Supporting Object Detection Across Multiple Deep Learning Frameworks - High Level Steps

A modular, scalable approach to support object detection using different DL frameworks such as TensorFlow, PyTorch, and ONNX.

0. Running Models via Docker & Docker Compose

```
services:
  tf_model:
    container_name: tf_model
    image: tensorflow/serving
    ports:
      - "8501:8501"
    # Add appropriate model-store and config

  pytorch_model:
    container_name: pytorch_model
    image: pytorch/torchserve:latest
    ports:
      - "8502:8080"
    # Add appropriate model-store and config

  onnx_model:
    container_name: onnx_model
    image: mcr.microsoft.com/onnxruntime/server
    ports:
      - "8503:8001"
    # Add appropriate model-store and config
```

Run the services with:

```
docker-compose up -d
```

1. Interface (Common Base for All Models)

```
# domain/ports.py
from abc import ABC, abstractmethod
from typing import BinaryIO, List
from counter.domain.models import Prediction

class ObjectDetectionModel(ABC):
    @abstractmethod
    def predict(self, image: BinaryIO) → List[Prediction]:
        pass
```

2. Concrete Model Adapters (One per Framework)

a. TensorFlow Adapter

```
# adapters/models/tensorflow_model.py
import tensorflow as tf

class TensorFlowObjectDetector(ObjectDetectionModel):
    def __init__(self, host, port, model):
        self.url = f"http://{host}:{port}/v1/models/{model}/predict"

    def predict(self, image):
        # Preprocess, infer, postprocess
        return parsed_predictions
```

b. PyTorch Adapter

```
# adapters/models/pytorch_model.py
import torch
```

```
class PyTorchObjectDetector(ObjectDetectionModel):
    def __init__(self, host, port, model):
        self.url = f"http://{host}:{port}/v1/models/{model}/predict"

    def predict(self, image):
            # Preprocess, infer, postprocess
        return parsed_predictions
```

c. ONNX Adapter

```
# adapters/models/onnx_model.py
import onnxruntime

class ONNXObjectDetector(ObjectDetectionModel):
    def __init__(self, host, port, model):
        self.url = f"http://{host}:{port}/v1/models/{model}/predict"

    def predict(self, image):
            # Preprocess, infer, postprocess
        return parsed_predictions
```

3. Factory Function to Dynamically Select the Model

Use a central function to map model names to the correct implementation:

```
# config.py
def get_model(model_name: str) -> ObjectDetectionModel:
    if model_name == "rfcn":
        return TensorFlowObjectDetector("tf_model", 8501, "rfcn")
    elif model_name == "yolov5":
        return PyTorchObjectDetector("pytorch_model", 8502, "yolov5")
    elif model_name == "ssd_onnx":
        return ONNXObjectDetector("onnx_model", 8503, "ssd_onnx")
    else:
        raise ValueError(f"Unsupported model: {model_name}")
```

4. Execution (Orchestration Layer)

```
model = get_model(requested_model)
results = model.predict(image)
```

Optional Enhancements

- **Model Registry**: Maintain a JSON or database config for all supported models.
- **Caching**: Avoid repeated loading using `@lru_cache` or Singleton pattern.
- **GPU/Device Support**: Add framework-specific flags to control device behavior.
- **Batch Support**: Extend the `predict()` method to optionally take batches.