Deployment of Deep Learning Architectures in Docker Containers

Elias Alfonso Carrasco Guerrero[1]

University of Alicante, Alicante, Spain
eacg2@gcloud.ua.es

Abstract. This report describes the setup, configuration, and deployment of a deep learning inference system within Docker containers, as specified in the practical assignment. Using a pre-trained neural network model for digit recognition, this system comprises two main containers: one for inference using a Flask API and another for a Gradio-based web interface. The containers communicate through a Docker Compose network setup. The main goal of the project is to illustrate the encapsulation of deep learning models and graphical interfaces within isolated Docker environments to facilitate modular deployment. Key configuration files, Dockerfiles, and performance-related observations are discussed in detail.

Keywords: Docker, Deep Learning, Flask, Gradio, Inference, Docker Compose

1 Introduction

The aim of this assignment is to encapsulate a deep learning model for digit recognition within Docker containers, separating the model inference and user interface for enhanced modularity. This separation enables flexible deployment and minimizes dependencies between components. The selected configuration employs a neural network trained on the MNIST dataset to recognize handwritten digits, with Flask for API-based inference and Gradio for the user interface.

2 System Architecture

The system architecture consists of two primary containers:

- **Inference Container**: Hosts the TensorFlow-based inference model and serves predictions via a Flask API.
- **Interface Container**: A Gradio-based web interface enables users to interact with the model by drawing digits and receiving predictions.

Both containers are orchestrated through Docker Compose, ensuring network isolation and simplified setup.

3 Docker Configuration and Setup

3.1 Docker Compose Configuration

Docker Compose is utilized to define and coordinate the two containers. The `docker-compose.yml` file specifies each service, ports, dependencies, and environment variables. This setup guarantees that the interface container waits until the inference container is ready before making requests.

Listing 1.1. docker-compose.yml

```
version: '3'
services:
  inference:
    build: ./docker-tf-practica
    ports:
      - "5000:5000"
    container_name: contenedor_inferencia
    environment:
      - PYTHONUNBUFFERED=1

  interface:
    build: ./gradio-interface
    ports:
      - "7860:7860"
    container_name: contenedor_interface
    depends_on:
      - inference
    environment:
      - PYTHONUNBUFFERED=1
```

3.2 Inference Container

The inference container runs on a TensorFlow-based image. It hosts the Flask API server, which receives the user's image, preprocesses it, and returns a prediction. The following Dockerfile defines the environment setup, including required libraries.

Listing 1.2. docker-tf-practica/Dockerfile

```
FROM tensorflow/tensorflow:2.8.0
WORKDIR /app
RUN pip install flask Pillow
COPY . /app
EXPOSE 5000
CMD ["python", "-u", "inference.py"]
```

3.3 Interface Container

The interface container provides the Gradio front-end, where users can draw digits on a canvas. This Dockerfile installs Gradio and requests modules, then runs the interface.

Listing 1.3. gradio-interface/Dockerfile

```
FROM python:3.8-slim
WORKDIR /app
RUN pip install gradio requests
COPY . /app
EXPOSE 7860
CMD ["python", "-u", "interface.py"]
```

4 System Implementation

The deep learning inference service is structured into three main steps: training on the MNIST dataset, preprocessing images, and making predictions. Each component is encapsulated within a Flask API for inference within a Docker container.

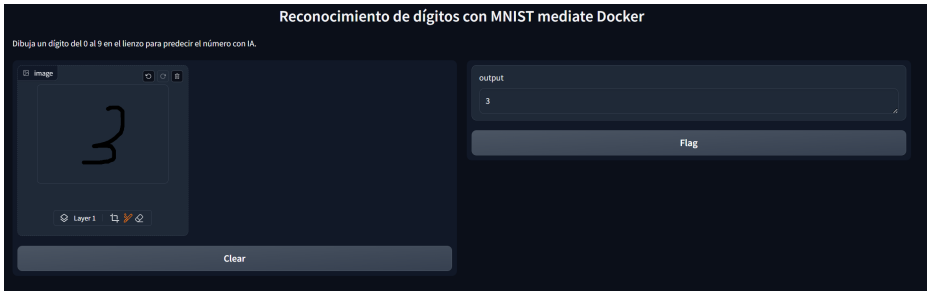

Fig. 1. MNIST interface with gradio

4.1 Model Training on the MNIST Dataset

The first step involves training a neural network on MNIST, a widely-used dataset of 28x28 grayscale digit images (0-9). The network architecture comprises two hidden layers with 256 and 128 neurons, respectively, each using the ReLU activation for non-linearity. To prevent overfitting, a dropout layer with a 0.1 rate is included after the second hidden layer. The output layer has 10 neurons with softmax activation, providing probability distributions across classes.

The model is compiled with the Adam optimizer and sparse categorical cross-entropy loss, and is trained over 10 epochs with a validation split of 0.3.

Listing 1.4. Model Training Code

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
model.fit(x_train, y_train, epochs=10, validation_split=0.3)
```

4.2 Image Preprocessing

After training, input images are preprocessed to ensure compatibility with the model. The image is converted to grayscale and resized to 28x28 pixels, matching the MNIST format. The resized image is then transformed into a Numpy array, and a batch dimension is added for model compatibility. The preprocessing function includes a check for empty images, returning an error if no user input is detected.

Listing 1.5. Image Preprocessing Code

```
def preprocess_image(image_data):
    try:
        img = Image.open(io.BytesIO(image_data)).convert('L')
        img = img.resize((28, 28))
        img_array = np.array(img)

        if np.all(img_array == 255):
            print("Empty image detected.")
            return None

        img_array = np.expand_dims(img_array, axis=0)
        return img_array
    except Exception as e:
        print(f"Error processing image: {e}")
        return None
```

4.3 Making Predictions

With the preprocessed image, the Flask APIâĂŹs ‘/predict‘ endpoint handles predictions. The endpoint reads and processes the image, passing it to ‘model.predict‘. The result is an array of probabilities for each digit, from which the label with the highest probability is selected using ‘np.argmax‘. The prediction is then returned in JSON format for interpretation by the interface container.

Listing 1.6. Prediction Code

```
@app.route('/predict', methods=['POST'])
def predict():
    try:
        file = request.files['file']
        image_data = file.read()
        img_array = preprocess_image(image_data)

        if img_array is None:
            return jsonify({'error': 'Empty or failed image processing'})

        predictions = model.predict(img_array)
        predicted_label = np.argmax(predictions)

        return jsonify({'prediction': int(predicted_label)})
    except Exception as e:
        return jsonify({'error': str(e)})
```

5 Docker Concepts and Commands

Docker enables the creation and management of isolated environments, or containers, that package applications and their dependencies. This project leverages Docker's capabilities to separate components into distinct containers for modularity, using Docker Desktop and terminal commands for management. Key Docker concepts and commands used in this setup include multi-container applications, multi-layered images, and efficient container management.

5.1 Docker Compose for Multi-Container Applications

Docker Compose is a tool that defines and manages multiple containers within a single application, using a `docker-compose.yml` file. This setup is beneficial when components, such as a backend server and a frontend interface, need to work together but benefit from isolation. In this project, Docker Compose orchestrates two main services: the inference container and the Gradio interface container.

To work efficiently with Docker Compose, several commands were used to start, manage, and monitor these containers:

- **Updating Services Without Restarting**: Docker Compose can automatically watch for file changes, thanks to tools like `nodemon`, which allows changes in code to be detected and reloaded without stopping the containers. This is useful during development for live updates. The command:

  ```
  docker compose watch
  ```

  runs in the terminal to watch for code changes in real-time.

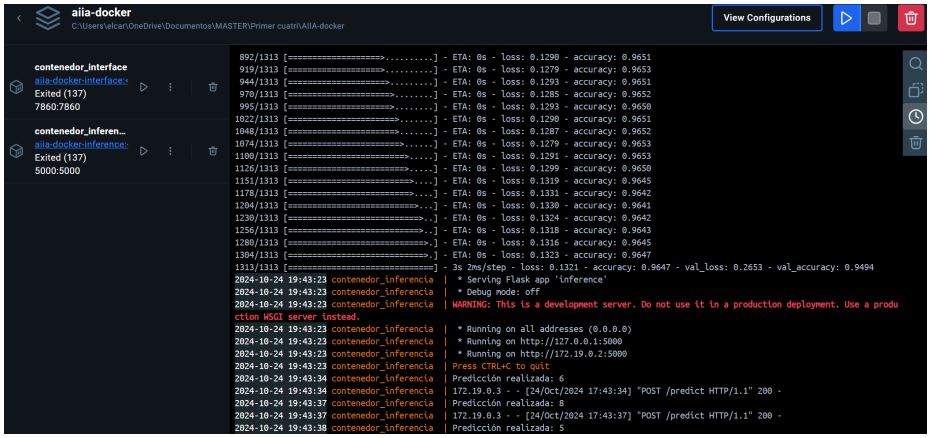

Fig. 2. Executing a multi-container application with Docker

– **Running Services in Detached Mode**: To run multiple containers together without monitoring for changes, the command:

```
docker compose up –d
```

launches all containers in detached mode. This is typically used in production environments or when monitoring changes is not needed.

5.2 Rebuilding and Managing Docker Containers

During development, it is often necessary to rebuild images to reflect updates in code or dependencies. The following commands facilitate container lifecycle management:

– `docker-compose down`: Stops and removes all containers in the Compose setup, ensuring a clean environment for any updates or rebuilds.
– `docker-compose up -build`: Rebuilds and runs the containers, incorporating any changes in the `Dockerfile` or codebase. The `-build` flag forces a rebuild of the images before starting the containers, ensuring that all updates are reflected.

5.3 Multi-Layer Images and Dockerfile Optimization

To improve efficiency, Docker images are built in layers, with each command in the `Dockerfile` adding a layer. Layers allow for caching, so when rebuilding, unchanged layers are reused, reducing build times. Optimization strategies used include:

– Base Images and Layered Builds: This project uses specific base images (`tensorflow/tensorflow:2.8.0` for the inference container and `python:3.8-slim`

for the Gradio interface). Each base image is selected to provide only the necessary dependencies, minimizing image size and improving startup times.
- Order of Instructions in the Dockerfile: Instructions that change frequently, such as copying code files, are placed toward the end of the Dockerfile. This leverages Docker's caching mechanism by reusing the initial layers (base image and library installations) and rebuilding only the final layers with modified code.
- Minimal Dependencies: Only essential libraries (`flask`, `Pillow`, `gradio`, and `requests`) are installed in each container. This reduces image size, speeds up builds, and simplifies dependency management.

5.4 Creating Components and Images in Docker

In Docker terminology, an image is a read-only template that serves as the base for running a container. Components within Docker can refer to individual services defined in the Compose file or to distinct Dockerfiles that build these images. For example:

- Components: Each service (inference and interface) is defined as a component in Docker Compose, with individual configurations for each. This approach maintains modularity, making each component independently manageable.
- Image Layers: Using multi-layered builds, Docker caches and reuses unchanged layers when building images, which enhances efficiency during frequent rebuilds.
- Custom Containers: By creating custom Dockerfiles, each component is optimized for its specific task, from the backend model inference to the frontend interface, resulting in a well-organized, multi-container application.

6 Results and Observations

Testing the system confirms that the containers effectively isolate the interface and inference functionalities, and Gradio provides a straightforward interface for user input. Inferences are accurate with sufficient training epochs, and the modular container structure allows easy scaling and maintenance.

7 Conclusions

This project demonstrates the deployment of a deep learning model in a modular, containerized environment using Docker. The separation of inference and user interface into distinct containers allows improved scalability and flexibility, and Docker Compose streamlines the setup and network configuration. Potential enhancements include integrating GPU support for faster inferences and deploying the model to a cloud service for broader accessibility.