Sesión 4

Orientación a Objetos Avanzada

Overview

Dentro de Python **todo es un objeto**. Funciones, clases, cadenas e incluso tipos son objetos en Python.

Un objeto tiene dos

Pese a ello, Python no obliga a aplicar el paradigma de orientación a objetos, se puede usar como un sencillo script o junto a toros paradigmas.

¿Qué problemas pueden surgir con la Orientación a Objetos?.

Definir clases pueden ayudar a la estructura lógica, pero el estado creado en las propiedades de la clase puede modificar a lo largo de su ciclo de vida. Esto puede llevar a **condiciones de carrera o problemas de concurrencia.**

Por ejemplo, en un programa web, si una petición pide que un objeto cambie el estado, pero al mismo tiempo entra otra petición pidiendo la eliminación de ese objeto, y éste se ejecuta primero, tendremos un objeto "eliminado" con el estado cambiado.

Es por ello que es recomendable usar **funciones sin estado** o al menos funciones nucleares que modifiquen lo menos posible el estado de un objeto.

@Property

En Python `property()` es una función que crea un objeto **property**. El formato de esta función es:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Siendo cada argumento:

- `fget` es una función para acceder al valor del atributo.
- `fset` es una función para insertar un valor al atributo.
- `fdel` es una función para eliminar el atributo.
- `doc` es un docstring

Y un objeto tiene tres métodos, `getter()`, `setter()` y `deleter()`.

Es por ello que se puede montar el objeto como una construcción con decoradores.

```
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value

water = Celsius(10)

print(water.temperature)

# freeze the watter
water.temperature = -20

print(water.temperature)
```

@Classmethod

En Python `classmethod()` devuelve un método de clase, puede ser útil para crear métodos estáticos dentro de una clase.

Normalmente, en cualquier método de clase el primer argumento es `self`, que es la instancia del objeto. En `@classmethod` el primer argumento es la clase sin instanciar. Así, una clase norma podría ser:

```
class Student(object):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

scott = Student('Scott',  'Robinson')
```

Pero usando `@classmethod` podríamos crear algo parecido a esto:

```
class Student(object):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @classmethod
    def from_string(cls, name_str):
        first_name, last_name = map(str, name_str.split(' '))
        student = cls(first_name, last_name)
        return student

scott = Student.from_string('Scott Robinson')
print(scott.first_name)
```

Esto es muy útil para tener métodos auxiliares para crear objetos de alguna fuente como json o serializado:

```
class Student(object):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @classmethod
    def from_string(cls, name_str):
        first_name, last_name = map(str, name_str.split(' '))
        student = cls(first_name, last_name)
        return student

    @classmethod
    def from_json(cls, json_obj):
        # parse json...
        return student

    @classmethod
    def from_pickle(cls, pickle_file):
        # load pickle file...
        return student
```

@Staticmethod

`@staticmethod` es similiar a `@classmethod` pero sin pasar un primer argumento obligatorio,

```
class Student(object):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

scott = Student('Scott',  'Robinson')
```

Pero usando `@classmethod` podríamos crear algo parecido a esto:

```
class Student(object):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @classmethod
    def from_string(cls, name_str):
        first_name, last_name = map(str, name_str.split(' '))
        student = cls(first_name, last_name)
        return student

scott = Student.from_string('Scott Robinson')
print(scott.first_name)
```

Esto es muy útil para tener métodos auxiliares para crear objetos de alguna fuente como json o serializado:

```
class Student(object):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @classmethod
    def from_string(cls, name_str):
        first_name, last_name = map(str, name_str.split(' '))
        student = cls(first_name, last_name)
        return student

    @classmethod
    def from_json(cls, json_obj):
        # parse json...
        return student

    @classmethod
    def from_pickle(cls, pickle_file):
        # load pickle file...
        return student
```

Excepciones

Python provee un mecanismo muy potente de control de errores. Saber utilizar esta herramienta es muy importante ya que protege el código de errores inesperados y permite anteponerse y ejecutar diferentes lineas de código dependiendo de si nuestro programa rompe su ejecución de una manera determinada.

Los tipos de excepciones que podemos encontrar son enormes y están recogidos en la documentación oficial de Python.

La sintaxis para el tratamiento de errores es la siguiente:

```
try:
    Aquí va el código que queremos controlar, siempre identado...
    .......
    .......
except Exception1 as e:
    Aquí tratar la excepción 1 que hemos renombrado a "e"
except Exception2:
    Puede ocurrir otro tipo de excepción
.........
.........
else:
    Este bloque se ejecuta si no ha habido errores.
```

Hay varios puntos a tratar sobre las excepciones:

- Un solo bloque **try** puede tener múltiples excepciones. Esto es muy útil cuando estamos tratando un fragmento de código que puede lanzar distintas excepciones.
- También puede haber un **except** genérico que recoja cualquier excepción.
- Después de una clausula **except** se puede añadir un bloque **else** para ejcutar código.

```
dict_test = {"hello": "hello", "world": "world"}

try:
  test = dict_test["other"]
except KeyError as e:
  print(e)
  test = "Not found"

print(f"We reach this part of the program with {test}")
```

Type Annotations

Pese a que Python es un lenguaje debilmente tipado (no es necesario declarar los tipos al inicializar una variable ya que el interprete los infiere) hemos visto que la omisión de estos puede crear algunas desventajas como:

- Necesidad de añadir más documentación para añadir el contexto de los tipos en funciones y clases
- Empeorar legibilidad del código al añadir ambigüedad al flujo de ejecución
- Aumenta la posibilidad de errores en tiempo en ejecución al no capturar errores de tipo en la fase de desarrollo/compilación.

Es por ello que desde hace unos años, gracias a la especificación PEP 484 introducida en Python 3.5. contamos con la funcionalidad de **Type Hinting**.

Gracias a esta funcionalidad, podemos anotar el tipo de parámetros, variables, tipo de retorno, atributos y métodos...

```
def greet(name: str) -> str:
    return "Hello, " + name
```

La palabra reservada `str` indica que el parametro de entrada y el retorno de la función tienen que ser de tipo `string`.

```
def padding(text: str, number: int = 0) -> str:
  return f"{text.title()}\n{' ' * number}"

padding("hello world", 10)
```

Para esta funcionalidad, **PEP 8** recomienda:

- Usar las mismas reglas para los dos puntos (:), es decir que no haya espacios antes y un solo espacio después de los dos puntos. `text: str`
- Añadir espacios alrededor del signo igual (=) cuando se combina el tipo con un valor por defecto. `default: bool = True`
- Añadir espacios alrededor de la flecha (->). `def padding(...) -> str`

Use normal rules for colons, that is, no space before and one space after a colon (text: str). Use spaces around the = sign when combining an argument annotation with a default value (align: bool = True). Use spaces around the -> arrow (def headline(...) -> str).

Visualización de datos con matplotlib

Matplotlib es una biblioteca de visualización en Python que produce figuras y gráficos de alta calidad en una variedad de formatos estáticos, animados e interactivos. Es especialmente útil

para aquellos que trabajan en ML, ya que permite visualizar datos y resultados de modelos de una manera comprensible y atractiva.

Su uso es bastante sencillo, simplemente vamos a manejar gráficas, por lo tanto vamos a tener que pensar en el eje x y el eje y para luego crear los tipos de gráfica

```
# Gráfico de lineas

import matplotlib.pyplot as plt
import numpy as np

# Datos
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Crear el gráfico
plt.plot(x, y)

# Título y etiquetas
plt.title('Gráfico de Líneas')
plt.xlabel('X')
plt.ylabel('sin(X)')

# Mostrar el gráfico
plt.show()
```

```
# Histograma

import matplotlib.pyplot as plt
import numpy as np

# Datos: 1000 números aleatorios de una distribución normal
datos = np.random.randn(1000)

# Crear el histograma
plt.hist(datos, bins=50)

# Título y etiquetas
plt.title('Histograma')
plt.xlabel('Valor')
plt.ylabel('Frecuencia')

# Mostrar el gráfico
plt.show()
```

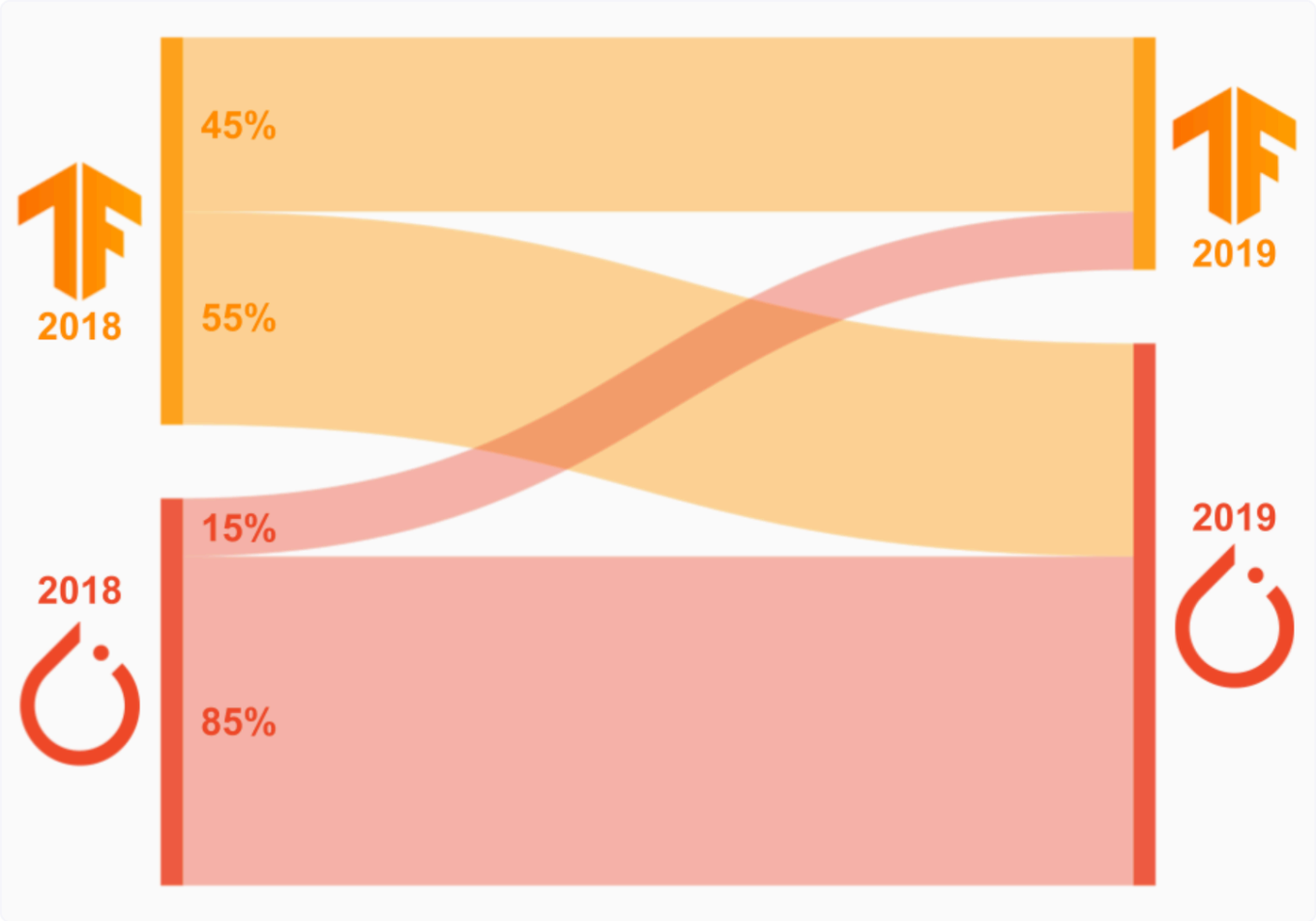
```
# Gráfico de dispersión

import matplotlib.pyplot as plt
import numpy as np

# Datos
x = np.random.randn(100)
y = x * 3 + np.random.randn(100) * 2

# Crear el gráfico de dispersión
plt.scatter(x, y)

# Título y etiquetas
plt.title('Gráfico de Dispersión')
plt.xlabel('X')
plt.ylabel('Y')

# Mostrar el gráfico
plt.show()
```

Pytorch vs Tensorflow

Actualmente, el panorama de desarrollo de modelos IA tiene enfrentado a estos dos frameworks, los puntos más relevantes del estado de los modelos es:

1. **Disponibilidad de Modelos**:
    - PyTorch tiene una ventaja significativa en la disponibilidad de modelos pre-entrenados, especialmente en la plataforma HuggingFace, donde casi el 92% de los modelos son exclusivos de PyTorch.
    - En la comunidad de investigación, PyTorch también domina con una adopción que creció rápidamente en los últimos años, siendo utilizado en casi el 80% de los papers que comparaban ambos frameworks.
2. **Infraestructura de Despliegue**:
    - La infraestructura de despliegue es crucial para poner en práctica los modelos entrenados, destacando la importancia de un despliegue eficiente, en este terreno tensorflow lleva la ventaja con tsx, aunque PyTorch está presentando actualmente novedades en este campo.
3. **Ecosistemas e integración**:
    - Un framework que se integre bien en diversos ecosistemas y hardware especializado es esencial para facilitar el desarrollo en aplicaciones móviles, locales y de servidor, Tensorflow tiene mucha facilidad de ejecutarse en casi cualquier entorno sin apenas configuración, PyTorch tiene un modelo más manual, pero a veces más efectivo.

A pesar de que TensorFlow solucionó varios problemas con la versión 2 y la inclusión de Keras como API principal, PyTorch sigue siendo el preferido en la comunidad de investigación. La elección entre estos frameworks dependerá de las necesidades específicas de los proyectos y las preferencias de los desarrolladores. Y así ha sido desde 2019.

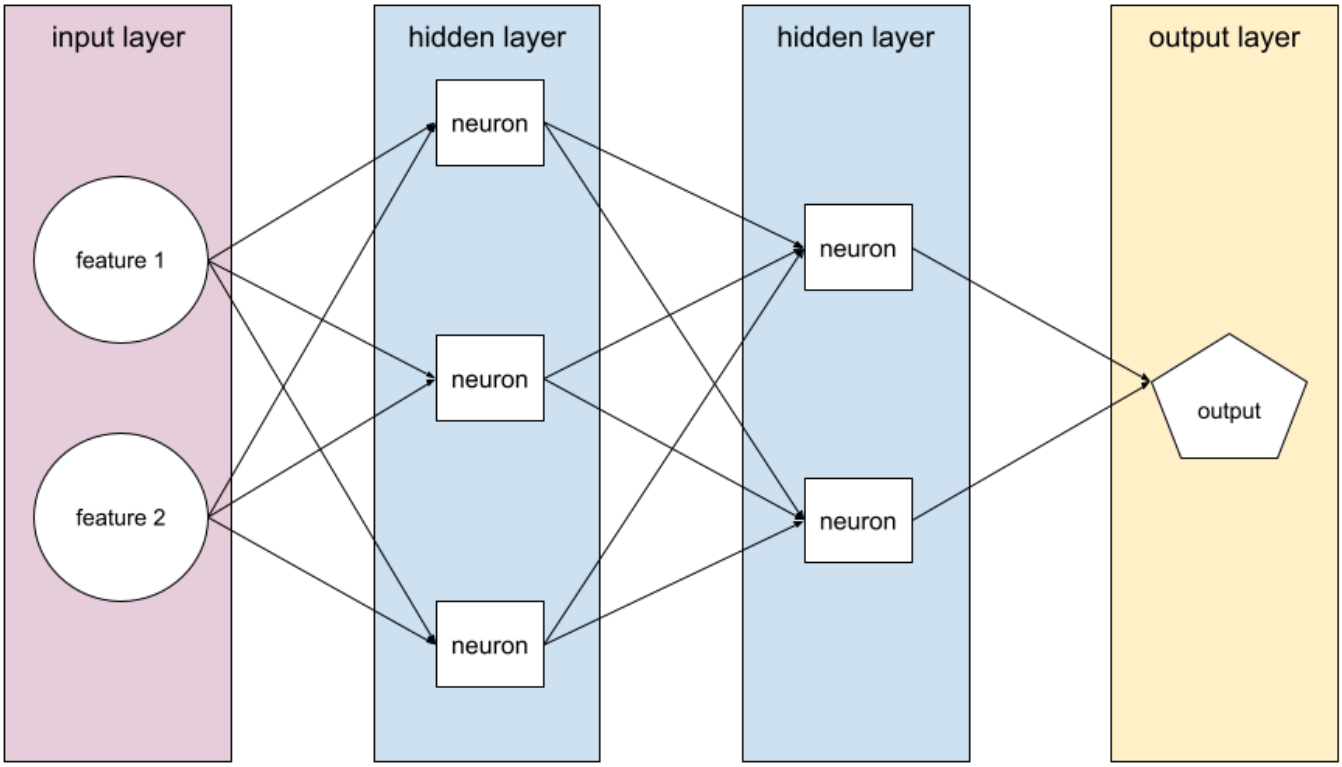


Actualmente se está desarrollando Keras 3, que solventaría parte de estos problemas al poder ejecutarse en TF, Pytorch y JAX.

Modelo de Red Neuronal en Keras

Introducción

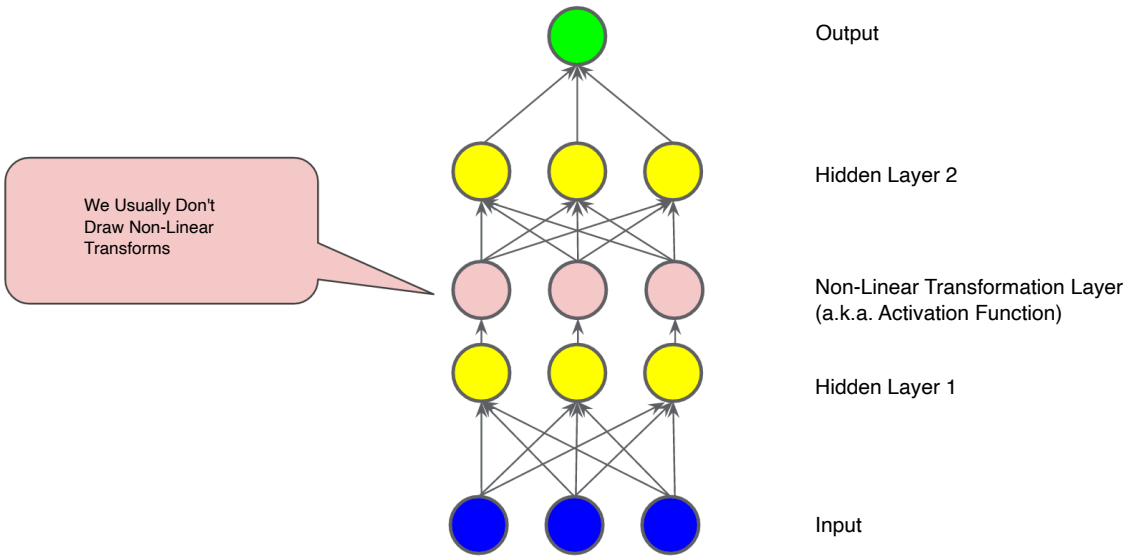
Las redes neuronales son modelos que se caracterizan por tener una serie de entradas, una serie de salidas y una o varias capas ocultas que continen **neurones**.



Cada **neurón** conecta con todos los nodos de la siguiente capa y contienen una **función de activación** para introducir una no linearidad entre las transformaciones de las capas.

Se usa por ejemplo en problemas de clasificación **no linear** es decir, que no se puede poner una linea para una clasificación o en modelos de predicción **no lineares**, justo lo opuesto al ejemplo que hemos visto en la sección anterior.

Para introducir activaciones **no lienares** entre capas, se utiliza una **función de activación**, que cambiará el valor de cada nodo antes de pasarse a la suma de las siguientes funciones.



Algunas de las funciones de activación más comunes son:

Sigmoide

La función de activación **sigmoide** convierte la suma ponderada entre un valor de 0 y 1

$$F(x) = \frac{1}{1 + e^{-x}}$$

```
import matplotlib.pyplot as plt
import numpy as np
import math

x = np.linspace(-10, 10, 100)
z = 1/(1 + np.exp(-x))

plt.plot(x, z)
plt.xlabel("x")
plt.ylabel("Sigmoid(X)")

plt.show()
```

ReLU

La función de activación de **unidad lineal rectificada** a menudo funciona mejor que una función continua como la sigmoide y es mucho más fácil de calcular

$$F(x) = max(0, x)$$

```
import matplotlib.pyplot as plt
import numpy as np
import math

x = np.linspace(-10, 10, 100)
z = np.maximum(0, x)

plt.plot(x, z)
plt.xlabel("x")
plt.ylabel("ReLU(X)")

plt.show()
```

MNIST

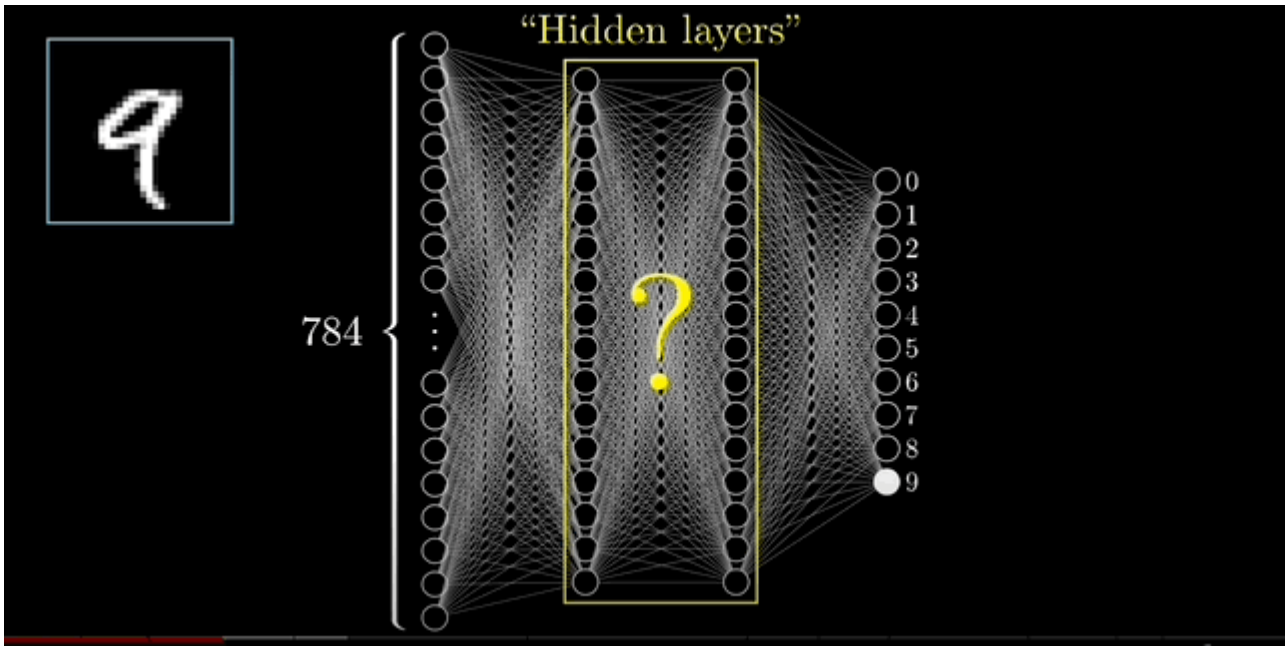
El dataset de dígitos de MNIST (Mixed National Institute of Standards and Technology) es uno de los datasets más usados dentro de la investigación y la Visión Artificial, y ha tenido un rol muy importante dentro del desarrollo de Redes Neuronales pretenecientes al *depp learning*.

Algunas de las figuras que continen el dataset serían las siguientes.

El el siguiente cuadro de código veremos un clasificador muy simple llamado softmax regression. No entraremos en profundidad en la parte matemática ya que nos interesa más bien la estructura de TensorFlow.

Su funcionamiento consiste en que el modelo intentará de resolver, por cada pixel, que digitos tienen valores altos y bajos en cada espacio. Es decir, en los píxeles centrales es muy probable que sean negros para los 0 mientras que sean blancos para los 1.

El objetivo de este modelo consiste en encontrar pesos que nos asegure la evidencia de la existencia de cada uno de los dígitos, sin usar la información espacial del pixel.



Explicación de 3Blue1Brown

Es por ello que para el siguiente ejemplo usaremos Keras para:

1. Construir una red neuronal que clasifique imágenes.
2. Añadir capas ocultas para su entrenamiento.
3. Entrenar esa red neuronal.
4. Evaluar la precisión del modelo

Importar dependencias

Lo primero sería importar tensorflow y preparar el dataset de MNIST, para ello vamos a acceder al módulo `mnist`. Tensorflow tiene una gran variedad de datos de catalogo para entrenar nuestros modelos, puedes consultarlos aquí.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
```

Establecer los datos de entrenamiento

Al igual que en el ejemplo anterior el siguiente paso será establecer los datos para entrenamiento y test, en este caso convertiremos los datos de enteros a numeros con coma flotante:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
print(x_train[:1])
```

Crear el modelo

Ahora constuiremos el modelo con la API `tf.keras.Sequiential`, como podemos observar, podemos declarar las capas en variables o directamente instanciarlas en la propia declaración del modelo. Elegiremos un optimizador y una función de perdida.

```
l0 = tf.keras.layers.Flatten(input_shape=(28, 28))

model = tf.keras.models.Sequential([
  l0,
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

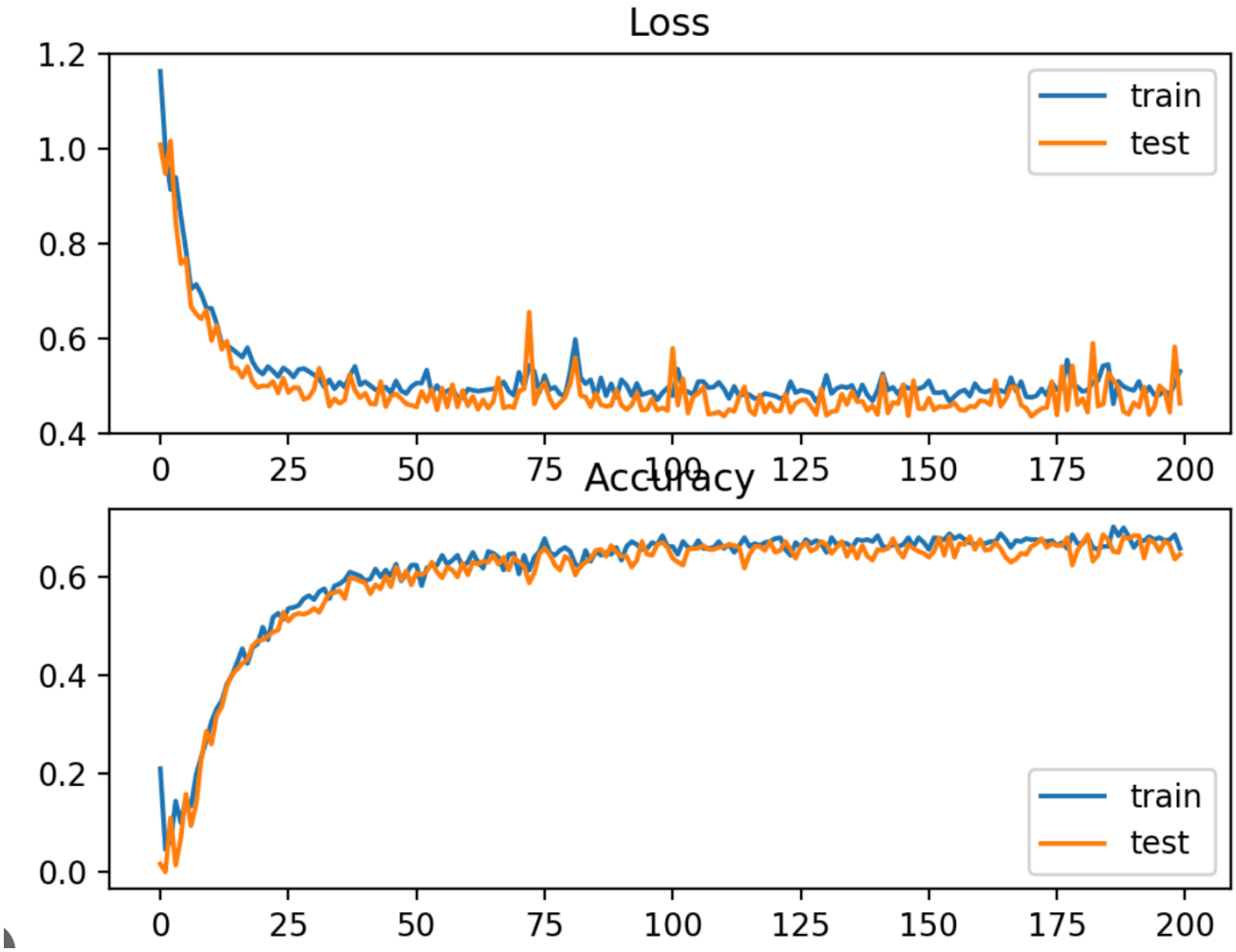
Vamos a ver la información de las capas antes del entrenamiento:

```
for index, layer in enumerate(model.layers):
  print(f"--------------{index}--------------")
  print(layer.name)
  print(layer.weights)
  print(layer.trainable)
  print(f"-----------------------------")
```

También se puede ver un resumen del modelo de forma sencilla con este método

```
model.summary()
```

Función de perdida



La [función de perdida](#) `losses.SparseCategoricalCrossentropy` coge un vector de logits y un Inidce `True` y devuelve una perdida escalar por cada ejemplo.

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

Precisión del modelo sin entrenar

Como curiosidad, vamosa a ver qué es lo que devuelve el modelo (antes de ser entrenado) al pasarle datos de entrenamiento.

Por cada ejemplo, el modelo devuelve un vetor de "logits" o "log-odds", uno para cada clase, que son vectores de predicción del modelo.

```
predictions = model(x_train[:1]).numpy()
predictions
```

La función `tf.nn.softmax` convierte estos logits en probabilidades para cada clase.

Básicamente estas predicciones no normalizadas son convertidas en predicciones reales mediante esta función

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \dots, K$$

```
tf.nn.softmax(predictions).numpy()
```

Ahora como curiosidad, y **antes de entrenar el modelo** vamos a calcular la función de pérdida para saber el **porcentaje de acierto** de nuestro modelo antes de ser entrenado.

Esta perdida es igual a la probabilidad logarítmica negativa de la clase: Es cero si el modelo está seguro de la clase correcta.

El modelo sin entrenar da probabilidades cercanas a algo aleatorio (1/10 por cada clase), así que la perdida inicial suele estar cerca de: `-tf.math.log(1/10) ~= 2.3`.

```
loss_fn(y_train[:1], predictions).numpy()
```

```
-tf.math.log(1/10).numpy()
```

Compilar el modelo

Vamos a configurar el modelo para su entrenamiento, como vimos en la sesión 2 vamos a hacer uso de una **función de optimización** y una **función de pérdida** para entrenar el modelo.

Además, vamos a introducir el concepto de **métricas**, básicamente son funciónes similares a la función de perdida, que permiten evaluar la precisión del modelo, pero a diferencia de la función de perdida, no es usada en el entrenamiento.

```
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

Entrenar el modelo

Vamos a entrenar el modelo con los datos de entrenamiento llamando al método `fit`.

Símplemente vamos a coger como parámetros las **features** (`x_train`) las **labels** (`y_train`), los ciclos de cada entrenamiento (`epochs`) y con ello nos va a devolver un objeto `history` con el que medir el entrenamiento.

```
history = model.fit(x_train, y_train, epochs=5)
```

Así, los pesos de las capas quedarían así:

```
for index, layer in enumerate(model.layers):
  print(f"--------------{index}--------------")
  print(layer.name)
  print(layer.weights)
  print(layer.trainable)
  print(f"-----------------------------")
```

Mostrar las estadísticas del entrenamiento.

El método `fit` devuelve un objeto `history`. Podemos usar este objeto para mostrar una gráfica con la variación de la **función de perdida** entre cada *epoch*. Una perdida grande significa que la predicción está muy lejos del resultado.

Vamos a usar Matplotlib para visualizar el resultado. Como podemos ver, el modelo mejora rápidamente para luego frenarse al final de las ejecuciones.

```
import matplotlib.pyplot as plt
plt.xlabel('Epoch Number')
plt.ylabel("Loss Magnitude")
plt.plot(history.history['loss'])
```

Precisión del modelo entrenado

Siguiendo los pasos que hemos hecho en la sección de la precisión sin entrenar, vamos a comprobar que las predicciones son más precisas con el modelo entrenado.

```
predictions_trained = model(x_test[:1]).numpy()
predictions_trained
```

La función `tf.nn.softmax` convierte estos logits en probabilidades para cada clase.

```
tf.nn.softmax(predictions_trained).numpy()
```

Ahora como curiosidad, y **antes de entrenar el modelo** vamos a calcular la función de pérdida para saber el **porcentaje de acierto** de nuestro modelo antes de ser entrenado.

La función de perdida `losses.SparseCategoricalCrossentropy` coge un vector de logits y un Inidce `True` y devuelve una perdida escalar por cada ejemplo.

```
loss_fn_trained = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

Como podemos observar, la función de pérdida ahora será muy cercana al cero, al ser unos datos de comprobación, podemos tener confianza de que el modelo es bastante preciso sin entrar en overfitting

```
loss_fn_trained(y_test[:1], predictions_trained).numpy()
```

Mostrar el desempeño del modelo

El método `Model.evaluate` comprueba el desempeño del modelo.

```
model.evaluate(x_test,  y_test, verbose=2)
```

Ahora el modelo está entrenado para tener una probabilidad del 97%

Modelo de Red Neuronal en Pytorch

Vamos a ver los conceptos fundamentales de Pytorch mediante unos ejemplos.

Trabajando con datasets

Pytorch tiene dos primitivas para trabajar con datos: `torch.utils.data.DataLoader` y `torch.utils.data.Dataset`. `Dataset` almacena los ejemplos y las etiquetas y `DataLoader` añade un iterable al `Dataset`.

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
```

PyTorch ofrece bibliotecas específicas para diferentes dominios como TorchText, TorchVision y TorchAudio, todas las cuales incluyen conjuntos de datos. Para este ejmplo, utilizaremos un conjunto de datos de TorchVision.

El módulo `torchvision.datasets` contiene objetos `Dataset` para muchos datos de visión del mundo real como CIFAR, COCO (lista completa aquí). En este ejemplo, usamos el conjunto de datos FashionMNIST. Cada `Dataset` de TorchVision incluye dos argumentos: `transform` y `target_transform` para modificar las muestras y las etiquetas respectivamente.

```
# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

Pasamos el `Dataset` como un argumento a `DataLoader`. Esto envuelve un iterable sobre nuestro conjunto de datos y soporta la agrupación automática, muestreo, mezcla y carga de datos multiproceso. Aquí definimos un tamaño de lote de 64, es decir, cada elemento en el iterable del cargador de datos devolverá un lote de 64 características y etiquetas.

```
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

Creando modelos

Para definir una red neuronal en PyTorch, creamos una clase que hereda de nn.Module. Definimos las capas de la red en la función `__init__` y especificamos cómo los datos pasarán a través de la red en la función `forward`. Para acelerar las operaciones en la red neuronal, la movemos a la GPU o MPS si están disponibles. Espero que te sea útil. Si necesitas más ayuda, no dudes en preguntar.

```
# Get cpu, gpu or mps device for training.
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

Optimizando los parámetros del modelo

Para entrenar el modelo, necesitamos una función de perdida y un optimizador.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

En un solo ciclo de entrenamiento, el modelo hace predicciones en el conjunto de datos de entrenamiento (que se le proporciona en lotes) y retropropaga el error de predicción para ajustar los parámetros del modelo.

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

También comprobamos el rendimiento del modelo con el conjunto de datos de prueba para asegurarnos de que está aprendiendo.

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss
```

El proceso de entrenamiento se lleva a cabo durante varias iteraciones (épocas). Durante cada época, el modelo aprende parámetros para hacer predicciones mejores. Imprimimos la precisión y la pérdida del modelo en cada época; nos gustaría ver que la precisión aumenta y la pérdida disminuye con cada época.

```
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

Huggingface API

Huggingface tiene una librería y una serie de APIs para diferentes acciones dentro del mundo de la IA, hoy veremos dos de las más sencillas y como usarlas en Python. Lo primero será definir qué es una API.

API

API significa Interfaz de Programación de Aplicaciones (Application Programming Interface en inglés). Es un conjunto de reglas y mecanismos que permiten que diferentes software o aplicaciones interactúen entre sí. Una API define los métodos y estructuras de datos que los desarrolladores pueden usar para interactuar con el software componente (ya sea sistema operativo, bibliotecas o diferentes servicios). En el caso de las APIs web, permiten la interacción entre diferentes servicios en la web utilizando generalmente el protocolo HTTP.

Vamos a ver un ejemplo con una de las librerías más famosas, https://pokeapi.co/

```
!pip install requests

import requests

# Definir la URL de la API
url = 'https://pokeapi.co/api/v2/pokemon/1'

# Hacer una petición GET a la API
response = requests.get(url)

# Verificar si la petición fue exitosa (código de estado 200)
if response.status_code == 200:
    # Convertir la respuesta a JSON
    pokemon_data = response.json()
    # Imprimir el nombre del Pokémon
    print(f'Nombre del Pokémon: {pokemon_data["name"]}')
else:
    print(f'Error: No se pudo obtener los datos. Código de estado: {response.stat
```

En este código:

1. Importamos la biblioteca requests.
2. Definimos la URL de la API de PokeAPI que queremos acceder.
3. Hacemos una petición GET a la API usando requests.get(url).
4. Verificamos si la petición fue exitosa comprobando el código de estado de la respuesta.

5. Si la petición fue exitosa, convertimos la respuesta a JSON usando `response.json()` y luego imprimimos el nombre del Pokémon.

AutoClasses

Las AutoClasses son una serie de clases diseñadas para trabajar con diferentes modelos de manera automática. Estas clases seleccionan automáticamente la clase adecuada para el modelo que se está utilizando, lo que facilita trabajar con diferentes modelos sin tener que cambiar el código.

```
!pip install transformers

from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Cargar el tokenizer y el modelo
tokenizer = AutoTokenizer.from_pretrained("nlptown/bert-base-multilingual-uncased
model = AutoModelForSequenceClassification.from_pretrained("nlptown/bert-base-mul

# Tokenizar y clasificar una frase
inputs = tokenizer("I love programming!", return_tensors="pt")
outputs = model(**inputs)

# Obtener la clasificación
print(outputs.logits)
```

En este ejemplo, utilizamos las clases AutoTokenizer y AutoModelForSequenceClassification para cargar un tokenizer y un modelo preentrenados. Luego, tokenizamos una frase y la clasificamos utilizando el modelo.

Pipeline

Pipelines de Hugging Face es una API de alto nivel diseñada para facilitar el uso de modelos preentrenados. Puede utilizarse para diversas tareas como análisis de sentimiento, traducción de lenguaje, generación de texto, entre otros, utilizando solo unas pocas líneas de código.

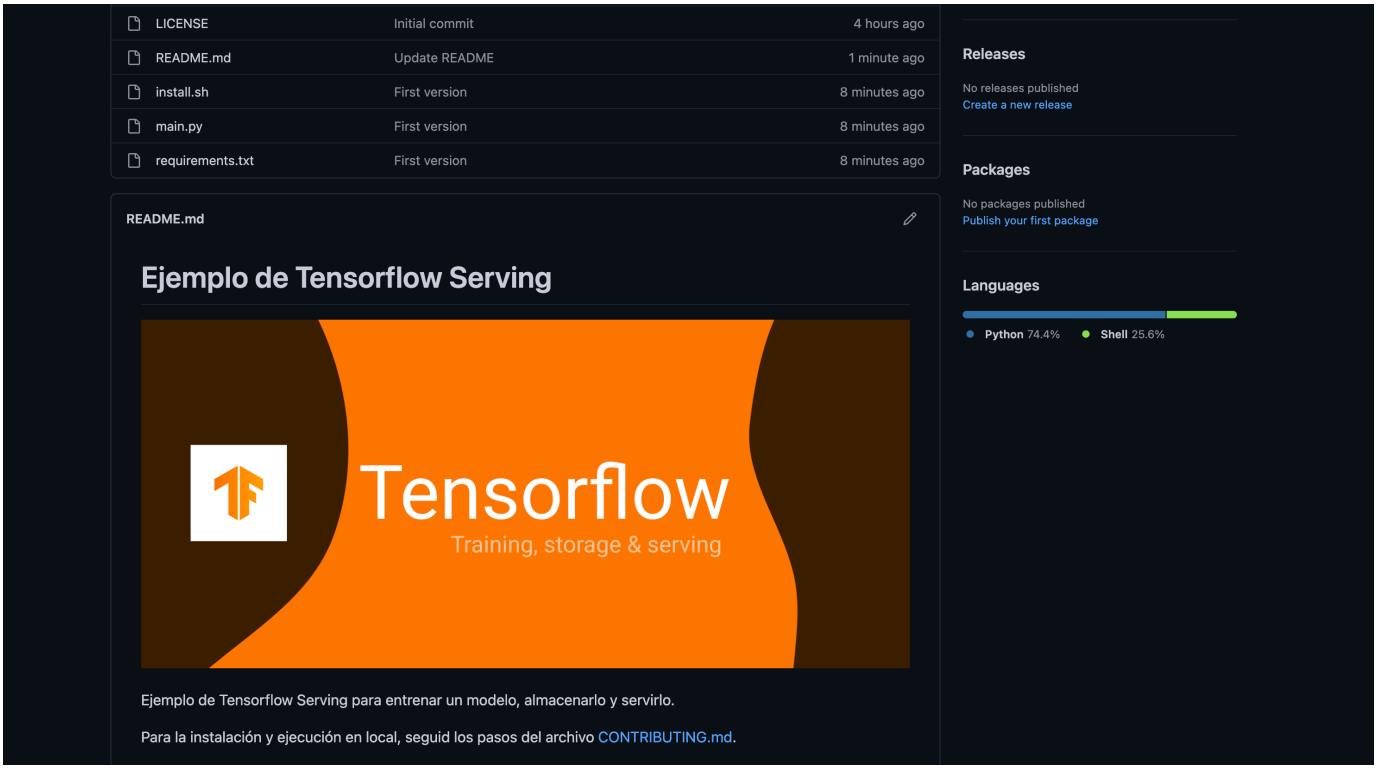
```
# import transformers if not done above
from transformers import pipeline

# Crear un pipeline para análisis de sentimiento
nlp = pipeline("sentiment-analysis")

# Analizar el sentimiento de una frase
result = nlp("I love programming!")
print(result)
```

En este ejemplo, primero importamos la función pipeline de la biblioteca transformers. Luego, creamos un pipeline de análisis de sentimiento y lo utilizamos para analizar el sentimiento de una frase.

Estos son ejemplos básicos para introducir a los estudiantes en el uso de las APIs de Hugging Face. Desde aquí, pueden explorar más funcionalidades y modelos proporcionados por Hugging Face para abordar diferentes tareas de procesamiento del lenguaje natural (NLP) y aprendizaje automático.

Ejecución de un modelo en local

Vamos a ver un ejemplo de un modelo ejecutado en local a través del repositorio [Tensorflow Serving Example]



Puesta en marcha

Este modelo está pensado para ejecutarse en local. Para ello hay que hacer lo siguiente:

1. Clonar el código en tu equipo

```
git clone git@github.com:lucferbux/Tensorflow-Serving-Example.git
```

2. Crear el entrono virtual e instalar dependencias

linux/macos

```
./setup.sh
source venv/bin/activate
```

windows

```
pip3 install virtualenv
virtualenv venv
source venv/bin/activate
pip3 install --no-cache-dir -r ./requirements.txt
```

3. Ejecutar el modelo

```
python3 main.py
```

4. Lanzar Tensorflow Serving

```
docker run -p 8501:8501 \
  --privileged=true --platform linux/amd64 \
  --mount type=bind,source={export_path},target=/models/fashion_model \
  -e MODEL_NAME=fashion_model -t tensorflow/serving
```

5. Lanzar la predicción

Para ver la predicción que lanzamos, podemos ejecutar:

```
python3 model/plot_inference_request.py
```

Esto nos permitirá ver qué estamos mandando como predicción, y que recibirá nuestro modelo al lanzar el curl.

6. Procesar la respuesta

Podemos ver la respuesta de la predicción si ejecutamos:

```
python3 model/process_inference_response.py
```

En el que nos dará el resultado que estamos buscando.

Estructura del repositorio

Es importante mantener una estructura lógica dentro de nuestro proyecto que sea fácil de entender por la comunidad.

Una manera sencilla de organizar nuestro código sería:

```
README.md
LICENSE
requirements.txt
model/__init__.py
model/model_training.py
model/plot_inference_request.py
model/process_inference_response.py
persistance/__init__.py
persistance/persistance.py
docs/CONTRIBUTING.md
setup.sh
```

Módulo

- **Localización ->** `./model/__init__.py` o `model/model_training.py/`
- **Objetivo ->** La lógica de la aplicación.

Vamos a tener la lógica de nuestro código dividido en carpetas para poder realizar separación de funcionalidades. Para importar el código de nuestros módulos vamos a usar **import**.

License

- **Localización ->** `./LICENSE`
- **Objetivo ->** Parte legal

Parte legal del proyecto, determinará los usos que se puedan dar, hay muchas licencias para elegir y repositorios remotos como **Github** o **Gitlab** permiten añadirlo.

Requirements File