

# Auxiliar Deploy DL en Flask CC6409

---

En este auxiliar construiremos una aplicación web mediante Flask, la cual instanciará un modelo preentrenado de Deep Learning y lo usará para dar lugar a predicciones según input del usuario.

- Específicamente, el modelo es una DenseNet121 [\[1608.06993\] Densely Connected Convolutional Networks](#).
- La tarea sobre la cual se entrena es la de, dada una imagen, predecir la clase correspondiente según el conjunto de datos ImageNet 1K, el cual contiene millones de imágenes etiquetadas según 1000 clases distintas.

El proyecto se dividirá en dos servidores de Flask:

- Una aplicación corresponderá al frontend, donde vivirá el formulario para subir una imagen y el espacio para mostrar la clase asociada.
- La otra aplicación corresponderá a una API REST que hará de backend: recibirá solicitudes POST con imágenes y devolverá una respuesta en formato JSON.

Para ello usaremos:

- [Flask](#), como *framework* de desarrollo web.
- [Pytorch](#), como *framework* para el desarrollo de modelos DL
- [GitHub](#), como sistema de versionamiento
  - Necesita registro. Pueden acceder al servicio completo usando el [Student Developer Pack](#)
- [Pycharm](#) como IDE (Integrated Development Environment) de Python.
  - Pueden optar a una [Free Educational License](#) con su correo uchile.
- [Anaconda](#) como administrador de ambientes.

## A. Configuración inicial

---

1. Comenzaremos creando un repositorio en GitHub, el cual usaremos para dejar todos nuestros proyectos del ramo.
2. En paralelo, creamos un nuevo proyecto de Pycharm el cual llamaremos CC6409.

- Location debe terminar en \CC6409
  - Seleccionamos la opción Intérprete de Python, con Conda como ambiente y la versión 3.10 de Python.
3. Le damos a Create. Esto generará un ambiente con un proyecto de Anaconda, lo cual podría demorarse unos 3-5 min.
4. Una vez creado el proyecto, Pycharm nos entrega (entre otras cosas) un terminal. Ingresamos el comando para clonar nuestro repositorio de GitHub:

```
git clone https://github.com/matiasvergaras/CC6409.git
```

5. Esto trae nuestro repositorio al proyecto, que debiese aparecer como una carpeta de la raíz. Entramos a esa carpeta.

```
cd CC6409 # o como se llame su repositorio
```

6. El siguiente paso es instalar las librerías necesarias. Para ello, corremos los siguientes comandos en el terminal.
- Nótese que el comando para instalar Pytorch varía según el sistema operativo, revisar [sitio de instalación](#).
  - Si se nos pregunta por librerías adicionales, aprobar su instalación ( 'y' + Enter )

```
conda install flask  
conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```

7. Teniendo el repositorio y el ambiente listo, procedemos a inicializar el proyecto de Flask. Para ello ejecutamos los siguientes comandos:

```
# Dentro del repositorio, es decir, en CC6409/CC6409  
mkdir classifier-app # aquí vivirá nuestro proyecto frontend  
mkdir pytorch-api    # aquí vivirá la API REST
```

- A diferencia de otros *frameworks* de desarrollo web como Django, Flask es bastante minimalista: no tenemos un comando `start` que nos arme todo sino que, por el contrario, lo vamos armando a medida que lo necesitamos. Esto puede parecer extraño

a primera vista, pero a la larga resulta mucho más cómodo que tener 1234 módulos y funciones precargadas de las cuales solo se usa un tercio.

8. Hasta aquí ya tenemos gran parte del trabajo hecho, así que guardémoslo - o mejor dicho, 'versionémoslo') - :

```
# En CC6409/CC6409
git add .
git commit -m"Configuración inicial"
git pull
git push
```

- El comando `git add` "prepara" los cambios para ser enviados al servidor de GitHub. Recibe como parámetro un archivo o un directorio y lo agrega de forma recursiva (nótese que en este caso usamos `.` para indicar el directorio actual, que contiene todo el proyecto).
- El comando `git commit` hace un "checkpoint" de lo que hemos preparado hasta ahora, y lo guarda con el mensaje que le demos a través de la *flag* `-m"mensaje"`.
- El comando `git pull` trae los cambios que hayan en el repositorio a nuestro equipo. Trabajando de forma individual y en un solo computador no suele ser muy útil, pero siempre es bueno acostumbrarse a usarlo antes de hacer push (cuando no se ocupa en trabajos grupales, suele dar lugar a *conflictos*).
- El comando `git push`, por último, envía nuestro último checkpoint al servidor de GitHub, asegurándonos que todo lo que hicimos quede respaldado.

## B. Creando la API REST que contendrá el modelo

Para esta parte necesitaremos el archivo `imagenet_class_index.json`, disponible en Material Docente bajo el nombre de `Deploy en Flask` (junto a otros scripts extensos que iremos usando).

Nuestra primera tarea será crear la API REST que contendrá al modelo. En específico, las tareas de esta API serán:

- Instanciar el modelo. Nótese que nos interesará hacer este paso una sola vez (cuando levantemos la API), y no una vez por cada solicitud (sería muy ineficiente).
- Recibir imágenes por solicitudes POST,
- Pasar las imágenes por el modelo (con los ajustes que sean necesarios),

- Devolver un JSON con la clase predicha.
1. Para ello, trabajaremos en la carpeta `pytorch-api` y comenzaremos creando algunos scripts:
    - `app.py` , donde dejaremos las variables globales y todo lo relacionado a configuración de la aplicación
    - `main.py` , que será la puerta de entrada a nuestra API y contendrá tanto las rutas de la aplicación.
    - `utils.py` , donde dejaremos funciones útiles que nos servirán para procesar las imágenes y generar predicciones.
    - Movemos aquí también el archivo `imagenet_class_index.json` .

Una gracia de Flask es que todos los `.py` podrían vivir en un único módulo, sin embargo, separar las funcionalidades más importantes permite mantener el orden en proyectos grandes.

2. Luego comenzaremos a poblar los archivos que acabamos de crear. Comenzamos con `app.py` :

```
# en CC6409/CC6409/pytorch-api/app.py
import json
from torchvision import models
from flask import Flask

app = Flask(__name__)
imagenet_class_index = json.load(open('imagenet_class_index.json'))
model = models.densenet121(pretrained=True)
model.eval()
```

Hasta aquí, lo que hemos hecho es:

- Instanciar Flask en la variable `app`
- Instanciar el modelo preentrenado de DenseNet121 y dejarlo en modo inferencia
- Cargar el índice de clases de ImageNet.

Todo esto como variables de módulo (globales), por lo cual podremos importarlas en otros módulos. Seguimos con `main.py` :

```

from app import app
from utils import get_prediction
from flask import Flask, jsonify, request

@app.route('/predict', methods=['POST'])
def predict():
    if request.method == 'POST':
        file = request.files['file']
        img_bytes = file.read()
        class_id, class_name = get_prediction(image_bytes=img_bytes)
        return jsonify({'class_id': class_id, 'class_name': class_name})

# Cuando ejecutemos el módulo, la aplicación iniciará en el puerto 5001
if __name__ == "__main__":
    app.run(port=5001)

```

- Aquí estamos diciéndole a Flask que, cuando reciba una request de tipo `POST` a la ruta `/predict`, responda con esta función (una *vista*).
- Lo que la función en cuestión hace es extraer la imagen que venga en el campo `file` de la request, convertirla a bytes y pasarla a la función `get_prediction`, que retorna los resultados de la clasificación (un número de clase y el nombre de la misma). Luego retorna un `JSON` con dicha información.
- Sin embargo, esa función `get_prediction` aún no existe. Desarrollémosla.

Vamos ahora al módulo `utils.py`:

```

from app import model, imagenet_class_index
import io

import torchvision.transforms as transforms
from PIL import Image

def transform_image(image_bytes):
    my_transforms = transforms.Compose([transforms.Resize(255),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize(
                                           [0.485, 0.456, 0.406],
                                           [0.229, 0.224, 0.225])])

    image = Image.open(io.BytesIO(image_bytes))
    return my_transforms(image).unsqueeze(0)

def get_prediction(image_bytes):
    tensor = transform_image(image_bytes=image_bytes)
    outputs = model.forward(tensor)
    _, y_hat = outputs.max(1)
    predicted_idx = str(y_hat.item())
    return imagenet_class_index[predicted_idx]

```

- Aquí lo que estamos haciendo es usar una función `transform_image` para darle a la imagen de entrada el formato impuesto por DenseNet, que corresponde a una imagen de 224x224. También aprovechamos de aplicar una normalización de intensidades según valores recomendados por la literatura.
- En `get_prediction`, por otro lado, llamamos a la transformación de la imagen, le hacemos `forward` por el modelo y calculamos la clase predicha.

3. Y eso es todo! Al menos con la API. Para lanzarla, ejecutamos el siguiente comando:

```

# En pytorch-api
python main.py

```

Y con eso ya tenemos lista la API. Sin embargo, si vamos a `localhost:5001` veremos que no hay nada (de hecho, la URL no existe).

- Está bien. No nos interesa usar el sitio para mostrar nada, solo queremos que cuando le llegue una request POST, responda con un json.
- Lo que hacemos al entrar al sitio es una request GET. Como no pusimos nada para este caso en `main.py`, es normal que no veamos nada ;)

- Para probar que la API esté funcionando existen diversas herramientas, la más conocida es Postman. También hay sitios web que hacen el mismo trabajo, como [este](#).

## Creando la aplicación de Frontend

Tenemos una API que hace lo que queremos, pero no tenemos ninguna aplicación que la consulte con datos ingresados por el usuario. Ahora abarcaremos esa tarea.

1. Ahora trabajamos en la carpeta `classifier-app` y comenzamos creando los mismos scripts de antes: `app.py`, `main.py`, `utils.py`.
2. Vamos poblando los scripts. Comenzamos con `app.py`:

```
from flask import Flask

UPLOAD_FOLDER = 'static/uploads/'

app = Flask(__name__)
app.secret_key = "secret key"
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

- Nótese que fijamos una variable global `UPLOAD_FOLDER` y la setteamos como parte de la configuración de la app. Ahí es donde quedarán los archivos que vayamos recibiendo.
- También setteamos `MAX_CONTENT_LENGTH`. Este es el tamaño máximo de los archivos que nos pueden enviar, por ahora pusimos 16MB.
- También les dejé `app.secret_key` con un placeholder. Como esta aplicación las usarán usuarios reales, es importante que fijemos una llave para proteger a los clientes en ciertos aspectos (véase [Flask weak secret key - Vulnerabilities - Acunetix](#))

Continuemos ahora con `utils.py`. A priori solo necesitaremos la siguiente función:

```
ALLOWED_EXTENSIONS = set(['png', 'jpg', 'jpeg'])

def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in
ALLOWED_EXTENSIONS
```

Que usaremos para verificar si nos llegan imágenes en formatos apropiados. Vamos ahora por `main.py`:

```

import os
import json
from app import app
import requests
from flask import request, redirect, url_for, render_template
from werkzeug.utils import secure_filename
from utils import allowed_file

@app.route('/')
def index_form():
    return render_template('index.html')

@app.route('/', methods=['POST'])
def index_image():
    if 'file' not in request.files:
        error = 'No se envió ningún archivo'
        return render_template('index.html', error=error)
    file = request.files['file']
    if file.filename == '':
        error = 'No se seleccionó ningún archivo'
        return render_template('index.html', error=error)
    if file and allowed_file(file.filename):
        # Podría mejorarse: usar un hash para evitar sobreescribir.
        filename = secure_filename(file.filename)
        filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(filepath)
        files = {'file': open(filepath, 'rb')}
        apicall = requests.post('http://127.0.0.1:5001/predict', files=files)
        if apicall.status_code == 200:
            error = None
            apicall = json.loads(apicall.content.decode('utf-8'))
            result = {'predicted_label': apicall['class_name'], 'class_id':
apicall['class_id']}
        else:
            error = 'Error al procesar la imagen'
            result = {'predicted_label': None, 'class_id': None}
        return render_template('index.html', filename=filename, result=result,
error=error)
    else:
        error = 'Archivo no permitido. Solo se permite JPG, JPEG o PNG.'
        return render_template('index.html', error=error)

@app.route('/display/<filename>')
def display_image(filename):
    return redirect(url_for('static', filename='uploads/' + filename), code=301)

```



```
if __name__ == "__main__":
    app.run(port=5000)
```

- Comenzamos viendo `index_form()`. Esta función, enrutada `/`, será llamada cuando se acceda a `https://localhost:5000`, y lo que hará es renderizar el HTML `index.html`. Nótese que por defecto Flask asocia las rutas al método GET.
- Luego tenemos `index_image()`, asociada al método POST. Esta función sirve el mismo HTML anterior, sin embargo, lo hace con "información adicional" (campos filename, error, result). Al respecto:
  - Usaremos POST para subir la imagen, por lo cual será razonable esperar que si llega una solicitud POST, venga con imagen.
  - ...Sin embargo, no podemos darlo por hecho: podría ser una solicitud hecha desde afuera de nuestra aplicación, y dejarla pasar sin verificar algunas condiciones sería peligroso.
  - Por eso es que tenemos tantos ifs :). En clase los revisaremos en detalle. Para el registro, lo más importante es el siguiente bloque:

```
if file and allowed_file(file.filename):
    # Podría mejorarse: usar un hash para evitar sobrescribir.
    filename = secure_filename(file.filename)
    filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
    file.save(filepath)
    files = {'file': open(filepath, 'rb')}
    apicall = requests.post('http://127.0.0.1:5001/predict', files=files)
    if apicall.status_code == 200:
        error = None
        apicall = json.loads(apicall.content.decode('utf-8'))
        result = {'predicted_label': apicall['class_name'], 'class_id':
apicall['class_id']}
```

- Aquí lo que estamos haciendo es que, si nos llega un archivo y es de un formato que nos sirve, lo guardamos en nuestro servidor y lo abrimos como archivo binario.
- Luego enviamos ese binario a `http://127.0.0.1:5001/predict`, que corresponde al puerto en donde está corriendo la API!
- Guardamos lo que nos responda la API en una variable `apicall`, y revisamos que tenga `status_code = 200` (OK). De ser así, decodificamos su contenido y lo usamos para armar la respuesta en el diccionario `result`, que posteriormente entregamos a la función `render_template`.

- De esta manera, ya tenemos nuestra aplicación y la API conversando :)
  - Por último tenemos la función `display_image`, que recibe un `filename` y se enruta a `/display/<filename>`. Esta función la usaremos para servir la imagen que nos subieron de vuelta al usuario, junto a su etiqueta predicha.
3. Ya estamos casi listos! solo nos falta hacer el "rostro" de nuestro sitio. Para ello ejecutamos los siguientes comandos:

```
# en classifier-app
mkdir static
mkdir static/css
mkdir static/uploads
mkdir templates
```

- Que serán nuestros directorios para guardar los archivos estáticos (por ahora solo tendremos css y imágenes, que guardaremos en uploads) y templates (html).
- Luego creamos un archivo de nombre `index.html` en `templates`, y le damos el siguiente contenido:

```

<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ImageNet Classification App</title>
  <!-- Una fuente de Google -->
  <link href='https://fonts.googleapis.com/css?family=Roboto Condensed'
rel='stylesheet'>
  <!-- Cargamos el CSS que haremos para este sitio -->
  <link href="{ url_for('static' , filename='css/index.css') }"
rel="stylesheet" type="text/css"/>
</head>

<div id="clf-container">
  <h1>ImageNet Classifier App</h1>
  <div>
    Aplicación de ejemplo para CC6409. El formulario recibe una imagen y le
    asocia su clase de ImageNet según
    DenseNet121 a través de un llamado a una API, también en Flask.
  </div>
  <form action="/CC6409-Flask/classifier-app/static" class="form-classifier"
  enctype="multipart/form-data" method="post">
    <input autocomplete="off" name="file" required type="file">
    <button class="hvr-grow-shadow" type="submit">Enviar</button>
  </form>
  {% if filename %}
  {% if result['predicted_label'] is not none %}
  <div class="response response-ok">
    Clase predicha: <b>{{ result['predicted_label'] }}</b>
  </div>
  {% else %}
  <div class="response response-error">
    Error de predicción. No se encontró ninguna clase.
    {% if error %}
    <div>Detalle del error: {{error}}</div>
    {% endif %}
  </div>
  {% endif %}
  
  {% endif %}
  {% if not filename and error %}
  <div class="response response-ok">
    Error: {{error}}
  </div>
  {% endif %}
</div>
</body>
</html>

```

- Algunos aspectos importantes:
- Nótese que usamos instrucciones de ¿Python? entre llaves y paréntesis. Esto resulta útil para mostrar distintos contenidos según las variables que hayamos entregado a la función `render_template`, específicamente según `error`, `filename` y `result`.
- ...Y no, no son instrucciones de Python. Son instrucciones de [Jinja2](#), un motor para plantillas web. Funcionan casi igual, pero su sintaxis es algo distinta. `{% llave y paréntesis para instrucciones%}`, `{{doble llave para variables}}`.
- En el `head`, segundo `link`, aparece la instrucción `url_for('static', ...)`. Esta función es una magia de Flask que internamente genera vistas (como las que codeamos en `main.py`) para traer archivos estáticos, los cuales busca en la carpeta `static`. En este caso la usamos para ir a buscar un archivo `css` que ya pronto crearemos.
- En el tag de `img` volvemos a usar esta función `url_for`, esta vez para pedir la url asociada a la función `display_image` con el parámetro `filename`. Esto nos entrega una URL con la imagen que recibimos, lo cual usamos para desplegarla de vuelta al usuario.
- Lo demás es puro HTML. Si alguien no es del DCC o necesita ayuda entendiéndolo, puede escribirme :)
- Por último, sacamos el archivo `index.css` del zip que descargamos al inicio y lo ponemos en `static/css`.
- Este archivo tiene reglas de estilo, que es una forma de decirle al explorador cómo debe mostrar nuestro HTML (colores, tamaños, espacios, etc).


...Y estamos listos con el frontend! Para levantarlo, corremos `python main.py` (desde la carpeta `classifier-app`). Esto dejará el servidor corriendo en el puerto 5000, y podremos acceder desde `localhost:5000`. Deberían ver algo más o menos así (sin el gato):

### ImageNet Classifier App

Aplicación de ejemplo para CC6409. El formulario recibe una imagen y le asocia su clase de ImageNet según DenseNet121.

Imagen:  Ninguno ...hivo selec.

Clase predicha: **tiger\_cat**



Si no apagaron su API mientras hacían esto, deberían estar listos. Si no, acá estan las instrucciones finales para lanzar nuestra aplicación:

Primero levantamos la API:

```
cd pytorch-api
python main.py
```

Y luego el servidor de frontend:

```
cd classifier-app
python main.py
```

---

Los códigos relativos a la carpeta `pytorch-api` han sido extraídos del tutorial original de Pytorch para Flask [Deploying PyTorch in Python via a REST API with Flask — PyTorch Tutorials 1.12.1+cu102 documentation](#) y se encuentran bajo licencia MIT.

*CC6409 - Taller de Desarrollo de Proyectos de IA, 2022-2.*

*Profesor: Juan Manuel Barrios*

*Auxiliar: Matías Vergara Silva*

*Ayudante: Felipe Arias*