Project 2: ML Inference - Design Document

Introduction

The project implements a simple image inference serving system that handles *HTTP* requests to query the model and return the classification results. The backend server is a *Flask* application which uses a pre-trained *Densenet-121* model from the open source machine learning library *PyTorch* to perform image classification. The server is hosted in a *Docker* container using a *Python 3.7* image and a basic client program written in *Python* provides the ability to query the system using an *HTTP POST* request that sends the image file to the server. The server responds with the inferred *class ID* and *class name* for the specified image.

Pytorch Densenet

Pytorch DenseNet is a *dense convolutional network*. The DenseNet architecture has one convolutional layer connected to all other ones. This is in contrast to traditional CNN models with L layers, where a convolutional layer is connected to only the subsequent one leading to L connections. Densenet has *L(L+1)/2* connections due to its fully connected nature. The DenseNet architecture has been shown to have multiple advantages including the following:

1. The vanishing gradient problem is handled well by the architecture.
2. Feature propagation is strengthened due to the dense connection between layers.
3. Features are reused as all the layers are interconnected and a layer can use any other layers' output as an input.
4. Lesser number of parameters.

In this project, we are using a pre-trained version of DenseNet-121 to run inference on images that are being sent to the server that is hosted in a Docker container.

ML Inference using a pre-trained Densenet-121 model and Flask on Docker

The application runs using a *client-server* architecture. The server runs on a *Docker* container on localhost and the client, running on the local machine, can make a *HTTP POST* request with the image to the server.

1. **Docker:** The environment required to run the server is defined and set up using *./prediction/Dockerfile*. The docker container used to run the server is created using *Python 3.7* as the base image. Python libraries like *Pillow, torch, torchvision*, and *flask* are installed. Then the python script *app.py* is executed in this environment to start the server.
2. **Server:** The server is a *Flask* application running on localhost (*127.0.0.1*) at port *5000*. It receives an *HTTP POST* request on the path */run_inference* through which it receives an image. It then calls the *predict_label()* function that processes the image by resizing and transforming it so that it can be provided as an input to *Densenet-121*. This image is then sent to the *Densenet-121* model which returns a vector with the probability values for each label. The file *imagenet_class_index.json* was downloaded from the *ImageNet* website to map the label with the *highest probability* with the correct class ID and class name. This class ID and class name is converted to JSON format and returned as the output.
3. **Client:** The client is a *Python* program running on the local machine (outside the container). The container exposes port *5000* so that an *HTTP POST* can be sent to the server. The client sends a *POST* request, using Python's *requests* library, to *localhost:5000/run_inference*. The response to this request is the JSON file that is returned by the server containing the *class ID* and the *class name* of the image.

Design Trade-Off

The project is a minimalistic implementation of an inference serving system that uses a pretrained model to perform image classification.

Running a Local Instance of the Application: The system is designed to run on a docker container which hosts the application on localhost, thus allowing it to be accessed only via HTTP POST requests on a local instance. Ideally, the docker image would be deployed to some cloud system with access management for the HTTP requests using the configuration parameters of the cloud system. In such a case, the docker-compose config file would contain all the relevant parameters to set up the container as well and we would directly use the *docker-compose up* command instead of manually running the *docker run -p 5000:5000* we're using to facilitate the local HTTP requests.