aumentado bastante. Debemos tomar esto en cuenta para generar una arquitectura más versátil en la parte convolucional.

**3.3 Métricas para medir la eficiencia de un modelo: Callback**

En Keras, un callback es una función que se puede pasar como argumento al método fit() de un modelo y que se ejecuta en ciertos puntos durante el entrenamiento del modelo. Los callbacks se utilizan comúnmente para realizar tareas como la visualización del progreso del entrenamiento, el guardado de modelos intermedios, el ajuste de la tasa de aprendizaje, la detención temprana del entrenamiento, entre otras.

Algunos ejemplos de cómo se pueden usar los callbacks en Keras incluyen:

* [ModelCheckpoint:](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint) Permite guardar el modelo en puntos específicos durante el entrenamiento para evitar la pérdida de datos.
* [EarlyStopping:](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping) Detiene el entrenamiento temprano si la mejora en la función de pérdida en el conjunto de validación disminuye.
* [ReduceLROnPlateau:](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau) Reduce la tasa de aprendizaje del modelo si la mejora en la función de pérdida en el conjunto de validación se detiene o se estanca.
* [TensorBoard:](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/TensorBoard) Permite visualizar el progreso del entrenamiento en una interfaz gráfica de usuario fácil de usar.
* [CustomCallbacks:](https://www.tensorflow.org/guide/keras/custom_callback?hl=es-419) Los usuarios pueden crear sus propios callbacks personalizados para realizar tareas específicas y/o implementar su lógica personalizada.

**Ejemplo en código**

**Nota:**

El código de esta sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F3%20Callback%2Fmain.py)

En este ejemplo vamos a definir un pequeño Custom Callback

Supongamos que queremos detener el proceso de entrenamiento del modelo cuando el mismo haya logrado alcanzar al menos 80% de accuracy en el set de validación.

Lo único que necesitamos hacer es crear una clase que contenga nuestra CustoCallback:

from keras.callbacks import Callback

class TrainingCallbackStop(Callback):

def \_\_init\_\_(self, acc\_stop):

super().\_\_init\_\_()

self.acc\_stop = acc\_stop

def on\_epoch\_end(self, epoch, logs=None):

if logs.get("val\_accuracy") > self.acc\_stop:

print(f"Lo hemos logrado, el modelo ha llegado a un {self.acc\_stop\*100}%")

self.model.stop\_training = True

Existen varios overrides que podemos hacer para crear nuestras Callbacks propias, entre ellas podemos encontrar las siguientes:

from keras.callbacks import Callback

class TrainingCallback(Callback):

def on\_train\_begin(self, logs=None):

print('Starting training....')

def on\_epoch\_begin(self, epoch, logs=None):

print('Starting epoch {}'.format(epoch))

def on\_train\_batch\_begin(self, batch, logs=None):

print('Training: Starting batch {}'.format(batch))

def on\_train\_batch\_end(self, batch, logs=None):

print('Training: Finished batch {}'.format(batch))

def on\_epoch\_end(self, epoch, logs=None):

print('Finished epoch {}'.format(epoch))

def on\_train\_end(self, logs=None):

print('Finished training!')

class TestingCallback(Callback):

def on\_test\_begin(self, logs=None):

print('Starting testing....')

def on\_test\_batch\_begin(self, batch, logs=None):

print('Testing: Starting batch {}'.format(batch))

def on\_test\_batch\_end(self, batch, logs=None):

print('Testing: Finished batch {}'.format(batch))

def on\_test\_end(self, logs=None):

print('Finished testing!')

class PredictionCallback(Callback):

def on\_predict\_begin(self, logs=None):

print('Prediction testing....')

def on\_predict\_batch\_begin(self, batch, logs=None):

print('Prediction: Starting batch {}'.format(batch))

def on\_predict\_batch\_end(self, batch, logs=None):

print('Prediction: Finished batch {}'.format(batch))

def on\_predict\_end(self, logs=None):

print('Finished prediction!')

Las callbacks son asignadas durante el proceso de entrenamiento del modelo:

callback = TrainingCallbackStop(acc\_stop=0.80)

history = conv\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128,

callbacks=[callback])

Respuesta esperada:

Epoch 1/20

215/215 [==============================] - 4s 10ms/step - loss: 1.6634 - accuracy: 0.5015 - val\_loss: 0.7740 - val\_accuracy: 0.7698

Epoch 2/20

215/215 [==============================] - 2s 10ms/step - loss: 0.3744 - accuracy: 0.8893 - val\_loss: 0.7070 - val\_accuracy: 0.7895

Epoch 3/20

209/215 [============================>.] - ETA: 0s - loss: 0.1551 - accuracy: 0.9629

Lo hemos logrado, el modelo ha llegado a un 80.0%

215/215 [==============================] - 2s 10ms/step - loss: 0.1538 - accuracy: 0.9633 - val\_loss: 0.6205 - val\_accuracy: 0.8337

Aquí el proceso se ha detenido a pesar de que le hemos pedido que hiciera 20 epochs esto muestra como hemos hecho funcionar correctamente nuestra propuesta de EarlyStop basada en el val\_accuracy.

**3.4 Monitoreo del entrenamiento en tiempo real: early stopping y patience**

En esta clase veremos un el callback de EarlyStopping del cual ya hemos tenido una clase en [early stopping y checkpoints](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/2%20Curso%20de%20Redes%20Neuronales%20Convolucionales#63-callbacks-early-stopping-y-checkpoints)

Este es solamente un breve repaso:

**Nota:**

El código completo lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F4%20Early%20stopping%2Fmain.py)

Recordemos que:

* monitor: Es la variable a la cuál vamos a estar monitoreando para detener el proceso de aprendizaje
* patience: Es la cantidad de épocas que vamos a esperar a que un resultado mejore (en caso de ACC) o decremente (en caso de LOSS)
* mode: Puede ser: auto, min, max dependiendo de si el monitor es acc o loss, por ejemplo si fuera loss entonces el modo debe ser min.

from keras.callbacks import EarlyStopping

callback = EarlyStopping(monitor="val\_accuracy", patience=3, mode="auto")

history = conv\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128,

callbacks=[callback])

Respuesta esperada:

Epoch 1/20

215/215 [==============================] - 4s 10ms/step - loss: 1.6012 - accuracy: 0.5136 - val\_loss: 0.7442 - val\_accuracy: 0.7516

Epoch 2/20

215/215 [==============================] - 2s 10ms/step - loss: 0.3211 - accuracy: 0.9085 - val\_loss: 0.5699 - val\_accuracy: 0.8470

Epoch 3/20

215/215 [==============================] - 2s 10ms/step - loss: 0.1211 - accuracy: 0.9756 - val\_loss: 0.5369 - val\_accuracy: 0.8568

Epoch 4/20

215/215 [==============================] - 2s 9ms/step - loss: 0.0758 - accuracy: 0.9884 - val\_loss: 0.6879 - val\_accuracy: 0.8386

Epoch 5/20

215/215 [==============================] - 2s 9ms/step - loss: 0.0575 - accuracy: 0.9932 - val\_loss: 0.6424 - val\_accuracy: 0.8484

Epoch 6/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0521 - accuracy: 0.9937 - val\_loss: 0.7010 - val\_accuracy: 0.8344

El mejor histórico lo alcanzo en la época 3 con un val\_accuracy de 0.8568 y como durante 3 épocas NO mejoro este resultado, entonces automáticamente detuvo el proceso de entrenamiento.

**PLUS: guardando el modelo con checkpoints**

Crearemos otro callback para guardar el modelo cada que encuentre una época que optimice el parámetro de accuracy. Lo guardaremos en la carpeta models con el nombre de: best\_model.h5.

from keras.callbacks import EarlyStopping, ModelCheckpoint

callback = EarlyStopping(monitor="val\_accuracy", patience=3, mode="auto")

checkpoint = ModelCheckpoint(filepath="models/best\_model.h5", save\_best\_only=True, save\_weights\_only=False,

mode="auto", verbose=1, monitor="val\_accuracy")

history = conv\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128,

callbacks=[callback, checkpoint])

Respuesta esperada:

Epoch 1/20

213/215 [============================>.] - ETA: 0s - loss: 1.5060 - accuracy: 0.5490

Epoch 1: val\_accuracy improved from -inf to 0.78667, saving model to models/best\_model.h5

215/215 [==============================] - 4s 11ms/step - loss: 1.4962 - accuracy: 0.5520 - val\_loss: 0.6782 - val\_accuracy: 0.7867

Epoch 2/20

211/215 [============================>.] - ETA: 0s - loss: 0.2680 - accuracy: 0.9291

Epoch 2: val\_accuracy improved from 0.78667 to 0.84561, saving model to models/best\_model.h5

215/215 [==============================] - 2s 10ms/step - loss: 0.2658 - accuracy: 0.9299 - val\_loss: 0.5747 - val\_accuracy: 0.8456

Epoch 3/20

211/215 [============================>.] - ETA: 0s - loss: 0.1038 - accuracy: 0.9800

Epoch 3: val\_accuracy improved from 0.84561 to 0.85965, saving model to models/best\_model.h5

215/215 [==============================] - 2s 10ms/step - loss: 0.1034 - accuracy: 0.9801 - val\_loss: 0.5672 - val\_accuracy: 0.8596

Epoch 4/20

211/215 [============================>.] - ETA: 0s - loss: 0.0667 - accuracy: 0.9913

Epoch 4: val\_accuracy did not improve from 0.85965

215/215 [==============================] - 2s 10ms/step - loss: 0.0665 - accuracy: 0.9914 - val\_loss: 0.6343 - val\_accuracy: 0.8491

Epoch 5/20

210/215 [============================>.] - ETA: 0s - loss: 0.0542 - accuracy: 0.9947

Epoch 5: val\_accuracy did not improve from 0.85965

215/215 [==============================] - 2s 10ms/step - loss: 0.0540 - accuracy: 0.9948 - val\_loss: 0.6946 - val\_accuracy: 0.8414

Epoch 6/20

212/215 [============================>.] - ETA: 0s - loss: 0.0473 - accuracy: 0.9957

Epoch 6: val\_accuracy improved from 0.85965 to 0.86456, saving model to models/best\_model.h5

215/215 [==============================] - 2s 10ms/step - loss: 0.0473 - accuracy: 0.9957 - val\_loss: 0.6618 - val\_accuracy: 0.8646

Epoch 7/20

211/215 [============================>.] - ETA: 0s - loss: 0.0440 - accuracy: 0.9963

Epoch 7: val\_accuracy did not improve from 0.86456

215/215 [==============================] - 2s 10ms/step - loss: 0.0440 - accuracy: 0.9962 - val\_loss: 0.6135 - val\_accuracy: 0.8632

Epoch 8/20

215/215 [==============================] - ETA: 0s - loss: 0.0404 - accuracy: 0.9970

Epoch 8: val\_accuracy did not improve from 0.86456

215/215 [==============================] - 2s 10ms/step - loss: 0.0404 - accuracy: 0.9970 - val\_loss: 0.6987 - val\_accuracy: 0.8618

Epoch 9/20

210/215 [============================>.] - ETA: 0s - loss: 0.0391 - accuracy: 0.9969

Epoch 9: val\_accuracy improved from 0.86456 to 0.86526, saving model to models/best\_model.h5

215/215 [==============================] - 2s 10ms/step - loss: 0.0392 - accuracy: 0.9969 - val\_loss: 0.6518 - val\_accuracy: 0.8653

Epoch 10/20

210/215 [============================>.] - ETA: 0s - loss: 0.0375 - accuracy: 0.9975

Epoch 10: val\_accuracy improved from 0.86526 to 0.86947, saving model to models/best\_model.h5

215/215 [==============================] - 2s 10ms/step - loss: 0.0373 - accuracy: 0.9976 - val\_loss: 0.7292 - val\_accuracy: 0.8695

Epoch 11/20

210/215 [============================>.] - ETA: 0s - loss: 0.0371 - accuracy: 0.9974

Epoch 11: val\_accuracy did not improve from 0.86947

215/215 [==============================] - 2s 10ms/step - loss: 0.0370 - accuracy: 0.9974 - val\_loss: 0.7306 - val\_accuracy: 0.8632

Epoch 12/20

210/215 [============================>.] - ETA: 0s - loss: 0.0338 - accuracy: 0.9982

Epoch 12: val\_accuracy did not improve from 0.86947

215/215 [==============================] - 2s 10ms/step - loss: 0.0339 - accuracy: 0.9982 - val\_loss: 0.7325 - val\_accuracy: 0.8421

Epoch 13/20

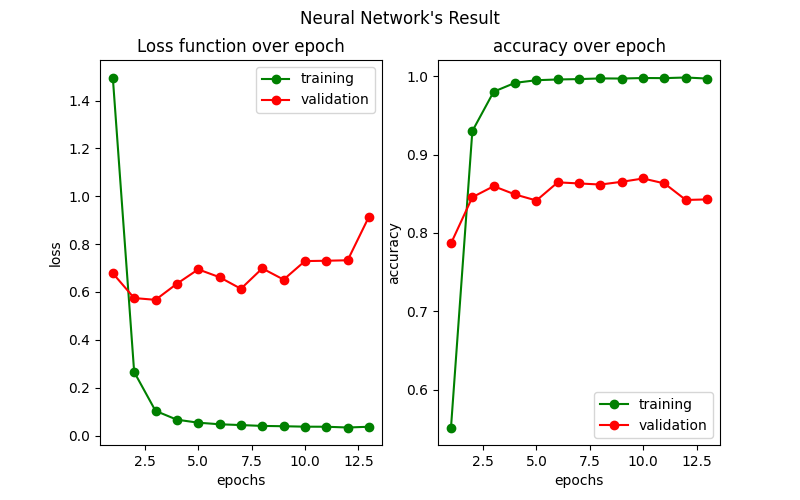
210/215 [============================>.] - ETA: 0s - loss: 0.0370 - accuracy: 0.9969

Epoch 13: val\_accuracy did not improve from 0.86947

215/215 [==============================] - 2s 10ms/step - loss: 0.0370 - accuracy: 0.9969 - val\_loss: 0.9151 - val\_accuracy: 0.8428

57/57 [==============================] - 1s 9ms/step - loss: 0.8228 - accuracy: 0.8509

Process finished with exit code 0

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F4%20Early%20stopping%2Fconv_results_early.png)

En este momento hemos guardado un modelo funcional que ha logrado obtener 0.8509 en el test set.

**3.5 kerasTuner: Construyendo el modelo**

Keras Tuner es una biblioteca de Python que ayuda a encontrar los mejores hiperparámetros para los modelos de Keras. Los hiperparámetros son configuraciones que no se aprenden durante el entrenamiento del modelo, pero que influyen en cómo se entrena el modelo y cómo se hace la predicción. Por ejemplo, el número de capas y nodos en una red neuronal, la tasa de aprendizaje, la función de activación, entre otros, son ejemplos de hiperparámetros que deben ajustarse adecuadamente para obtener un modelo preciso y eficiente.

Keras Tuner utiliza la búsqueda de hiperparámetros para encontrar los mejores valores de hiperparámetros para un modelo específico. Hay varios tipos de búsquedas que se pueden realizar con Keras Tuner, como la búsqueda aleatoria, la búsqueda en cuadrícula y la búsqueda de hiperbanda.

* La búsqueda aleatoria implica seleccionar aleatoriamente valores para cada hiperparámetro en un rango especificado.
* La búsqueda en cuadrícula implica seleccionar un conjunto discreto de valores para cada hiperparámetro y probar todas las combinaciones posibles.
* La búsqueda de hiperbanda es una combinación de las dos anteriores y se enfoca en identificar los mejores hiperparámetros mediante una eliminación temprana de modelos subóptimos.

Keras Tuner puede utilizarse con cualquier modelo de Keras, desde una simple red neuronal hasta una red neuronal convolucional o una red neuronal recurrente. Los hiperparámetros se pueden definir de forma personalizada o utilizar las funciones predefinidas de Keras Tuner.

Dentro del conjunto de HyperParametersque son óptimizables tenemos los siguientes:

* [Boolean method](https://keras.io/api/keras_tuner/hyperparameters/#boolean-method)
* [Choice method](https://keras.io/api/keras_tuner/hyperparameters/#choice-method)
* [Fixed method](https://keras.io/api/keras_tuner/hyperparameters/#fixed-method)
* [Float method](https://keras.io/api/keras_tuner/hyperparameters/#float-method)
* [Int method](https://keras.io/api/keras_tuner/hyperparameters/#int-method)

Entre otros.

Y como mencionamos anteriormente, tenemos varios tipos de tuner entre los que podemos encontrar los siguientes:

* [The base Tuner class](https://keras.io/api/keras_tuner/tuners/base_tuner)
* [Objective class](https://keras.io/api/keras_tuner/tuners/objective)
* [RandomSearch Tuner](https://keras.io/api/keras_tuner/tuners/random)
* [GridSearch Tuner](https://keras.io/api/keras_tuner/tuners/grid)
* [BayesianOptimization Tuner](https://keras.io/api/keras_tuner/tuners/bayesian)
* [Hyperband Tuner](https://keras.io/api/keras_tuner/tuners/hyperband)
* [Sklearn Tuner](https://keras.io/api/keras_tuner/tuners/sklearn)

Para más información puedes leer la documentación oficial de [Keras Tuner](https://keras.io/keras_tuner/)

Para instalar keras-tuner:

pip install keras-tuner --upgrade

**Información adicional:**

Keras-tuner NO es la única herramienta que nos permite hacer este ajuste de hiperparámetros, también existe otra herramienta llamada talos de forma similar se instala como pip install talos puedes leer la documentación oficial [Aquí](https://autonomio.github.io/talos/#/)

**Código de ejemplo:**

**Nota:**

El código completo de esta y la próxima sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F5%20Keras%20Tuner%2Fmain.py)

En esta clase nos vamos a enfocar en construir la arquitectura del modelo que nos permita ser optimizable por el Hyperband tuner

**1: Importando bibliotecas de siempre**

from keras.preprocessing.image import ImageDataGenerator

import matplotlib.pyplot as plt

import string

from keras.models import Sequential

from keras.layers import Flatten, Dense, Dropout, Conv2D, MaxPooling2D, BatchNormalization

from keras.regularizers import l2

from keras.callbacks import EarlyStopping, ModelCheckpoint

from keras.optimizers import Adam

Sin embargo, en está ocasión vamos a importar nuevas librerías que NO habíamos utilizado antes:

from abc import ABC

from keras\_tuner import HyperModel, Hyperband

import json

Nos vamos a enfocar principalmente en HyperModel y en Hyperband las cuales nos van a permitir construir una clase con la arquitectura del modelo y ciertos parámetros preestablecidos por el constructor de la clase.

**2: Definiendo la clase de la arquitectura deseada**

Por documentación de keras tuner para crear nuestra propia clase de HyperModel debemos heredar de ella y de ABC definimos nuevos parámetros que en este caso serán: input\_shape y n\_classes

class CNNArchitecture(HyperModel, ABC):

def \_\_init\_\_(self, input\_shape, n\_classes):

super().\_\_init\_\_()

self.input\_shape = input\_shape

self.n\_classes = n\_classes

De esta forma cuando creemos un objeto de la clase CNNArchitecture podremos definir esos parámetros, los cuáles no se definen por la arquitectura del modelo sino por la naturaleza del problema. En nuestro caso nuestras imágenes del dataset son de (28x28x1) y tenemos un total de 24 clases diferenciables.

**3: Construimos la arquitectura del modelo y compilamos**

El parámetro hp hace referencia a un diccionario que contenga los valores pre-cargados para los hiperparámetros. Si no se pasa un valor entonces utilizará los parámetros que tenga disponibles para entrenar.

Hasta este momento nuestra arquitectura es idéntica a las que ya hemos venido creado en clases anteriores, con la única diferencia de que ahora al ser un método de una clase accedemos al input\_shape como self.input\_shape puesto que este valor lo definimos en el constructor de la clase.

def build(self, hp):

model = Sequential()

model.add(Conv2D(filters=128, kernel\_size=(3, 3), activation="relu", input\_shape=self.input\_shape))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

Aquí empieza lo verdaderamente interesante, definir las siguientes capas NO con valores fijos sino con valores que pueden ser variados dentro de un rango predeterminado de valores. Para la siguiente capa que normalmente habiamos definido de la siguiente manera:

model.add(Dense(256, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

Podemos hacer que el número de neuronas 256 y el valor de dropout 0.2 No sean constantes si no que sean elegibles dentro de un rango conocido.

model.add(Dense(units=hp.Int("units\_1", min\_value=64, max\_value=512, step=64, default=128),

activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(rate=hp.Float("dropout\_1", min\_value=0.2, max\_value=0.6, default=0.5, step=0.10)))

model.add(BatchNormalization())

Lo más interesante es que en lugar de definir 256 neuronas para la capa Dense es que estamos proponiendo un rango de 64 a 512 en intervalos de 64, con un valor por defecto de 128. Aquí estamos ocupando el método Int pero recordemos que tenemos otros disponibles. Como es el caso de Float el cual definimos para el cambiar el rate del dropout.

Continuamos con la arquitectura de nuestro modelo:

model.add(Dense(128, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(BatchNormalization())

model.add(Dense(self.n\_classes, activation="softmax"))

Hasta aquí nada interesante, son cosas que ya entendemos y conocemos, salvo el hecho de que la última capa la encargada de clasificación está definida por self.n\_classes variable definida por el constructor de la clase.

Sin embargo, para terminar, debemos compilar el modelo:

model.compile(Adam(hp.Float("learning\_rate", min\_value=1e-4, max\_value=1e-2, sampling="LOG", default=1e-3,)),

loss="categorical\_crossentropy", metrics=['accuracy'])

# Alternativa a hp.Float -> hp.Choice('learning\_rate', values = [1e-2, 1e-3, 1e-4])

return model

Nota interesante, que el optimizador ADAM también puede verse beneficiado de keras tuner puesto que su learning rate ahora no es un solo valor si no que es un rango de valores, puesto que los valores son sumamente pequeños es necesario poner que el sampling es "LOG".

La arquitectura completa del modelo queda de la siguiente manera:

def build(self, hp):

model = Sequential()

model.add(Conv2D(filters=128, kernel\_size=(3, 3), activation="relu", input\_shape=self.input\_shape))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(units=hp.Int("units\_1", min\_value=64, max\_value=512, step=64, default=128),

activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(rate=hp.Float("dropout\_1", min\_value=0.2, max\_value=0.6, default=0.5, step=0.10)))

model.add(BatchNormalization())

model.add(Dense(128, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(BatchNormalization())

model.add(Dense(self.n\_classes, activation="softmax"))

model.compile(Adam(hp.Float("learning\_rate", min\_value=1e-4, max\_value=1e-2, sampling="LOG", default=1e-3,)),

loss="categorical\_crossentropy", metrics=['accuracy'])

# Alternativa a hp.Float -> hp.Choice('learning\_rate', values = [1e-2, 1e-3, 1e-4])

return model

**3.6 KerasTuner: Buscando la mejor configuración para tu modelo**

Cómo en la clase pasada ya hemos definido la arquitectura del modelo, en esta clase vamos a ver como utilizarla en conjunto al Hyperband para hacer nuestra búsqueda de parámetros óptimos para nuestra red:

Empezamos con lo básico: Crear las particiones de nuestros datos

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data()

Ahora podemos crear el modelo a partir de nuestra clase CNNArchitecture

cnn\_model = CNNArchitecture(input\_shape=(28, 28, 1), n\_classes=len(classes))

Aquí empieza la búsqueda de los mejores hiperparámetros, primero debemos crear un objeto tuner de la clase Hyperband con nuestro hypermodel llamado cnn\_model:

tuner = Hyperband(hypermodel=cnn\_model, objective="val\_accuracy", max\_epochs=20, factor=3, directory="models/",

project\_name="test")

Te recomiendo leer [Hyperband Tuner](https://keras.io/api/keras_tuner/tuners/hyperband/) para entender más de la documentación de este tuner sin embargo, a continuación te dejo sus parámetros disponibles.

keras\_tuner.Hyperband(

hypermodel=None,

objective=None,

max\_epochs=100,

factor=3,

hyperband\_iterations=1,

seed=None,

hyperparameters=None,

tune\_new\_entries=True,

allow\_new\_entries=True,

max\_retries\_per\_trial=0,

max\_consecutive\_failed\_trials=3,

\*\*kwargs

)

Ahora nuestro objeto tuner es un hypermodel de cnn\_model con tuner podemos empezar a hacer la busqueda de los mejores hiperparámetros:

tuner.search(train\_generator, epochs=20, validation\_data=validation\_generator)

Respuesta esperada:

Trial 29 Complete [00h 00m 44s]

val\_accuracy: 0.8933333158493042

Best val\_accuracy So Far: 0.9038596749305725

Total elapsed time: 00h 08m 02s

Search: Running Trial #30

Value |Best Value So Far |Hyperparameter

512 |512 |units\_1

0.4 |0.2 |dropout\_1

0.0021972 |0.00048085 |learning\_rate

20 |7 |tuner/epochs

0 |0 |tuner/initial\_epoch

0 |1 |tuner/bracket

0 |0 |tuner/round

Ahora nuestro hypermodel ya se encuentra entrenado, podemos acceder a los mejores hiperparámetros que encontro:

best\_hps = tuner.get\_best\_hyperparameters(num\_trials=1)[0]

print(best\_hps)

Respuesta esperada:

<keras\_tuner.engine.hyperparameters.hyperparameters.HyperParameters object at 0x7fd2d8edcdf0>

Esta configuración la podemos usar para construir un modelo:

conv\_model = tuner.hypermodel.build(best\_hps)

De esta manera conv\_model ya NO es un hypermodel ya es un modelo convencional como los que hemos estado trabajando a lo largo del curso. conv\_model ya se encuentra tuneado para tener la mejor configuración de hiperparmáetros posible, entonces es buena idea guardar la arquitectura del modelo para reutilizarla después:

config\_dict = conv\_model.get\_config()

print(config\_dict)

with open('config\_model.json', 'w') as outfile:

json.dump(config\_dict, outfile)

Respuesta esperada:

{'name': 'sequential\_1', 'layers': [{'class\_name': 'InputLayer', 'config': {'batch\_input\_shape': (None, 28, 28, 1), 'dtype': 'float32', 'sparse': False, 'ragged': False, 'name': 'conv2d\_1\_input'}}, {'class\_name': 'Conv2D', 'config': {'name': 'conv2d\_1', 'trainable': True, 'dtype': 'float32', 'batch\_input\_shape': (None, 28, 28, 1), 'filters': 128, 'kernel\_size': (3, 3), 'strides': (1, 1), 'padding': 'valid', 'data\_format': 'channels\_last', 'dilation\_rate': (1, 1), 'groups': 1, 'activation': 'relu', 'use\_bias': True, 'kernel\_initializer': {'class\_name': 'GlorotUniform', 'config': {'seed': None}}, 'bias\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'kernel\_regularizer': None, 'bias\_regularizer': None, 'activity\_regularizer': None, 'kernel\_constraint': None, 'bias\_constraint': None}}, {'class\_name': 'MaxPooling2D', 'config': {'name': 'max\_pooling2d\_1', 'trainable': True, 'dtype': 'float32', 'pool\_size': (2, 2), 'padding': 'valid', 'strides': (2, 2), 'data\_format': 'channels\_last'}}, {'class\_name': 'Flatten', 'config': {'name': 'flatten\_1', 'trainable': True, 'dtype': 'float32', 'data\_format': 'channels\_last'}}, {'class\_name': 'Dense', 'config': {'name': 'dense\_3', 'trainable': True, 'dtype': 'float32', 'units': 512, 'activation': 'relu', 'use\_bias': True, 'kernel\_initializer': {'class\_name': 'GlorotUniform', 'config': {'seed': None}}, 'bias\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'kernel\_regularizer': {'class\_name': 'L2', 'config': {'l2': 9.999999747378752e-06}}, 'bias\_regularizer': None, 'activity\_regularizer': None, 'kernel\_constraint': None, 'bias\_constraint': None}}, {'class\_name': 'Dropout', 'config': {'name': 'dropout\_2', 'trainable': True, 'dtype': 'float32', 'rate': 0.2, 'noise\_shape': None, 'seed': None}}, {'class\_name': 'BatchNormalization', 'config': {'name': 'batch\_normalization\_2', 'trainable': True, 'dtype': 'float32', 'axis': ListWrapper([1]), 'momentum': 0.99, 'epsilon': 0.001, 'center': True, 'scale': True, 'beta\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'gamma\_initializer': {'class\_name': 'Ones', 'config': {}}, 'moving\_mean\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'moving\_variance\_initializer': {'class\_name': 'Ones', 'config': {}}, 'beta\_regularizer': None, 'gamma\_regularizer': None, 'beta\_constraint': None, 'gamma\_constraint': None}}, {'class\_name': 'Dense', 'config': {'name': 'dense\_4', 'trainable': True, 'dtype': 'float32', 'units': 128, 'activation': 'relu', 'use\_bias': True, 'kernel\_initializer': {'class\_name': 'GlorotUniform', 'config': {'seed': None}}, 'bias\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'kernel\_regularizer': {'class\_name': 'L2', 'config': {'l2': 9.999999747378752e-06}}, 'bias\_regularizer': None, 'activity\_regularizer': None, 'kernel\_constraint': None, 'bias\_constraint': None}}, {'class\_name': 'Dropout', 'config': {'name': 'dropout\_3', 'trainable': True, 'dtype': 'float32', 'rate': 0.2, 'noise\_shape': None, 'seed': None}}, {'class\_name': 'BatchNormalization', 'config': {'name': 'batch\_normalization\_3', 'trainable': True, 'dtype': 'float32', 'axis': ListWrapper([1]), 'momentum': 0.99, 'epsilon': 0.001, 'center': True, 'scale': True, 'beta\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'gamma\_initializer': {'class\_name': 'Ones', 'config': {}}, 'moving\_mean\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'moving\_variance\_initializer': {'class\_name': 'Ones', 'config': {}}, 'beta\_regularizer': None, 'gamma\_regularizer': None, 'beta\_constraint': None, 'gamma\_constraint': None}}, {'class\_name': 'Dense', 'config': {'name': 'dense\_5', 'trainable': True, 'dtype': 'float32', 'units': 24, 'activation': 'softmax', 'use\_bias': True, 'kernel\_initializer': {'class\_name': 'GlorotUniform', 'config': {'seed': None}}, 'bias\_initializer': {'class\_name': 'Zeros', 'config': {}}, 'kernel\_regularizer': None, 'bias\_regularizer': None, 'activity\_regularizer': None, 'kernel\_constraint': None, 'bias\_constraint': None}}]}

Excelente, ahora tenemos un archivo json llamado config\_model.json el cual ha guarado toda la arquitectura con la mejor configuración posible para nuestro modelo.

Ahora podemos seguir con los conocimientos que ya conocemos para con base en dicha arquitectura y ya con el modelo compilado utilizarlo para entrenarlo desde 0 con nuestro train\_generator y utilizar callbacks para monitorear el mismo:

callback = EarlyStopping(monitor="val\_accuracy", patience=3, mode="auto")

checkpoint = ModelCheckpoint(filepath="models/best\_model.h5", save\_best\_only=True, save\_weights\_only=False,

mode="auto", verbose=1, monitor="val\_accuracy")

history = conv\_model.fit(train\_generator, epochs=20, validation\_data=validation\_generator, batch\_size=128,

callbacks=[callback, checkpoint])

Respuesta esperada:

Epoch 1/20

212/215 [============================>.] - ETA: 0s - loss: 0.7691 - accuracy: 0.8091

Epoch 1: val\_accuracy improved from -inf to 0.65333, saving model to models/best\_model.h5

215/215 [==============================] - 3s 11ms/step - loss: 0.7590 - accuracy: 0.8117 - val\_loss: 2.4074 - val\_accuracy: 0.6533

Epoch 2/20

211/215 [============================>.] - ETA: 0s - loss: 0.0335 - accuracy: 0.9997

Epoch 2: val\_accuracy improved from 0.65333 to 0.84982, saving model to models/best\_model.h5

215/215 [==============================] - 2s 11ms/step - loss: 0.0333 - accuracy: 0.9997 - val\_loss: 1.1209 - val\_accuracy: 0.8498

Epoch 3/20

213/215 [============================>.] - ETA: 0s - loss: 0.0193 - accuracy: 1.0000

Epoch 3: val\_accuracy improved from 0.84982 to 0.88561, saving model to models/best\_model.h5

215/215 [==============================] - 2s 11ms/step - loss: 0.0193 - accuracy: 1.0000 - val\_loss: 0.4633 - val\_accuracy: 0.8856

Epoch 4/20

214/215 [============================>.] - ETA: 0s - loss: 0.0152 - accuracy: 1.0000

Epoch 4: val\_accuracy improved from 0.88561 to 0.89263, saving model to models/best\_model.h5

215/215 [==============================] - 2s 11ms/step - loss: 0.0152 - accuracy: 1.0000 - val\_loss: 0.4295 - val\_accuracy: 0.8926

Epoch 5/20

215/215 [==============================] - ETA: 0s - loss: 0.0131 - accuracy: 1.0000

Epoch 5: val\_accuracy did not improve from 0.89263

215/215 [==============================] - 2s 10ms/step - loss: 0.0131 - accuracy: 1.0000 - val\_loss: 0.4467 - val\_accuracy: 0.8842

Epoch 6/20

214/215 [============================>.] - ETA: 0s - loss: 0.0121 - accuracy: 1.0000

Epoch 6: val\_accuracy did not improve from 0.89263

215/215 [==============================] - 2s 10ms/step - loss: 0.0121 - accuracy: 1.0000 - val\_loss: 0.4593 - val\_accuracy: 0.8786

Epoch 7/20

215/215 [==============================] - ETA: 0s - loss: 0.0113 - accuracy: 1.0000

Epoch 7: val\_accuracy did not improve from 0.89263

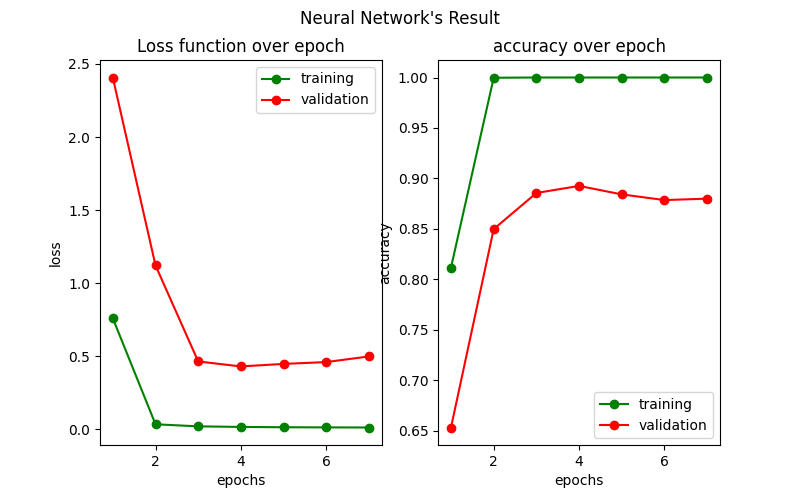
215/215 [==============================] - 2s 10ms/step - loss: 0.0113 - accuracy: 1.0000 - val\_loss: 0.4980 - val\_accuracy: 0.8800

Graficamos los resultados y evaluamos en el test\_generator

plot\_results(history, "accuracy", "hypermodel\_results.png")

results = conv\_model.evaluate(test\_generator)

Respuesta esperada:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F5%20Keras%20Tuner%2Fhypermodel_results.png)

57/57 [==============================] - 1s 19ms/step - loss: 0.5009 - accuracy: 0.8783

Genial, nos ha ido bastante bien, recordemos que en la última clase, nuestro último resultado de accuracy había sido el siguiente:

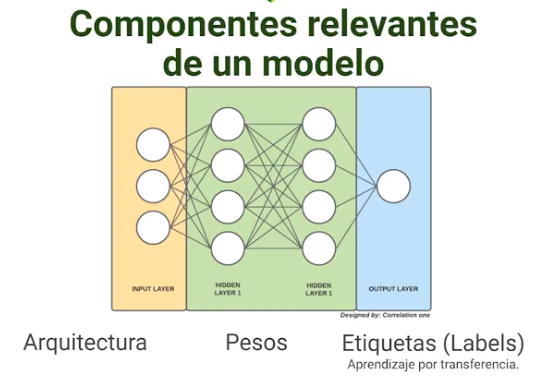
57/57 [==============================] - 1s 9ms/step - loss: 0.8228 - accuracy: 0.8509

Hemos mejorado más de 2.5% únicamente cambiando una configuración de hiperparámetros del modelo.

Nota: esta arquitectura también utilizo BatchNormalization() pero en general se entiende el punto de la clase. La clase original no utilizaba este método, yo lo he implementado buscando reducir el overfitting del modelo.

**4 Almacenamiento y carga de modelos**

En las próximas clases estaremos observando las formas en las que podemos almacenar nuestras redes neuronales.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F4%2F1.png)

Por un lado, podemos guardar únicamente la arquitectura del modelo, o solamente sus pesos, o una combinación de ambas para tener el modelo completamente funcional y listo para llevar a deploy. A lo largo de estas notas y gracias a cursos anteriores ya hemos utilizado callbacks de checkpoints por lo cual estas 2 clases no son muy relevantes, sin embargo; vamos a poner ejemplos pequeños que nos sirvan de demostraciones teóricas como guardar nuestros modelos.

**4.1 Almacenamiento y carga de modelos: pesos y arquitectura**

**Nota:**

El código completo lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/4%20Almacenamiento%20y%20carga%20del%20modelo%2F1%20Almacenamiento%20de%20pesos%20y%20arquitectura%2Fmain.py)

Primero importemos nuevas bibliotecas:

import json

from keras import Model

Estas serán usadas para guardar la arquitectura del modelo y cargar apropiadamente el modelo.

Ahora podemos definir sencillamente la arquitectura del modelo:

def conv\_architecture(input\_shape, n\_clases):

model = Sequential()

model.add(Conv2D(filters=128, kernel\_size=(3, 3), activation="relu", input\_shape=input\_shape))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(128, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(n\_clases, activation="softmax"))

print(model.summary())

return model

Y aquí viene la parte interesante, vamos a crear una función para generar los archivos de arquitectura y pesos del modelo:

def generate\_weights(classes, batch\_size, train\_generator, validation\_generator, test\_generator):

conv\_model = conv\_architecture(input\_shape=(28, 28, 1), n\_clases=len(classes))

config\_dict = conv\_model.get\_config()

with open('config\_model.json', 'w') as outfile:

json.dump(config\_dict, outfile)

conv\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

#

conv\_model.fit(train\_generator, epochs=5, validation\_data=validation\_generator, batch\_size=batch\_size)

conv\_model.evaluate(test\_generator)

conv\_model.save\_weights('only\_weights.h5')

Una vez creado el conv\_model desde la función conv\_architecture podemos acceder a su configuración con el método get\_config() esto nos regresa un diccionario con la configuración del modelo el cual podemos guardar en formato json, en este caso con el nombre de config\_model.json. Después debemos compilar el modelo y ponerlo a entrenar como siempre lo hemos hecho. Finalmente podemos evaluar el rendimiento del modelo y salvar los pesos con el método save\_weights(weights\_name.h5)

Respuesta esperada:

Epoch 1/5

215/215 [==============================] - 4s 10ms/step - loss: 1.6137 - accuracy: 0.5126 - val\_loss: 0.8008 - val\_accuracy: 0.7186

Epoch 2/5

215/215 [==============================] - 2s 9ms/step - loss: 0.3274 - accuracy: 0.9052 - val\_loss: 0.6053 - val\_accuracy: 0.8239

Epoch 3/5

215/215 [==============================] - 2s 10ms/step - loss: 0.1269 - accuracy: 0.9732 - val\_loss: 0.6527 - val\_accuracy: 0.8351

Epoch 4/5

215/215 [==============================] - 2s 9ms/step - loss: 0.0753 - accuracy: 0.9874 - val\_loss: 0.6898 - val\_accuracy: 0.8400

Epoch 5/5

215/215 [==============================] - 2s 10ms/step - loss: 0.0600 - accuracy: 0.9913 - val\_loss: 0.6979 - val\_accuracy: 0.8337

57/57 [==============================] - 1s 9ms/step - loss: 0.6454 - accuracy: 0.8458

Ahora podemos crear una función que cargue los pesos y la arquitectura del modelo:

def load\_model\_from\_architecture\_and\_weights(architecture, weights) -> Model:

json\_config = json.load(open(architecture))

new\_model = Sequential.from\_config(json\_config)

new\_model.load\_weights(weights)

new\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

return new\_model

Vemos que algo que en ningún momento se guardó fue la información de compilado del modelo, entonces debemos volver a compilar el modelo idealmente de la misma forma en la que lo guardamos.

Probamos el código:

loaded\_model = load\_model\_from\_architecture\_and\_weights("config\_model.json", "only\_weights.h5")

print("from loaded:")

loaded\_model.evaluate(test\_generator)

Respuesta esperada:

from loaded:

57/57 [==============================] - 1s 9ms/step - loss: 0.6454 - accuracy: 0.8458

Excelente, ambos obtienen exactamente el mismo resultado.

**4.2 Criterios para almacenar los modelos**

En la práctica real, normalmente NO almacenamos por separado la arquitectura de los pesos del modelo. Keras ofrece varias formas de almacenar esta información, por un lado, tenemos los métodos save() y load\_model() que nos ayudaran a guardar y cargar nuestros modelos completos. Por otro lado, podemos usar el Callback llamado ModelCheckpoint para ir guardando el modelo cada que durante una epoch consigue obtener mejores resultados. Veamos el caso de ejemplo básico:

Importamos las siguientes bibliotecas para usar correctamente el guardado de modelos:

from keras.callbacks import ModelCheckpoint

from keras import Model

import keras.models

Podemos usar exactamente la misma arquitectura que la vez pasada:

def conv\_architecture(input\_shape, n\_clases):

model = Sequential()

model.add(Conv2D(filters=128, kernel\_size=(3, 3), activation="relu", input\_shape=input\_shape))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(128, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(n\_clases, activation="softmax"))

print(model.summary())

return model

Y ahora hacemos una función para guardar dos modelos best y last

def save\_model(classes, batch\_size, train\_generator, validation\_generator, test\_generator):

conv\_model = conv\_architecture(input\_shape=(28, 28, 1), n\_clases=len(classes))

checkpoint = ModelCheckpoint(filepath="models/best\_model.h5", frecuency="epoch", save\_weights\_only=False,

monitor="val\_accuracy", save\_best\_only=True, verbose=1)

conv\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

conv\_model.fit(train\_generator, epochs=10, validation\_data=validation\_generator, batch\_size=batch\_size,

callbacks=[checkpoint])

conv\_model.evaluate(test\_generator)

conv\_model.save("models/last\_model.h5")

Para guardar best usamos el ModelCheckpoint es importante tener activada en True el parámetro save\_best\_only y para asegurarnos de que se está guardando el modelo completo ponemos en False el parámetro save\_only\_weights

Recordemos que NO necesariamente cada epoch sí o sí va a mejorar el entrenamiento general, esto significa que el modelo generado en la última capa NO necesariamente es el que da el mejor resultado, es una buena práctica guardar de forma manual el último modelo producido con: conv\_model.save(models/last\_model.h5

Ahora podemos hacer una función para cargar los modelos de best y last:

def load\_model(model: str) -> Model:

if model == "best":

return keras.models.load\_model("models/best\_model.h5")

elif model == "last":

return keras.models.load\_model("models/last\_model.h5")

else:

print("You must select (best or last)")

return

Podemos ejecutar el código completo:

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data()

save\_model(classes, batch\_size, train\_generator, validation\_generator, test\_generator)

best\_model = load\_model(model="best")

print("Best:")

best\_model.evaluate(test\_generator)

last\_model = load\_model(model="last")

print("Last")

last\_model.evaluate(test\_generator)

Respuesta esperada:

Best:

57/57 [==============================] - 1s 10ms/step - loss: 0.4993 - accuracy: 0.8836

Last

57/57 [==============================] - 1s 9ms/step - loss: 0.6007 - accuracy: 0.8656

Cómo es de esperarse, el mejor modelo tuvo un mejor desempeño que el último.

**5 Fundamentos de aprendizaje por transferencia**

Lo que estaremos aprendiendo a lo largo de este módulo es:

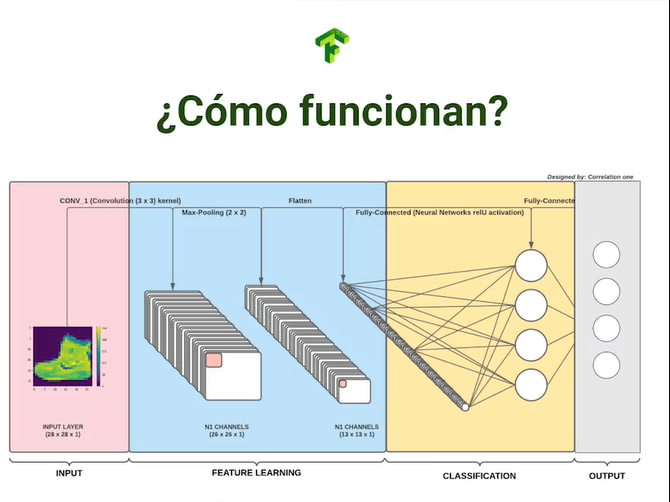
* Introducción al aprendizaje por transferencia
* Cómo cargar sistemas pre-entrenados en Keras
* Cómo utilizar sistemas pre-entrenados desde los repositorios de TensorFlow Hub
* Aplicar modelo pre-entrenado a nuestro proyecto

**5.1 Introducción al aprendizaje por transferencia**

Transfer learning es una técnica de Deep Learning en la que se utiliza un modelo pre-entrenado como punto de partida para un nuevo problema (clasificación, detección, regresiones) etc. En lugar de entrenar un modelo desde cero, se usa un modelo que ha sido entrenado previamente en una gran cantidad de datos y se ajusta para adaptarlo a una tarea específica (Fine tuning).

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F4%2F2.png)

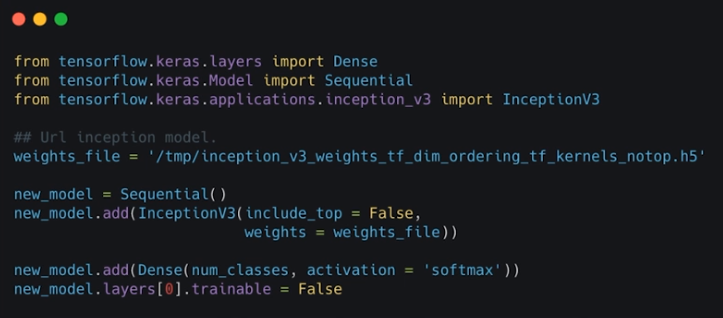
En Deep Learning, los modelos pre-entrenados se suelen entrenar en grandes conjuntos de datos, como ImageNet, que contiene millones de imágenes de objetos clasificados en miles de categorías. El objetivo de entrenar en un conjunto de datos tan grande es aprender características generales que se pueden aplicar a una amplia variedad de tareas, no sólo a la tarea específica del conjunto de datos.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F4%2F3.png)

Una vez que se entrena el modelo pre-entrenado, se utiliza para extraer características de nuevas imágenes o datos relacionados con la tarea específica. Estas características se usan como entrada para un nuevo modelo que se entrena específicamente para la tarea de interés. El nuevo modelo puede ser una red neuronal completamente conectada o una red neuronal convolucional que se entrena sólo en las últimas capas.

La ventaja del transfer learning es que permite el uso de modelos pre-entrenados que han demostrado ser efectivos en tareas relacionadas. En lugar de tener que entrenar un modelo completamente desde cero, se puede utilizar un modelo que ya ha aprendido características generales que pueden ser útiles para la tarea de interés. Esto puede reducir significativamente el tiempo y los recursos necesarios para entrenar un modelo, y a menudo resulta en una mejor precisión.

**Un ejemplo sencillo**

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F4%2F4.png)

Usar transfer learning puede ser tan sencillo como tener descargado en nuestra máquina local un archivo de pesos de un modelo de nuestro interés. En este ejemplo InceptionV3 creamos un new\_model y a la primera capa le vamos a añadir el modelo InceptionV3 con sus respectivos pesos, cabe mencionar que es importante NO incluir la última capa include\_top = False esto es importante, porque lo que nosotros queremos es adaptar estos pesos ya aprendidos a nuestro nuevo problema.

Podemos continuar con la arquitectura que nosotros propongamos, o en este caso agregar la última capa correspondiente a la capa de clasificación. También es MUY importante decirle a keras que la layer[0] NO debe ser entrenable, esta se debe quedar como ya estaba para cumplir con el propósito de transfer learning que es NO reentrenar el modelo desde 0.

**5.2 Cuándo utilizar aprendizaje por transferencia**

Podemos usar modelos pre-entrenados cuando tratemos problemas de procesamiento de lenguaje natural y visión computarizada, pasa ambos casos se suelen implementar arquitecturas robustas con altísimas cantidades de iteraciones, por lo que siempre será ideal dedicar tiempo a investigar qué configuraciones se han implementado similares a tu caso de uso.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F4%2F5.png)

**¿Por qué usar aprendizaje por transferencia?**

El aprendizaje por transferencia será especialmente útil cuando tengas muy pocos datos, dado que no tendrás que enseñar al modelo desde 0 las abstracciones, también te permitirá generar iteraciones muy rápidas (en caso de que debas generar un prototipo en poco tiempo o quieras tantear la calidad de tus datos). Estos modelos ya han generalizado las features, por lo que se podrán adaptar a tus necesidades en pocas iteraciones.

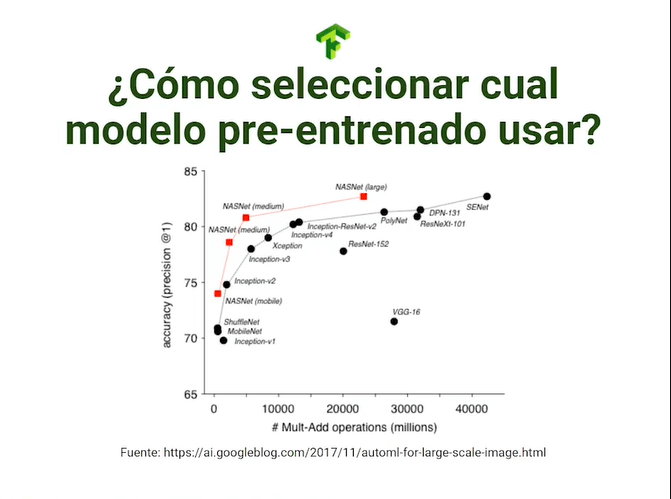
[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F4%2F6.png)

Si deseas enriquecerte con la documentación de algunas implementaciones, puedes leer los paper de [YOLO V3](https://pjreddie.com/media/files/papers/YOLOv3.pdf) y [AlexNet](https://cvml.ista.ac.at/courses/DLWT_W17/material/AlexNet.pdf), algunas de las configuraciones más usadas en redes convolucionales.

**¿Cómo saber cuál modelo seleccionar?**

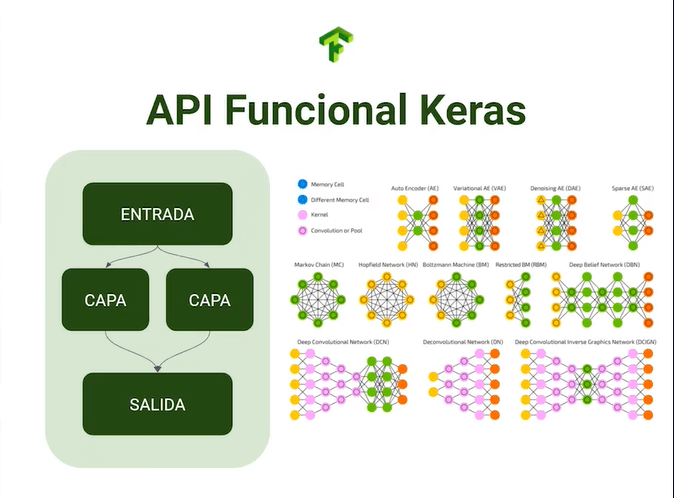
Las 2 métricas a seguir a la hora de seleccionar un modelo serán las de precisión y complejidad, donde según tu contexto deberás elegir cuál es más relevante. Si tu modelo requiere de reacción rápida entonces podrás sacrificar un poco de precisión por velocidad (este es el ejemplo de detección de objetos en vivo, como cámaras de seguridad o vehículos autónomos).

Si la precisión lo es todo (como en la clasificación de células cancerígenas) puedes darte el lujo de correr un modelo por bastante tiempo con el fin de obtener resultados precisos. Puedes guiarte en la noción de precisión vs cantidad de operaciones para elegir tu modelo, como siempre, tú determinarás las prioridades de tu modelo.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F4%2F7.png)

**5.3 Carga de sistemas pre-entrenados en Keras**

Existen múltiples métodos para cargar sistemas pre-entrenados, para esta ocasión interiorizaremos en el de Keras. Hasta ahora hemos trabajado con la API secuencial de Keras que nos permite apilar capas de manera lineal, su recorrido es unidireccional y siempre el mismo. La API Funcional de Keras rompe este paradigma y nos permite configurar redes a gusto, donde los saltos de capas pueden ser alterados y se pueden implementar configuraciones infinitamente más complejas.

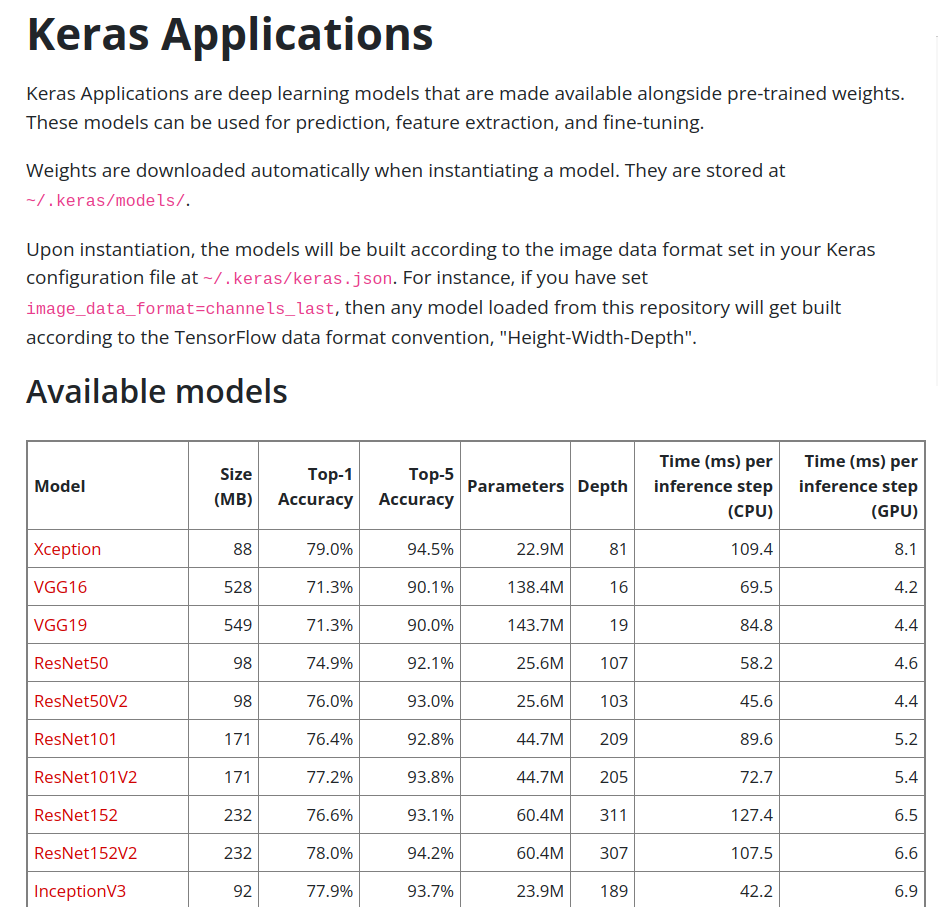
[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F1.png)

Estos modelos funcionales pueden dar mejores resultados que sus homólogos secuenciales, por lo que indagaremos como cargarlos y usarlos desde Keras.

**Usando la API funcional de Keras**

Antes de cargar la configuración es de vital importancia reconocer las dimensiones de entrada del modelo a usar, esto es especialmente necesario en contextos de detección de objetos y transfer learning. No importa si nuestro dataset consta de imágenes en 4k, si el modelo espera un input de 300x300 debemos redimensionarlo, inclusive si perdemos features.

Quieres conocer más modelos pre-entrenados, entonces date una vuelta por [Keras Applications](https://keras.io/api/applications/)

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F2.png)

Para esta ocasión usaremos la arquitectura [inception V3](https://keras.io/api/applications/inceptionv3/) que podrás encontrar en la documentación de Keras, puedes notar que sus dimensiones de entrada son (150, 150, 3) y nuestras imágenes son (28, 28, 1), por lo que debemos cambiar su tamaño y escala de colores (escalar imágenes hacia arriba no es recomendable dada la pérdida de información, por lo que siempre deberías buscar un modelo que se adapte a las dimensiones de tu dataset).

InceptionV3 Function

tf.keras.applications.InceptionV3(

include\_top=True,

weights="imagenet",

input\_tensor=None,

input\_shape=None,

pooling=None,

classes=1000,

classifier\_activation="softmax",

)

**Nota:**

El código completo de estas secciones lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/5%20Fundamentos%20de%20aprendizaje%20por%20transferencia%2F3%20Carga%20de%20sistemas%20pre-entrenados%2Fmain.py)

Empecemos importando las bibliotecas necesarias:

from keras import Model

from keras.layers import Dense, Dropout, Input, Flatten

from keras.applications.inception\_v3 import InceptionV3

from keras.callbacks import ModelCheckpoint

Lo más importante a resaltar aquí es que estamos usando un modelo pre-entrenado desde keras.applications en este caso InceptionV3 y que estamos importando a Modeldesde keras, este nos permitirá acceder al fucntional API de keras.

Conozcamos a [InceptionV3](https://keras.io/api/applications/inceptionv3/)

**Input Shape**

input\_shape: Optional shape tuple, only to be specified if include\_top is False (otherwise the input shape has to be

(299, 299, 3) (with channels\_last data format) or (3, 299, 299) (with channels\_first data format). It should have exactly

3 inputs channels, and width and height should be no smaller than 75. E.g. (150, 150, 3) would be one valid value.

input\_shape will be ignored if the input\_tensor is provided.

En términos sencillos nos está diciendo que el modelo fue creado con unas dimensiones de (299\*299\*3) y que las nuevas imágenes NO deben ser menores a 75\*75 y debe ser una imagen RGB, para nuestro código ocuparemos una dimensión de (150\*150\*3)

Así que lo primero que debemos hacer es modificar nuestra función get\_data() para cambiar el target\_size y el color\_mode de las imágenes del dataset de entrenamiento, validación y prueba.

def get\_data(target\_size, color\_mode):

train\_dir = "../../data/Train"

test\_dir = "../../data/Test"

\_bs = 128

train\_datagen = ImageDataGenerator(rescale=1 / 255)

test\_datagen = ImageDataGenerator(rescale=1 / 255, validation\_split=0.2)

\_train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=target\_size,

batch\_size=\_bs,

class\_mode="categorical",

color\_mode=color\_mode,

subset="training"

)

\_validation\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=target\_size,

batch\_size=\_bs,

class\_mode="categorical",

color\_mode=color\_mode,

subset="validation"

)

\_test\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=target\_size,

batch\_size=\_bs,

class\_mode="categorical",

color\_mode= color\_mode

)

\_classes = [char for char in string.ascii\_uppercase if char not in ("J", "Z")]

return \_classes, \_bs, \_train\_generator, \_validation\_generator, \_test\_generator

Perfecto, ahora nuestra función get\_data() está parametrizada con los argumentos: target\_size, color\_mode

Llamamos a la función para generar los datos:

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data(target\_size=(150, 150),

color\_mode="rgb"

Respuesta esperada:

Found 27455 images belonging to 24 classes.

Found 1425 images belonging to 24 classes.

Found 7172 images belonging to 24 classes.

Ahora creemos una función para cargar a nuestro modelo pre-entrenado InceptionV3:

def load\_inception\_v3(shape, include\_until: str):

Nuestra función load\_inception\_v3 por el momento tendrá únicamente 2 parámetros:

* shape: tamaño de las imágenes de entrenamiento, en este caso será (150x150x3)
* include\_until: Hasta que capa de la arquitectura del modelo original deseamos incluir para usarla en nuestra nueva red.

def load\_inception\_v3(shape, include\_until: str):

pretrained = InceptionV3(include\_top=False, input\_tensor=Input(shape=shape))

print(pretrained.summary())

Ahora podemos cargar el modelo al crear una instancia de la clase InceptionV3 llamada pretrained, es importante decir que NO queremos incluir la capa de clasificación del modelo por eso ponemos en False al parámetro include\_top. Podemos ver un resumen de la arquitectura:

Respuesta esperada:

Model: "inception\_v3"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param # Connected to

==================================================================================================

input\_1 (InputLayer) [(None, 150, 150, 3 0 []

)]

conv2d (Conv2D) (None, 74, 74, 32) 864 ['input\_1[0][0]']

batch\_normalization (BatchNorm (None, 74, 74, 32) 96 ['conv2d[0][0]']

alization)

activation (Activation) (None, 74, 74, 32) 0 ['batch\_normalization[0][0]']

conv2d\_1 (Conv2D) (None, 72, 72, 32) 9216 ['activation[0][0]']

batch\_normalization\_1 (BatchNo (None, 72, 72, 32) 96 ['conv2d\_1[0][0]']

rmalization)

...

activation\_93 (Activation) (None, 3, 3, 192) 0 ['batch\_normalization\_93[0][0]']

mixed10 (Concatenate) (None, 3, 3, 2048) 0 ['activation\_85[0][0]',

'mixed9\_1[0][0]',

'concatenate\_1[0][0]',

'activation\_93[0][0]']

==================================================================================================

Total params: 21,802,784

Trainable params: 21,768,352

Non-trainable params: 34,432

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

La arquitectura del modelo es MUY larga como para que valga la pena ponerla completa en este Readme, sin embargo, si quieres ver la arquitectura completa puedes consultarlo en el siguiente documento: [InceptionV3\_full\_arc.txt](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/5%20Fundamentos%20de%20aprendizaje%20por%20transferencia%2F3%20Carga%20de%20sistemas%20pre-entrenados%2FInceptionV3_full_arc.txt)

El modelo es gigantesco, Tiene más de 21 Millones de parámetros entrenables, pero nosotros NO queremos reentrenar al modelo, por el contrario, queremos usar los pesos que YA conoce y aprovechar ese conocimiento adquirido en nuestro nuevo problema de clasificación. De forma sencilla lo que podemos hacer es que todas las capas de este modelo NO sean entrenables, de modo que se conservan los pesos asignados:

for layer in pretrained.layers:

layer.trainable = False

Excelente, hasta este momento ya tenemos un modelo fijo, pero nosotros queremos por ejemplo NO incluir el modelo completo sino hasta una cierta capa en específico, para ello debemos hacer la siguiente configuración:

last\_layers = pretrained.get\_layer(include\_until)

Le estamos pidiendo al modelo que nos devuelva la capa cuya valor está en la variable include\_until que más adelante decidiremos que el valor de está variable será mixed7. Ahora MUY importante, el modelo funcional, necesita operar NO directamente con las capas, sino con las salidas de las capas, es por ello que debemos asignar la salida de esta última capa:

last\_output = last\_layers.output

Veamos todo el código hasta este momento:

def load\_inception\_v3(shape, include\_until: str):

pretrained = InceptionV3(include\_top=False, input\_tensor=Input(shape=shape))

print(pretrained.summary())

for layer in pretrained.layers:

layer.trainable = False

last\_layers = pretrained.get\_layer(include\_until)

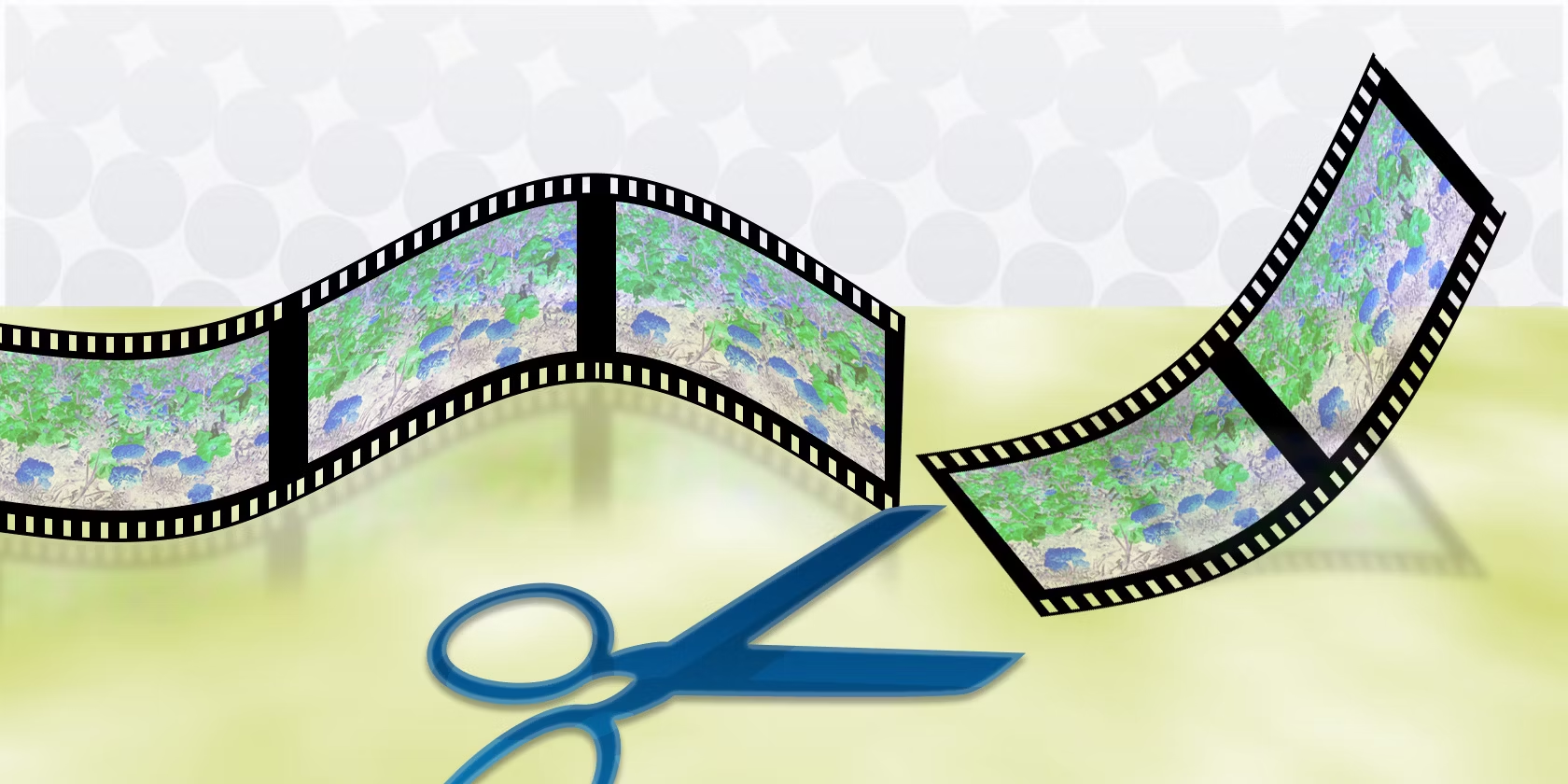
last\_output = last\_layers.output

return pretrained, last\_output

Hasta este momento hemos definido nuestra función load\_incetion\_v3 la cuál nos permite cargar el modelo completo, y también decidir desde qué capa vamos a "recortar" el modelo para añadir la nuestra:

Una simple, pero útil analogía de lo que hemos hecho hasta ahorita es la siguiente:

Podemos imaginarnos a InceptionV3 como una tira de video completa, y nuestro deber es editarla, removiendo la parte final del contenido, puesto que no la necesitamos.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F3.png)

Despues podemos crear nuestro propio "video" en este caso continuar con la arquitectura de nuestro nuevo modelo y finalmente unir ambas arquitecturas:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F4.png)

Esto es indispensable entender para manejar correctamente el Functional API de keras, puesto que esta herramienta no trabaja necesariamente forma secuencial, puede dar saltos de ser necesario, por eso es necesario indicar de dónde viene y a dónde va. Pero esto lo veremos más a detalle en la siguiente clase.

**5.4 API funcional de Keras**

Una vez que ya hemos definido nuestra función de carga del modelo pre-entrenado es momento de definir nuestra propuesta de arquitectura nueva que utilizará transfer learning para ganar las características que ya conocía el primer modelo. Empecemos por definir una función que utilice functional API de Keras:

def functional\_architecture(input\_shape, n\_clases):

# Loading pretrained model

pretrained\_model, last\_output = load\_inception\_v3(input\_shape, "mixed7")

Para este momento debería ser claro, que lo que estamos haciendo aquí es empezar nuestra nueva arquitectura tomando como base la arquitectura de InceptionV3 tomando hasta la capa mixed7. Ahora viene la parte de adjuntarle nuestra nueva arquitectura:

# Attaching our new architecture

new\_architecture = Flatten()(last\_output)

new\_architecture = Dense(128, activation="relu")(new\_architecture)

new\_architecture = Dropout(0.2)(new\_architecture)

new\_architecture = Dense(n\_clases, activation="softmax")(new\_architecture)

Excelente, vemos como en realidad nuestra porción de arquitectura es MUY simple, y básicamente solo está enfocada en la última etapa del modelo, la clasificación de las 24 clases que tiene nuestro problema. Podemos observar que estamos siguiendo un enfoque Funcional y NO Secuencial porque NO definimos un model.Sequential() y tampoco estamos añadiendo cada capa con model.add() si no que cada capa es una función, y estamos usando la función de una capa como entrada para una nueva capa, de esta específica forma si tiene mucha forma de secuencial sin embargo, veamos donde está la parte más interesante de esta Functional API, para poder convertir estas capas en un modelo, es necesario usar la clase Model() que importamos y que no habíamos utilizado antes.

model = Model(pretrained\_model.input, new\_architecture)

print(model.summary())

Aquí estamos diciéndole a Keras que queremos crear un nuevo Modelo, cuya primer capa sea la de pretrained\_model.input (por eso era necesario que en esta capa definiéramos el input\_shape) y cuya última capa del modelo sea la definida en new\_architecture la cual nosotros definimos como una cada Dense de clasificación.

El código completo de esta función queda de la siguiente manera:

def functional\_architecture(input\_shape, n\_clases):

# Loading pretrained model

pretrained\_model, last\_output = load\_inception\_v3(input\_shape, "mixed7")

# Attaching our new architecture

new\_architecture = Flatten()(last\_output)

new\_architecture = Dense(128, activation="relu")(new\_architecture)

new\_architecture = Dropout(0.2)(new\_architecture)

new\_architecture = Dense(n\_clases, activation="softmax")(new\_architecture)

model = Model(pretrained\_model.input, new\_architecture)

print(model.summary())

return model

Respuesta esperada:

Model: "model"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param # Connected to

==================================================================================================

input\_1 (InputLayer) [(None, 150, 150, 3 0 []

)]

conv2d (Conv2D) (None, 74, 74, 32) 864 ['input\_1[0][0]']

...

mixed7 (Concatenate) (None, 7, 7, 768) 0 ['activation\_60[0][0]',

'activation\_63[0][0]',

'activation\_68[0][0]',

'activation\_69[0][0]']

flatten (Flatten) (None, 37632) 0 ['mixed7[0][0]']

dense (Dense) (None, 128) 4817024 ['flatten[0][0]']

dropout (Dropout) (None, 128) 0 ['dense[0][0]']

dense\_1 (Dense) (None, 24) 3096 ['dropout[0][0]']

==================================================================================================

Total params: 13,795,384

Trainable params: 4,820,120

Non-trainable params: 8,975,264

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

De nuevo, la arquitectura real es bastante amplia, no tan amplia como la original porque eliminamos varias capas, pero si tienes curiosidad puedes ver la arquitectura completa en el siguiente documento: [transfer\_learning\_arc.txt](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/5%20Fundamentos%20de%20aprendizaje%20por%20transferencia%2F3%20Carga%20de%20sistemas%20pre-entrenados%2Ftransfer_learning_arc.txt)

Esta función devuelve la arquitectura de nuestro nuevo modelo, utilizando el Functional API y transfer learning pero a este punto ya es una arquitectura como las que hemos manejado en el curso, así que vamos a compilarla, entrenarla y ponerle callbacks para guardar el modelo:

functional\_model = functional\_architecture(input\_shape=(150, 150, 3), n\_clases=len(classes\_))

functional\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

checkpoint = ModelCheckpoint(filepath="models/best\_model.h5", frecuency="epoch", save\_weights\_only=False,

monitor="val\_accuracy", save\_best\_only=True, verbose=1)

history = functional\_model.fit(train\_generator\_, epochs=5, validation\_data=validation\_generator\_,

batch\_size=batch\_size\_, callbacks=[checkpoint])

plot\_results(history, "accuracy", "pretrained results.png")

functional\_model.evaluate(test\_generator\_)

Respuesta esperada:

Epoch 1/5

215/215 [==============================] - ETA: 0s - loss: 0.1294 - accuracy: 0.9632

Epoch 1: val\_accuracy improved from -inf to 0.98386, saving model to models/best\_model.h5

215/215 [==============================] - 19s 69ms/step - loss: 0.1294 - accuracy: 0.9632 - val\_loss: 0.0554 - val\_accuracy: 0.9839

Epoch 2/5

215/215 [==============================] - ETA: 0s - loss: 0.0030 - accuracy: 0.9998

Epoch 2: val\_accuracy did not improve from 0.98386

215/215 [==============================] - 12s 58ms/step - loss: 0.0030 - accuracy: 0.9998 - val\_loss: 0.0514 - val\_accuracy: 0.9832

Epoch 3/5

215/215 [==============================] - ETA: 0s - loss: 0.0104 - accuracy: 0.9975

Epoch 3: val\_accuracy did not improve from 0.98386

215/215 [==============================] - 12s 57ms/step - loss: 0.0104 - accuracy: 0.9975 - val\_loss: 0.0784 - val\_accuracy: 0.9754

Epoch 4/5

215/215 [==============================] - ETA: 0s - loss: 0.0038 - accuracy: 0.9989

Epoch 4: val\_accuracy did not improve from 0.98386

215/215 [==============================] - 12s 57ms/step - loss: 0.0038 - accuracy: 0.9989 - val\_loss: 0.0777 - val\_accuracy: 0.9775

Epoch 5/5

215/215 [==============================] - ETA: 0s - loss: 5.7379e-04 - accuracy: 0.9999

Epoch 5: val\_accuracy did not improve from 0.98386

215/215 [==============================] - 12s 58ms/step - loss: 5.7379e-04 - accuracy: 0.9999 - val\_loss: 0.0621 - val\_accuracy: 0.9796

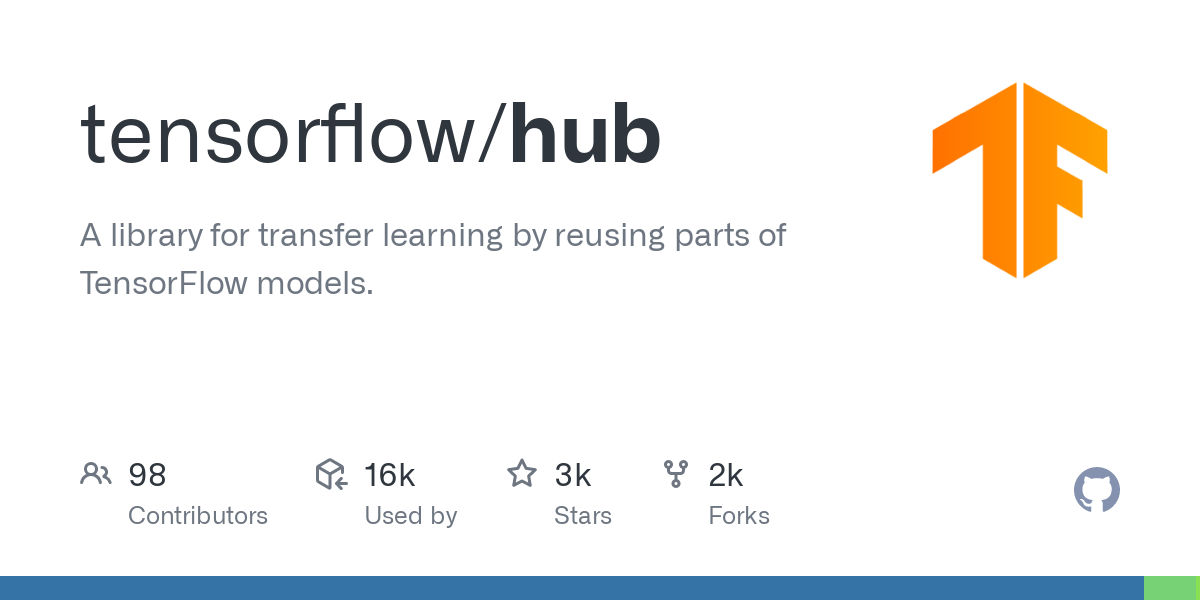
57/57 [==============================] - 3s 59ms/step - loss: 0.0649 - accuracy: 0.9791

Los resultados son excelentes, en solo 5 iteraciones, hemos logrado un val accuracy de casi 98% mismo que se ve reflejado en el test set. Y todo esto gracias a que hemos utilizado un modelo pre-entrenado y hemos ocupado el api funcional de keras para usar transfer learning.

**5.5 Uso de sistemas pre-entrenados de TensorFlow Hub**

**Introducción:**

TensorFlow Hub es una biblioteca de TensorFlow que permite a los desarrolladores compartir y reutilizar módulos de aprendizaje profundo (también conocidos como "módulos de TensorFlow") que pueden utilizarse como bloques de construcción en modelos de aprendizaje automático.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F5.png)

Los módulos de TensorFlow son bloques de construcción de aprendizaje profundo que contienen una o varias capas que han sido entrenadas para resolver una tarea específica, como la clasificación de imágenes o el reconocimiento de voz. Los módulos pueden ser utilizados como bloques de construcción para crear modelos de aprendizaje profundo personalizados, sin la necesidad de entrenarlos desde cero.

Los módulos de TensorFlow Hub están disponibles en diferentes formatos, incluyendo TensorFlow SavedModel, Keras y TensorFlow.js, lo que los hace compatibles con diferentes entornos de implementación. Puedes consultar más en su página oficial: <https://www.tensorflow.org/hub?hl=es-419>

TensorFlow Hub también proporciona una plataforma en línea donde los desarrolladores pueden compartir y buscar módulos de TensorFlow pre-entrenados. Esto permite a los desarrolladores ahorrar tiempo y recursos al utilizar módulos de TensorFlow que ya han sido entrenados y ajustados para tareas específicas. Los desarrolladores pueden buscar módulos por tarea, como clasificación de imágenes o reconocimiento de voz, o por arquitectura de modelo, como redes neuronales convolucionales o redes neuronales recurrentes.

Además, TensorFlow Hub proporciona herramientas para personalizar y ajustar los módulos de TensorFlow pre-entrenados para tareas específicas. Los desarrolladores pueden ajustar las capas de un módulo para adaptarlas a un conjunto de datos específico o combinar varios módulos para crear un modelo más complejo.

**Nota:**

El código de esta sección lo puedes encontrar [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/5%20Fundamentos%20de%20aprendizaje%20por%20transferencia%2F5%20Tensorflow%20HUB%2Fmain.py)

Para este ejemplo vamos a utilizar [mobilnet\_v1\_050\_160](https://tfhub.dev/google/imagenet/mobilenet_v1_050_160/classification/5)

Mobilenets come in various sizes controlled by a multiplier for the depth (number of features) in the convolutional layers. They can also be trained for various sizes of input images to control inference speed. This TF Hub model uses the TF-Slim implementation of mobilenet\_v1\_050 with a depth multiplier of 0.5 and an input size of 160x160 pixels.

The model contains a trained instance of the network, packaged to do the image classification that the network was trained on. If you merely want to transform images into feature vectors, use google/imagenet/mobilenet\_v1\_050\_160/feature\_vector/5 instead, and save the space occupied by the classification layer.

El proceso es sumamente sencillo:

Importamos bibliotecas

from keras.models import Sequential

from keras.layers import Dense, Dropout, InputLayer, Flatten

import tensorflow\_hub as hub

Para este ejemplo seguiremos usando el método get\_data() con su target size de 150x150

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data(target\_size=(150, 150),

color\_mode="rgb")

Creo una arquitectura con base en un modelo pre-entrenado de tensorflow hub:

def hub\_architecture(input\_shape, n\_clases, url\_model):

print("HUB")

model\_hub = Sequential(InputLayer(input\_shape=input\_shape))

# Una secuencia nueva

model\_hub.add(hub.KerasLayer(url\_model, trainable=False))

model\_hub.add(Flatten())

model\_hub.add(Dense(128, activation="relu"))

model\_hub.add(Dropout(0.2))

model\_hub.add(Dense(n\_clases, activation="sotfmax"))

print("BUILD")

# Una secuencia nueva por usar HUB de TensorFlow

model\_hub.build((None, ) + input\_shape)

print(model\_hub.summary())

return model\_hub

La única diferencia ahora es que agregamos una capa al modelo como: hub.KerasLayer ponemos como parámetro el URL del modelo que nos interesa y en esta ocasión ponemos que el modelo NO sea trainable, finalmente, una última diferencia es que debemos hacer un build del modelo antes de continuar con el proceso de compile y fit.

Respuesta esperada:

Model: "sequential\_5"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

keras\_layer\_1 (KerasLayer) (None, 1001) 1343049

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_9 (Flatten) (None, 1001) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_21 (Dense) (None, 128) 128256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_13 (Dropout) (None, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_22 (Dense) (None, 24) 3096

=================================================================

Total params: 1,474,401

Trainable params: 131,352

Non-trainable params: 1,343,049

Vemos como a diferencia de la implementación anterior, al usar TensorFlow HUB todo el modelo se encuentra en una capa enorme llamada keras\_layer\_1

Finalmente, compilamos y entrenamos el modelo:

def fit\_model(classes\_, batch\_size\_, train\_generator\_, validation\_generator\_, test\_generator\_):

url = "https://tfhub.dev/google/imagenet/mobilenet\_v1\_050\_160/classification/4"

h\_model = hub\_architecture(input\_shape=(150, 150, 3), n\_clases=len(classes\_), url\_model=url)

h\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

h\_model.fit(train\_generator\_, epochs=5, validation\_data=validation\_generator\_, batch\_size=batch\_size\_)

h\_model.evaluate(test\_generator\_)

Respuesta esperada:

Epoch 1/5

215/215 [==============================] - 28s 117ms/step - loss: 1.5822 - accuracy: 0.5156 - val\_loss: 0.8558 - val\_accuracy: 0.7228

Epoch 2/5

215/215 [==============================] - 25s 115ms/step - loss: 0.6975 - accuracy: 0.7729 - val\_loss: 0.6246 - val\_accuracy: 0.7747

Epoch 3/5

215/215 [==============================] - 25s 115ms/step - loss: 0.4832 - accuracy: 0.8419 - val\_loss: 0.5053 - val\_accuracy: 0.8168

Epoch 4/5

215/215 [==============================] - 25s 115ms/step - loss: 0.3800 - accuracy: 0.8744 - val\_loss: 0.4488 - val\_accuracy: 0.8337

Epoch 5/5

215/215 [==============================] - 25s 114ms/step - loss: 0.3175 - accuracy: 0.8960 - val\_loss: 0.4326 - val\_accuracy: 0.8344

Precisión en el test set:

57/57 [==============================] - 6s 107ms/step - loss: 0.4522 - accuracy: 0.8296

Este modelo tiene mucho menos ACC que el modelo anterior, sin embargo, veamos que el modelo anterior tenía las siguientes características:

Total params: 13,795,384

Trainable params: 4,820,120

Non-trainable params: 8,975,264

Sin embargo, el nuevo modelo tiene:

Total params: 1,474,401

Trainable params: 131,352

Non-trainable params: 1,343,049

El nuevo modelo es más de 36x más pequeño que el modelo anterior (véase la cantidad de parámetros entrenables)

**6 Resultados de entrenamiento**

A través de los módulos hemos explorado gran parte del ciclo de vida de los algoritmos de deep learning, desde la carga y limpieza de datos hasta la creación y optimización de los mismos, culminando con la implementación de modelos preentrenados desde Keras Applications y TensorHub.

Ahora indagaremos sobre TensorBoard, una herramienta que nos permitirá publicar los resultados de nuestros modelos a la comunidad.

A través de las siguientes sesiones, comprenderemos qué es Tensorboard, sus variables relevantes, cómo analizar y publicar nuestros resultados, además de algunas nociones para escalar nuestros modelos a producción.

* Introducción a variables relevantes de TensorBoard
* Análisis y publicación de resultados del entrenamiento
* Introducción para poner modelos en producción
* Cómo poner en producción

**6.1 Introducción a variables relevantes del TensorBoard**

TensorBoard es una herramienta de visualización para TensorFlow, un popular framework de aprendizaje automático. TensorBoard permite a los desarrolladores visualizar y monitorear el rendimiento de sus modelos de aprendizaje automático de manera interactiva y en tiempo real. Algunas de las características clave de TensorBoard incluyen:

1. Visualización de gráficos de TensorFlow: Con TensorBoard, puedes visualizar los gráficos de TensorFlow para ver cómo se conectan las operaciones y cómo fluyen los datos en el modelo.
2. Visualización de métricas y resúmenes de TensorFlow: TensorBoard permite la visualización de diferentes tipos de métricas, como las pérdidas y las métricas de precisión. También es posible agregar resúmenes personalizados a través de la API de TensorFlow.
3. Monitoreo de entrenamiento: TensorBoard permite la visualización en tiempo real de los indicadores de entrenamiento, como la precisión y la pérdida. Esto permite a los desarrolladores monitorear el progreso del entrenamiento y ajustar los hiperparámetros según sea necesario.
4. Visualización de gráficos de distribución: TensorBoard puede mostrar gráficos de distribución que permiten a los desarrolladores ver cómo se distribuyen los valores de los pesos y los sesgos del modelo.
5. Exploración de datos: TensorBoard permite la visualización de imágenes y otros tipos de datos para comprender mejor cómo se relacionan con el modelo.

Para este ejemplo vamos a ocupar una de nuestras primeras arquitecturas del curso véase: [3.2 Recomendaciones prácticas para ajustar un modelo](https://github.com/ichcanziho/Deep_Learnining_Platzi/tree/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow#32-recomendaciones-pr%C3%A1cticas-para-ajustar-un-modelo) en donde definimos nuestra conv\_architecture de la siguiente manera:

def conv\_architecture(input\_shape, n\_clases):

model = Sequential()

model.add(Conv2D(filters=128, kernel\_size=(3, 3), activation="relu", input\_shape=input\_shape))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(128, activation="relu", kernel\_regularizer=l2(1e-5)))

model.add(Dropout(0.2))

model.add(Dense(n\_clases, activation="softmax"))

print(model.summary())

return model

El código de la clase original está [Aquí](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/3%20Optimizaci%C3%B3n%20del%20modelo%2F2%20CNN%20base%2Fmain.py)

Vamos a modificar ligeramente el código para agregar un callback a nuestro código durante el model.fit

Primero importamos las nuevas bibliotecas que vamos a usar:

from keras.callbacks import TensorBoard

from datetime import date

Ahora generemos el modelo y añadamos el TensorBoard como un callback:

classes, batch\_size, train\_generator, validation\_generator, test\_generator = get\_data()

conv\_model = conv\_architecture(input\_shape=(28, 28, 1), n\_clases=len(classes))

conv\_model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=['accuracy'])

current\_day = date.today().strftime("%dd\_%mm\_%yyyy")

tensorboard\_cb = TensorBoard(log\_dir=f"logs/cnn\_model\_{current\_day}")

history = conv\_model.fit(train\_generator, epochs=25, validation\_data=validation\_generator, batch\_size=128,

callbacks=[tensorboard\_cb])

plot\_results(history, "accuracy", "conv\_results.png")

results = conv\_model.evaluate(test\_generator)

Respuesta esperada:

Found 27455 images belonging to 24 classes.

Found 1425 images belonging to 24 classes.

Found 7172 images belonging to 24 classes.

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d (Conv2D) (None, 26, 26, 128) 1280

max\_pooling2d (MaxPooling2D (None, 13, 13, 128) 0

)

flatten (Flatten) (None, 21632) 0

dense (Dense) (None, 256) 5538048

dropout (Dropout) (None, 256) 0

dense\_1 (Dense) (None, 128) 32896

dropout\_1 (Dropout) (None, 128) 0

dense\_2 (Dense) (None, 24) 3096

=================================================================

Total params: 5,575,320

Trainable params: 5,575,320

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

None

Epoch 1/20

215/215 [==============================] - 7s 21ms/step - loss: 1.5483 - accuracy: 0.5399 - val\_loss: 0.7625 - val\_accuracy: 0.7712

Epoch 2/20

215/215 [==============================] - 2s 10ms/step - loss: 0.3161 - accuracy: 0.9114 - val\_loss: 0.5910 - val\_accuracy: 0.8218

Epoch 3/20

215/215 [==============================] - 2s 10ms/step - loss: 0.1202 - accuracy: 0.9748 - val\_loss: 0.5877 - val\_accuracy: 0.8442

Epoch 4/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0768 - accuracy: 0.9870 - val\_loss: 0.6344 - val\_accuracy: 0.8421

Epoch 5/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0563 - accuracy: 0.9935 - val\_loss: 0.6284 - val\_accuracy: 0.8547

Epoch 6/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0517 - accuracy: 0.9944 - val\_loss: 0.7089 - val\_accuracy: 0.8491

Epoch 7/20

215/215 [==============================] - 2s 9ms/step - loss: 0.0482 - accuracy: 0.9948 - val\_loss: 0.7228 - val\_accuracy: 0.8512

Epoch 8/20

215/215 [==============================] - 2s 9ms/step - loss: 0.0427 - accuracy: 0.9966 - val\_loss: 0.7225 - val\_accuracy: 0.8449

Epoch 9/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0442 - accuracy: 0.9955 - val\_loss: 0.7963 - val\_accuracy: 0.8491

Epoch 10/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0385 - accuracy: 0.9976 - val\_loss: 0.7923 - val\_accuracy: 0.8533

Epoch 11/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0394 - accuracy: 0.9970 - val\_loss: 0.7773 - val\_accuracy: 0.8477

Epoch 12/20

215/215 [==============================] - 2s 11ms/step - loss: 0.0406 - accuracy: 0.9964 - val\_loss: 0.7820 - val\_accuracy: 0.8604

Epoch 13/20

215/215 [==============================] - 2s 12ms/step - loss: 0.0402 - accuracy: 0.9966 - val\_loss: 0.9315 - val\_accuracy: 0.8428

Epoch 14/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0417 - accuracy: 0.9960 - val\_loss: 0.8638 - val\_accuracy: 0.8407

Epoch 15/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0379 - accuracy: 0.9974 - val\_loss: 0.7973 - val\_accuracy: 0.8484

Epoch 16/20

215/215 [==============================] - 2s 11ms/step - loss: 0.0390 - accuracy: 0.9965 - val\_loss: 0.7694 - val\_accuracy: 0.8456

Epoch 17/20

215/215 [==============================] - 2s 11ms/step - loss: 0.0391 - accuracy: 0.9968 - val\_loss: 0.7550 - val\_accuracy: 0.8456

Epoch 18/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0369 - accuracy: 0.9975 - val\_loss: 0.8152 - val\_accuracy: 0.8561

Epoch 19/20

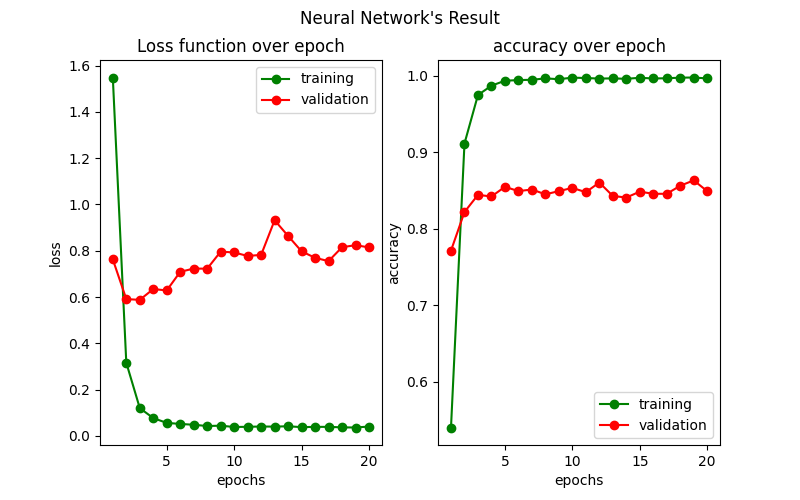
215/215 [==============================] - 2s 10ms/step - loss: 0.0363 - accuracy: 0.9977 - val\_loss: 0.8239 - val\_accuracy: 0.8632

Epoch 20/20

215/215 [==============================] - 2s 10ms/step - loss: 0.0405 - accuracy: 0.9965 - val\_loss: 0.8144 - val\_accuracy: 0.8491

57/57 [==============================] - 1s 18ms/step - loss: 0.7509 - accuracy: 0.8493

Process finished with exit code 0

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/6%20Resultados%20de%20entrenamiento%2F1%20Tensorboard%2Fconv_results.png)

Y podemos ver como genero la siguiente estructura de carpetas:

logs/

----cnn\_model\_24\_03\_2023/

----/-------------------/train

----/-------------------/---------/[events.out.tfevents.1679707300.ichcanziho.39824.0.v2](6%20Resultados%20de%20entrenamiento%2F1%20Tensorboard%2Flogs%2Fcnn\_model\_24\_03\_2023%2Ftrain%2Fevents.out.tfevents.1679707300.ichcanziho.39824.0.v2)

----/-------------------/validation

----/-------------------/---------/[events.out.tfevents.1679707307.ichcanziho.39824.1.v2](6%20Resultados%20de%20entrenamiento%2F1%20Tensorboard%2Flogs%2Fcnn\_model\_24\_03\_2023%2Fvalidation%2Fevents.out.tfevents.1679707307.ichcanziho.39824.1.v2)

En la siguiente clase vamos a ver como abrir estos logs y exportarlos a Tensorflow.dev

**6.2 Análisis y publicación de resultados del entrenamiento**

Conoce más información sobre TensorBoard.dev en <https://tensorboard.dev/> adicionalmente puedes ver otros ejemplos en: <https://www.tensorflow.org/tensorboard/get_started?hl=es-419>

Estando en la carpeta dónde se genero el folder de logs vamos a una terminal y escribimos:

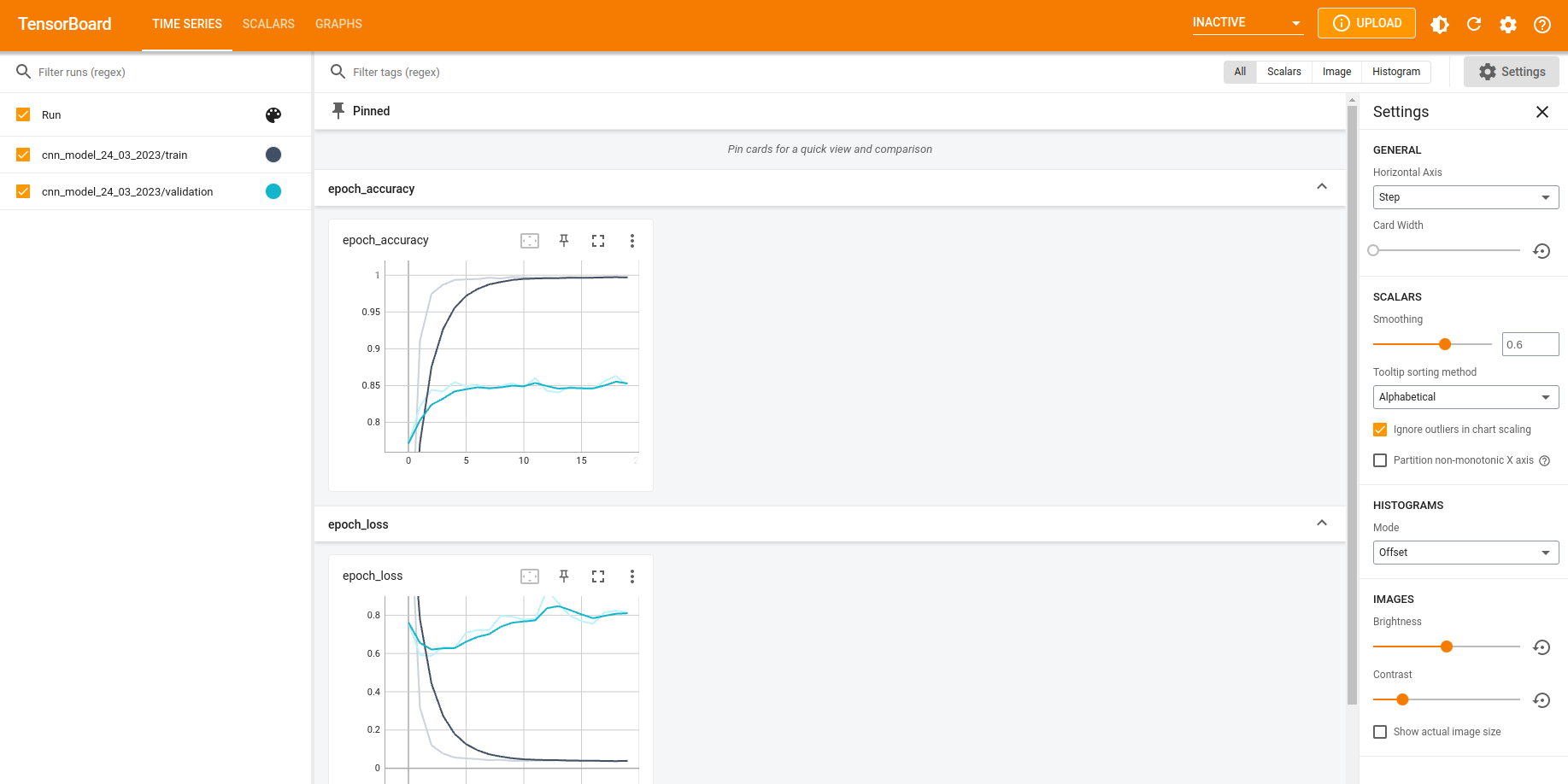
tensorboard --logdir logs

Lo cual me dara un servidor en localhost:6006

Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind\_all

TensorBoard 2.11.2 at http://localhost:6006/ (Press CTRL+C to quit)

Si accedo a dicha dirección en mi navegador:

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F6.png)

Tengo acceso a los resultados de TensorBoard

Y finalmente puedo compartir mis resultados con la comunidad de la siguiente manera:

tensorboard dev upload \

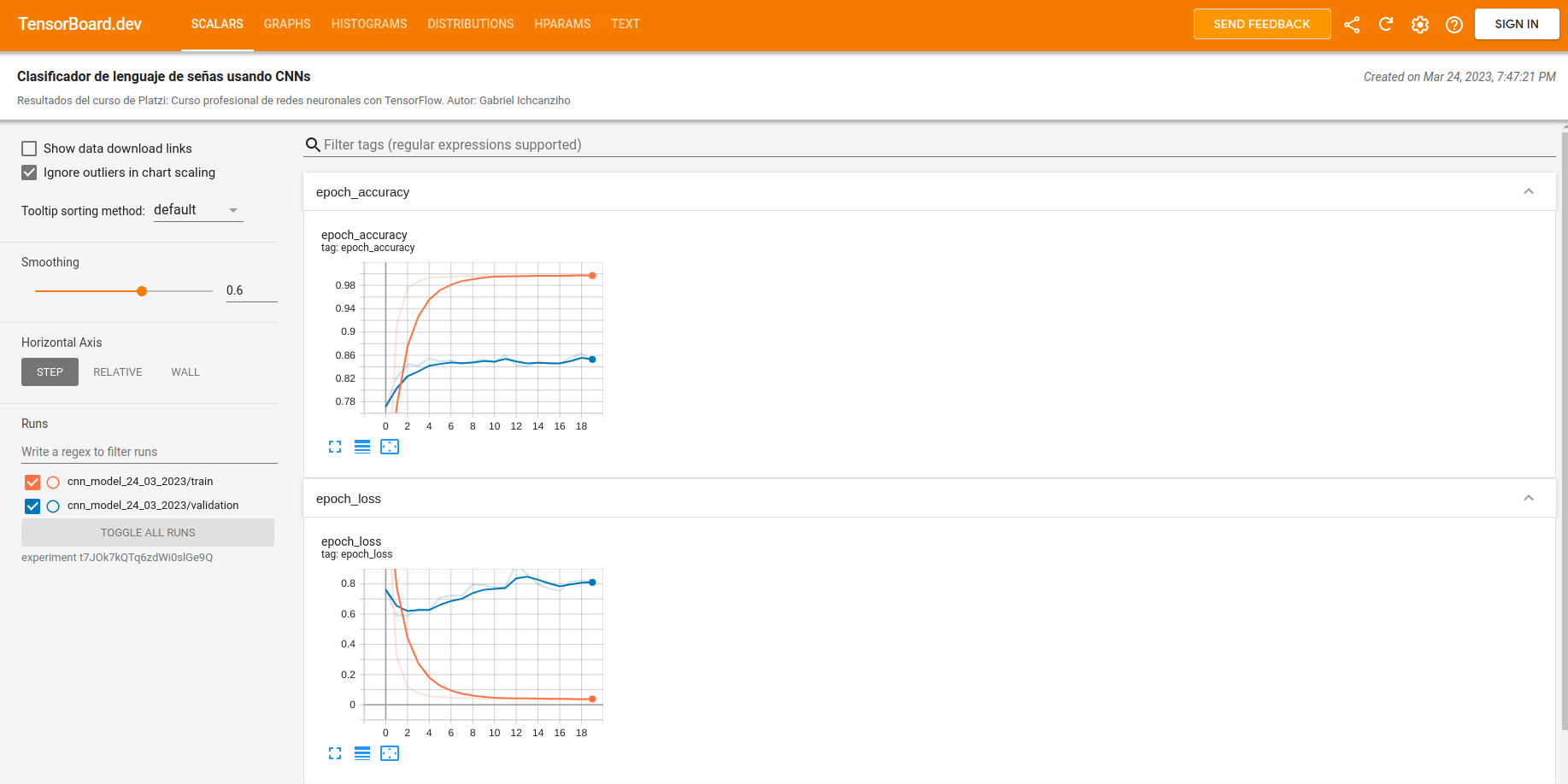
--logdir logs \

--name "Clasificador de lenguaje de señas usando CNNs" \

--description "Resultados del curso de Platzi: Curso profesional de redes neuronales con TensorFlow. Autor: Gabriel Ichcanziho" \

--one\_shot

Ahora puedes acceder a mis resultados en: <https://tensorboard.dev/experiment/t7JOk7kQTq6zdWi0slGe9Q/#scalars>

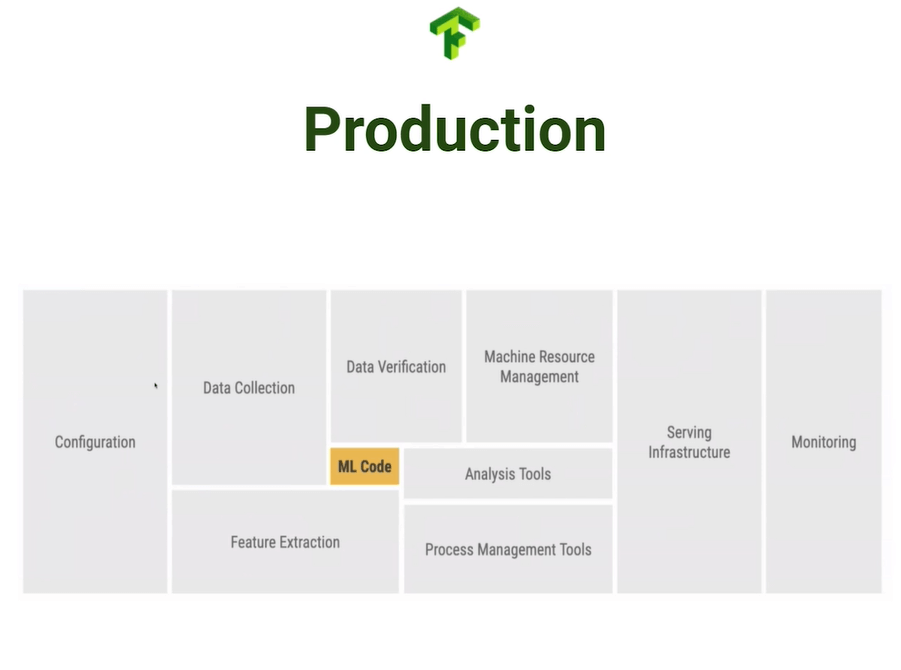
[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F7.png)

Y ese link está disponible para cualquier persona. TensorBoard.dev es una excelente forma de compartir tus resultados con la comunidad.

**6.3 Introducción al despliegue de modelos en producción**

La generación de código para Machine Learning es una parte vasta y en la que se puede profundizar increíblemente, sin embargo, en el gran esquema de las cosas implica una pequeña parte del ciclo de vida entero de un proyecto.

En las siguientes entregas de esta saga se interiorizará sobre el resto de etapas, donde aprenderás a profesionalizarlas.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F8.png)

**Ejemplos de producción**

Puedes desplegar tus modelos en diferentes dispositivos según tu necesidad.

Si tu proyecto va a ser de consumo masificado, entonces la opción natural será desplegarlo en la nube, donde Google Cloud, Azure, AWS u Oracle Cloud podrán ayudarte. Esta ventaja es especialmente útil si debes escalar tu modelo a mayores capacidades sin necesidad de adquirir un equipo propio.

Si necesitas hacer inferencias en vivo entonces podrías optar por equipo IoT, donde dispositivos como la Raspberry Pi o el Jatson Nanon te ofrecerán una capacidad de cómputo decente para tareas en tiempo real.

Si tienes los recursos necesarios o el proyecto no es tan robusto, puedes correr tus modelos de manera local, donde tus equipos se encargarán de las inferencias.

Un caso final (y una extensión a los últimos 2 casos) sería el de usar un USB Accelerator, hardware con alta capacidad de cómputo que procesa las inferencias con alta facilidad.

Puedes concentrar los recursos de predicción sobre este hardware y dejar descansar al resto del equipo.

**6.4 Siguientes pasos con deep learning**

Muchas Felicidades si has llegado hasta aquí has terminado exitosamente el curso de [Curso Profesional de Redes Neuronales con TensorFlow](https://platzi.com/cursos/redes-neuronales-tensorflow/) El curso es impartido por el profesor [Adonaí Vera](https://platzi.com/profes/adonai-vera/) Y este repositorio me pertenece a mí: [Gabriel Ichcanziho](https://www.linkedin.com/in/ichcanziho/).

A través de los módulos profesionalizamos el ciclo de vida de la creación del código de un proyecto de Machine Learning. A través de las sesiones aprendimos:

* Carga de bases de datos en múltiples formatos.
* Generación de modelos de Deep Learning.
* Optimizadores y regularizadores.
* Callbacks personalizados e inteligentes.
* Buscadores inteligentes de hiperparámetros (Keras tuner).
* Uso de redes pre-entrenadas (transfer learning).
* Carga y descarga de configuraciones.
* Análisis gráficos con TensorBoard.
* Introducción y tips en la puestra a producción.

El aprendizaje nunca para, por lo que puedes indagar sobre redes neuronales convolucionales, data augmentation, formatos TF Records, despliegue en producción y Computer Vision.

[](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow/imgs%2F5%2F9.png)

Ahora acompañame al siguiente curso de: [Curso de Transfer Learning con Hugging Face](https://github.com/ichcanziho/Deep_Learnining_Platzi/blob/master/3%20Curso%20profesional%20de%20Redes%20Neuronales%20con%20TensorFlow))