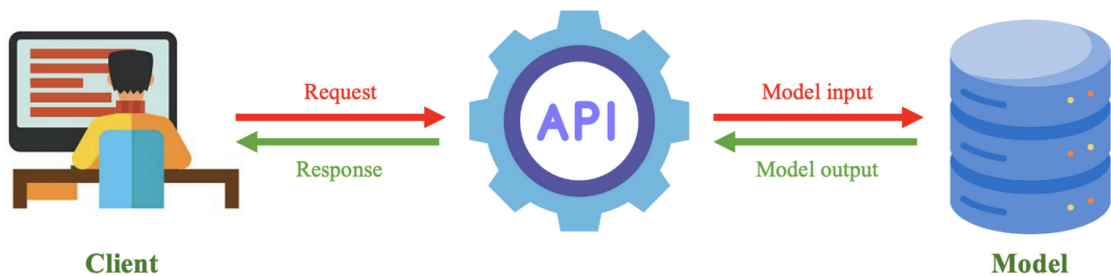


CHƯƠNG 1. TRIỂN KHAI MÔ HÌNH DEEP LEARNING VỚI FASTAPI

1.1 Giới thiệu

Triển khai mô hình Deep Learning (Model Deployment) là một trong những công việc quan trọng trong quy trình đưa mô hình ra sản phẩm thực tế, giúp cho người sử dụng có thể tương tác với trực tiếp với mô hình. Một trong những phương thức giúp chúng ta có thể làm điều này đó là triển khai mô hình thành một API. Theo đó, người dùng có thể thông qua API endpoint để sử dụng mô hình của chúng ta mà không cần phải sở hữu mã nguồn chương trình cũng như tài nguyên tính toán.



Hình 1.1: Triển khai mô hình Deep Learning dưới dạng một API.

Trong bài viết này, chúng ta sẽ triển khai một mô hình về **Image Classification** bằng thư viện FastAPI trong Python. Sau khi hoàn thành chương trình, chúng ta sẽ có thể sử dụng mô hình thông qua một địa chỉ web (URL).



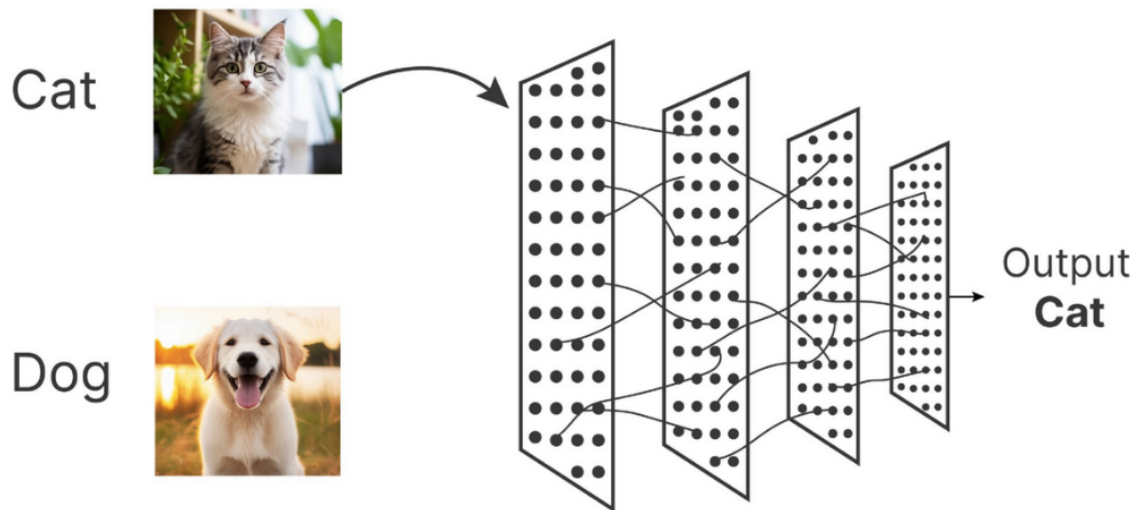
Hình 1.2: Logo của thư viện FastAPI.

1.2 Nội dung

Ở phần này, chúng ta sẽ xây dựng hai thành phần chính của chương trình bao gồm phần **xây dựng mô hình** và phần **FastAPI**. Cách thực hiện như sau:

1.2.1 Xây dựng mô hình

Đầu tiên, chúng ta sẽ bắt đầu với giai đoạn đầu như mọi project về Machine Learning đó là xây dựng mô hình. Như đã đề cập trong bài viết, chúng ta sẽ xây dựng một mô hình về Image Classification, cụ thể là bài toán Cat/Dog Classification.



Hình 1.3: Ảnh minh họa bài toán Cat/Dog Classification

Ở đây, ta sẽ sử dụng thư viện PyTorch và Google Colab để giải quyết phần này như sau:

a, Tải và import các thư viện cần thiết

Đầu tiên, ta cài đặt thư viện `datasets` để sử dụng trong việc tải dữ liệu Cat Dog về sau:

```
1 | pip install datasets==2.18.0 -q
```

Sau đó, ta thực hiện import các thư viện cần thiết trong việc huấn luyện mô hình:

```
1 | import torch
2 | import torch.nn as nn
3 |
4 | from PIL import Image
5 | from datasets import load_dataset
6 | from torch.utils.data import Dataset, DataLoader
7 | from torchvision.models import resnet18
8 | from torchvision import transforms
```

b, Tải bộ dữ liệu

Chúng ta sẽ sử dụng bộ dữ liệu Cat Dog có sẵn trên HuggingFace Dataset. Các bạn hãy chạy đoạn code bên dưới để tải cũng như load bộ dữ liệu:

```
1 | DATASET_NAME = "cats_vs_dogs"
2 | datasets = load_dataset(DATASET_NAME)
3 | datasets
```

Dataset Preview ⓘ






</> API

View in Dataset Viewer

Split (1)
train

▼

▶ The full dataset viewer is not available (click to read why). Only showing a preview of the rows.

image image	labels class label
	0 cat
	0 cat
	0 cat
	0 cat
	0 cat

Hình 1.4: Bộ dữ liệu về [Cat Dog Image Classification](#) trên HuggingFace Dataset.

c, Chia bộ dữ liệu train, val:

Từ bộ dữ liệu gốc, ta sẽ tách thành hai bộ train và val để phục vụ cho việc huấn luyện mô hình như sau:

```
1 TEST_SIZE = 0.2
2 datasets = datasets['train'].train_test_split(test_size=TEST_SIZE)
```

d, Xây dựng DataLoader

Để thuận tiện trong việc đọc dữ liệu khi thực hiện huấn luyện với PyTorch, chúng ta sẽ xây dựng DataLoader cho hai bộ train và val như sau:

- **Xây dựng hàm tiền xử lý dữ liệu ảnh:** Chúng ta sẽ xây dựng hàm transforms trong PyTorch để tiền xử lý dữ liệu ảnh như sau:

```
1 IMG_SIZE = 64
2 img_transforms = transforms.Compose([
3     transforms.Resize((IMG_SIZE, IMG_SIZE)),
4     transforms.Grayscale(num_output_channels=3),
5     transforms.ToTensor(),
6     transforms.Normalize(
7         mean=[0.485, 0.456, 0.406],
8         std=[0.229, 0.224, 0.225]
9     )
10 ])
11
```

- **Xây dựng class CatDogDataset:** Chúng ta xây dựng class CatDogDataset để đọc dữ liệu cho DataLoader:

```

1 class CatDogDataset(Dataset):
2     def __init__(self, data, transform=None):
3         self.data = data
4         self.transform = transform
5
6     def __len__(self):
7         return len(self.data)
8
9     def __getitem__(self, idx):
10        image = self.data[idx]['image']
11        label = self.data[idx]['labels']
12
13        if self.transform:
14            image = self.transform(image)
15
16        label = torch.tensor(label, dtype=torch.long)
17
18        return image, label
19

```

- **Khai báo DataLoader:** Cuối cùng, ta khai báo hai DataLoader với số batch size phù hợp:

```

1 # Batch size constants
2 TRAIN_BATCH_SIZE = 512
3 VAL_BATCH_SIZE = 256
4
5 # Dataset initialization
6 train_dataset = CatDogDataset(datasets['train'], transform=img_transforms)
7 test_dataset = CatDogDataset(datasets['test'], transform=img_transforms)
8
9 # Data loaders
10 train_loader = DataLoader(
11     train_dataset, batch_size=TRAIN_BATCH_SIZE, shuffle=True
12 )
13 test_loader = DataLoader(
14     test_dataset, batch_size=VAL_BATCH_SIZE, shuffle=False
15 )
16

```

e, Xây dựng mô hình

Ta xây dựng class CatDogModel cho bài toán Cat Dog Classification với backbone là pre-trained resnet18 như sau:

```

1 class CatDogModel(nn.Module):
2     def __init__(self, n_classes):
3         super(CatDogModel, self).__init__()
4
5         # Load pre-trained ResNet-18 model

```

```

6         resnet_model = resnet18(weights='IMAGENET1K_V1')
7
8         # Use all layers except the final fully connected layer
9         self.backbone = nn.Sequential(*list(resnet_model.children())[:-1])
10
11        # Freeze the backbone parameters
12        for param in self.backbone.parameters():
13            param.requires_grad = False
14
15        # Replace the final fully connected layer
16        in_features = resnet_model.fc.in_features
17        self.fc = nn.Linear(in_features, n_classes)
18
19        def forward(self, x):
20            x = self.backbone(x) # Extract features using backbone
21            x = torch.flatten(x, 1) # Flatten the features
22            x = self.fc(x) # Classify using the final layer
23            return x

```

Sau đó, ta khai báo mô hình và kiểm tra thử mô hình đã hoạt động được hay chưa như sau:

```

1 device = 'cuda' if torch.cuda.is_available() else 'cpu'
2 N_CLASSES = 2
3
4 model = CatDogModel(N_CLASSES).to(device)
5
6 test_input = torch.rand(1, 3, 224, 224).to(device)
7
8 with torch.no_grad():
9     output = model(test_input)
10
11 print(output.shape)

```

f, Thực hiện huấn luyện mô hình

Với mô hình và dataset đã sẵn sàng, chúng ta sẽ khai báo optimizer, criterion cũng như vòng lặp training để huấn luyện mô hình như sau:

```

1 # Hyperparameters
2 EPOCHS = 10
3 LR = 1e-3
4 WEIGHT_DECAY = 1e-5
5
6 # Optimizer and loss function
7 optimizer = torch.optim.Adam(model.parameters(), lr=LR, weight_decay=
    WEIGHT_DECAY)
8 criterion = torch.nn.CrossEntropyLoss()
9
10 # Training loop
11 for epoch in range(EPOCHS):
12     train_losses = []
13     model.train()

```

```

14
15     # Training phase
16     for images, labels in train_loader:
17         images = images.to(device)
18         labels = labels.to(device)
19
20         # Forward pass
21         outputs = model(images)
22         loss = criterion(outputs, labels)
23
24         # Backward pass and optimization
25         optimizer.zero_grad()
26         loss.backward()
27         optimizer.step()
28
29         # Record training loss
30         train_losses.append(loss.item())
31
32     train_loss = sum(train_losses) / len(train_losses)
33
34     # Validation phase
35     val_losses = []
36     model.eval()
37     with torch.no_grad():
38         for images, labels in test_loader:
39             images = images.to(device)
40             labels = labels.to(device)
41
42             # Forward pass
43             outputs = model(images)
44             loss = criterion(outputs, labels)
45
46             # Record validation loss
47             val_losses.append(loss.item())
48
49     val_loss = sum(val_losses) / len(val_losses)
50
51     # Print epoch results
52     print(f"EPOCH {epoch + 1}: \tTrain loss: {train_loss:.3f} \tVal loss: {
val_loss:.3f}")

```

g, Lưu trọng số mô hình

Với mô hình đã được huấn luyện xong, ta sẽ lưu mô hình lại dưới dạng file .pt và tải về máy file này:

```

1 # Path to save the model weights
2 SAVE_PATH = 'catdog_weights.pt'
3
4 # Save the model's state dictionary
5 torch.save(model.state_dict(), SAVE_PATH)

```

Như vậy, qua mục này, chúng ta đã có được một mô hình Deep Learning để sử dụng

trong API.

1.2.2 Xây dựng API

Sau khi có bộ trọng số mô hình Cat Dog Classification đã huấn luyện ở mục trước, chúng ta sẽ tiến hành xây dựng API để sử dụng mô hình này bằng FastAPI. Nội dung này chúng ta sẽ cài đặt ở máy tính cá nhân. Demo được thực hiện trên hệ điều hành MacOS. Các bước thực hiện như sau:

a, Tổ chức thư mục code:

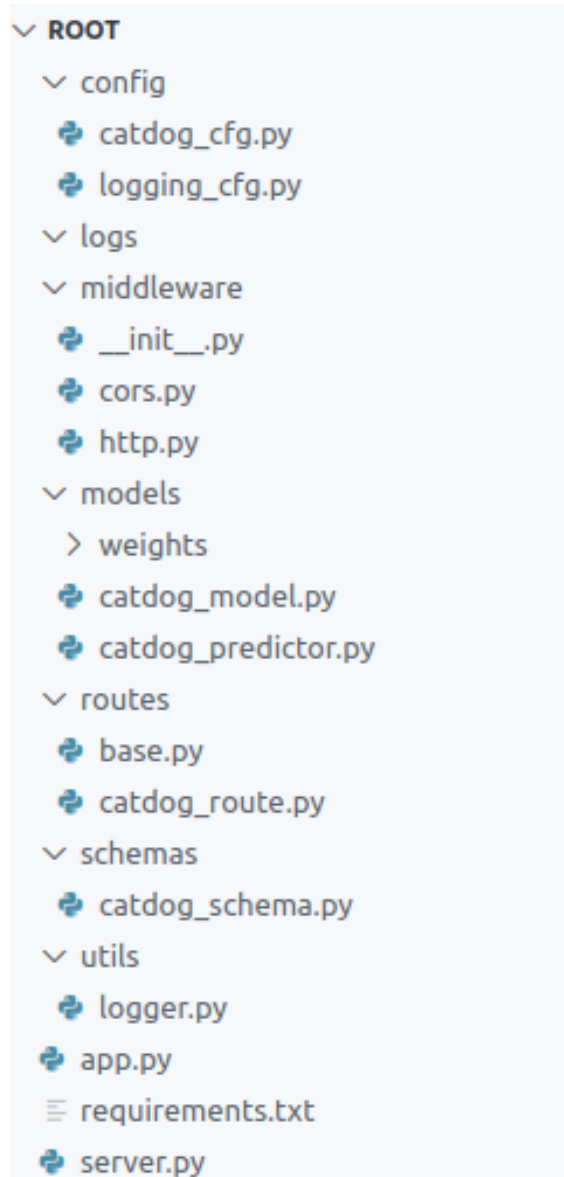
Để mã nguồn trở nên rõ ràng nhằm phục vụ cho mục đích đọc hiểu code, chúng ta sẽ tổ chức thư mục như sau:

```
root/
├── config/
│   ├── catdog_cfg.py
│   └── logging_cfg.py
├── logs/
├── middleware/
│   ├── __init__.py
│   ├── cors.py
│   └── http.py
├── models/
│   ├── weights/
│   │   └── catdog_weights.pt
│   ├── catdog_model.py
│   └── catdog_predictor.py
├── routes/
│   ├── base.py
│   └── catdog_route.py
├── schemas/
│   └── catdog_schema.py
├── utils/
│   └── logger.py
├── app.py
├── requirements.txt
└── server.py
```

Hình 1.5: Cấu trúc thư mục

Chúng ta sẽ có thư mục chứa mã nguồn có tên `root` (các bạn hoàn toàn có thể sử dụng tên gọi khác). Bên trong sẽ có các thư mục con và các tệp tin với ý nghĩa như sau:

- `config/`: Thư mục chứa code định nghĩa các tham số liên quan đến các module khác nhau như `model`, `logger`...
- `logs/`: Thư mục lưu trữ log. Log có thể bao gồm thông tin trạng thái các request mà API đã nhận được cũng như kết quả trả về của mô hình.
- `middleware/`: Thư mục chứa code tiền và hậu xử lý một request.
- `models/`: Thư mục chứa code của mô hình mà ta đã huấn luyện ở phần A.
- `routes/`: Thư mục chứa code định nghĩa các API endpoints.
- `schemas/`: Thư mục chứa code định nghĩa các cấu trúc dữ liệu input/output của API.
- `utils/`: Thư mục dùng để chứa một số code có các chức năng khác nhau tùy thuộc vào project.
- `app.py`: Tệp tin khai báo API.
- `requirements.txt`: Tệp tin chứa thông tin các gói thư viện cần thiết để chạy mã nguồn.
- `server.py`: Tệp tin để thực thi triển khai mô hình.



Hình 1.6: Minh họa cấu trúc cây thư mục của chương trình trong VSCode.

b, Cập nhật file `requirements.txt`:

Để bắt đầu, chúng ta sẽ liệt kê các gói thư viện cần thiết để chạy được chương trình này. Các bạn hãy cập nhật file `requirements.txt` với nội dung sau:

```
1 fastapi
2 uvicorn
3 torch
4 torchvision
5 python-multipart
```

c, Cập nhật thư mục `config`

Mục đích của thư mục này nhằm giúp chúng ta có thể quản lý những tham số của một số tính năng nào đó trong API, có thể kể đến là các tham số của mô hình Deep Learning, một cách thuận tiện và hiệu quả. Vì vậy, chúng ta sẽ khai báo các

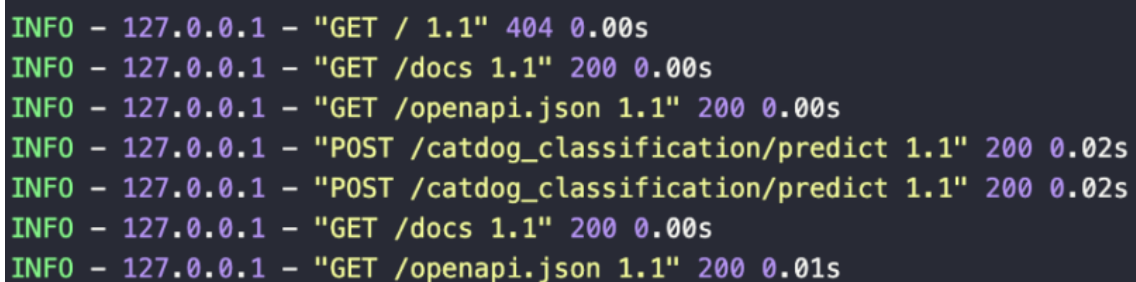
thông tin tham số cho bài toán Cat Dog Classification vào file `config/cat-dog_cfg.py` như sau:

```
1 import sys
2 from pathlib import Path
3
4 # Thêm đường dẫn của thư mục hiện tại vào sys.path
5 sys.path.append(str(Path(__file__).parent))
6
7 class CatDogDataConfig:
8     N_CLASSES = 2
9     IMG_SIZE = 64
10    ID2DLABEL = {0: 'Cat', 1: 'Dog'}
11    LABEL2ID = {'Cat': 0, 'Dog': 1}
12    NORMALIZE_MEAN = [0.485, 0.456, 0.406]
13    NORMALIZE_STD = [0.229, 0.224, 0.225]
14
15 class ModelConfig:
16     ROOT_DIR = Path(__file__).parent.parent
17     MODEL_NAME = 'resnet18'
18     MODEL_WEIGHT = ROOT_DIR / 'models' / 'weights' / 'catdog_weights.pt'
19     DEVICE = 'cpu'
```

Ở đây, ta quản lý các tham số bằng Python class. Việc đặt tên class có thể tùy chỉnh khác nhau nhưng nên được đặt tên cho phù hợp với ý nghĩa của chúng.

d, Xây dựng hàm logger:

Khi triển khai API, rất nhiều khả năng chúng ta sẽ bắt gặp lỗi ở một request tại thời điểm nào đó. Để biết chính xác thông tin này, chúng ta cần thực hiện logging. Logging là một kỹ thuật giúp chương trình ghi nhận lại lịch sử của chương trình, có thể bao gồm kết quả dự đoán của mô hình, kết quả request...



```
INFO - 127.0.0.1 - "GET / 1.1" 404 0.00s
INFO - 127.0.0.1 - "GET /docs 1.1" 200 0.00s
INFO - 127.0.0.1 - "GET /openapi.json 1.1" 200 0.00s
INFO - 127.0.0.1 - "POST /catdog_classification/predict 1.1" 200 0.02s
INFO - 127.0.0.1 - "POST /catdog_classification/predict 1.1" 200 0.02s
INFO - 127.0.0.1 - "GET /docs 1.1" 200 0.00s
INFO - 127.0.0.1 - "GET /openapi.json 1.1" 200 0.01s
```

Hình 1.7: Minh họa thông tin kết quả các lần request API mà logger ghi nhận được.

Trong bài này, chúng ta sẽ sử dụng module logging trong Python để xây dựng một logger. Ta sẽ triển khai code vào file `utils/logger.py` như sau:

```
1 import sys
2 import logging
3 from pathlib import Path
4 from logging.handlers import RotatingFileHandler
```

```

5 from config.logging_cfg import LoggingConfig
6
7 # Thêm đường dẫn của thư mục hiện tại vào sys.path
8 sys.path.append(str(Path(__file__).parent))
9
10 class Logger:
11     def __init__(self, name="", log_level=logging.INFO, log_file=None) -> None
12     :
13         self.log = logging.getLogger(name)
14         self.get_logger(log_level, log_file)
15
16     def get_logger(self, log_level, log_file):
17         self.log.setLevel(log_level)
18         self._init_formatter()
19         if log_file is not None:
20             self._add_file_handler(LoggingConfig.LOG_DIR / log_file)
21         else:
22             self._add_stream_handler()
23
24     def _init_formatter(self):
25         self.formatter = logging.Formatter(
26             "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
27         )
28
29     def _add_stream_handler(self):
30         stream_handler = logging.StreamHandler(sys.stdout)
31         stream_handler.setFormatter(self.formatter)
32         self.log.addHandler(stream_handler)
33
34     def _add_file_handler(self, log_file):
35         file_handler = RotatingFileHandler(
36             log_file, maxBytes=10000, backupCount=10
37         )
38         file_handler.setFormatter(self.formatter)
39         self.log.addHandler(file_handler)
40
41     def log_model(self, predictor_name):
42         self.log.info(f"Predictor name: {predictor_name}")
43
44     def log_response(self, pred_prob, pred_id, pred_class):
45         self.log.info(
46             f"Predicted Prob: {pred_prob} - Predicted ID: {pred_id} - Predicted
47             Class: {pred_class}"
48         )

```

Đối với việc logging, chúng ta sẽ có tham số về đường dẫn thư mục lưu file log. Vì vậy, ta cũng sẽ cập nhật tham số này trong file `config/logging_cfg.py` như sau:

```

1 from pathlib import Path
2
3 class LoggingConfig:
4     # Định nghĩa đường dẫn gốc
5     ROOT_DIR = Path(__file__).parent.parent

```

```

6
7     # Định nghĩa đường dẫn thư mục logs
8     LOG_DIR = ROOT_DIR / "logs"
9
10    # Tạo thư mục logs nếu chưa tồn tại
11    LOG_DIR.mkdir(parents=True, exist_ok=True)

```

e, Cập nhật thư mục `models/`:

Chúng ta sẽ đưa thông tin model đã huấn luyện vào thư mục chuyên dùng để chứa các mô hình đã huấn luyện. Theo đó, chúng ta sẽ lưu file `weights cat-dog_weights.pt` vào trong thư mục con `models/weights`.

Tiếp theo, để sử dụng mô hình PyTorch khi inference, chúng ta cần có định nghĩa class của mô hình cũng như hàm `prediction`. Vậy nên, ta sẽ đưa các thông tin trên thành hai file riêng biệt:

- Với `models/catdog_model.py`: Dùng để chứa định nghĩa class của mô hình. Ở đây, ta đơn giản copy lại thông tin ở file notebook vào:

```

1  import torch
2  import torch.nn as nn
3  from torchvision.models import resnet18
4
5  class CatDogModel(nn.Module):
6      def __init__(self, n_classes):
7          super(CatDogModel, self).__init__()
8
9          # Load pre-trained ResNet-18 model
10         resnet_model = resnet18(weights='IMAGENET1K_V1')
11
12         # Extract backbone layers (excluding the fully connected layer)
13         self.backbone = nn.Sequential(*list(resnet_model.children())
14                                       [:-1])
15
16         # Freeze backbone parameters to prevent updates during training
17         for param in self.backbone.parameters():
18             param.requires_grad = False
19
20         # Define a new fully connected layer for classification
21         in_features = resnet_model.fc.in_features
22         self.fc = nn.Linear(in_features, n_classes)
23
24     def forward(self, x):
25         # Pass input through the backbone
26         x = self.backbone(x)
27
28         # Flatten the output from the backbone
29         x = torch.flatten(x, 1)
30
31         # Pass the flattened output through the fully connected layer
32         x = self.fc(x)

```

```
33         return x
34
```

- Với `models/catdog_predictor.py`: Ta xây dựng class `Predictor`, dùng trong việc khởi tạo mô hình với bộ trọng số cho trước và cho phép ta thực hiện inference. Ta có thể xây dựng class này như sau:

```
1  import sys
2  from pathlib import Path
3
4  import torch
5  import torchvision
6  from PIL import Image
7  from torch.nn import functional as F
8
9  from utils.logger import Logger
10 from config.catdog_cfg import CatDogDataConfig
11 from .catdog_model import CatDogModel
12
13 # Initialize logger
14 LOGGER = Logger(__file__, log_file="predictor.log")
15 LOGGER.log.info("Starting Model Serving")
16
17 class Predictor:
18     def __init__(self, model_name: str, model_weight: str, device: str = "cpu"):
19         self.model_name = model_name
20         self.model_weight = model_weight
21         self.device = device
22         self.load_model()
23         self.create_transform()
24
25     async def predict(self, image):
26         # Open and preprocess the image
27         pil_img = Image.open(image)
28         if pil_img.mode == "RGBA":
29             pil_img = pil_img.convert("RGB")
30
31         transformed_image = self.transforms_(pil_img).unsqueeze(0)
32         output = await self.model_inference(transformed_image)
33
34         # Process model output
35         probs, best_prob, predicted_id, predicted_class = self.output2pred(
            output)
36
37         # Log the model and prediction details
38         LOGGER.log_model(self.model_name)
39         LOGGER.log_response(best_prob, predicted_id, predicted_class)
40
41         # Clear CUDA cache
42         torch.cuda.empty_cache()
43
44         # Construct response
45         resp_dict = {
```

```

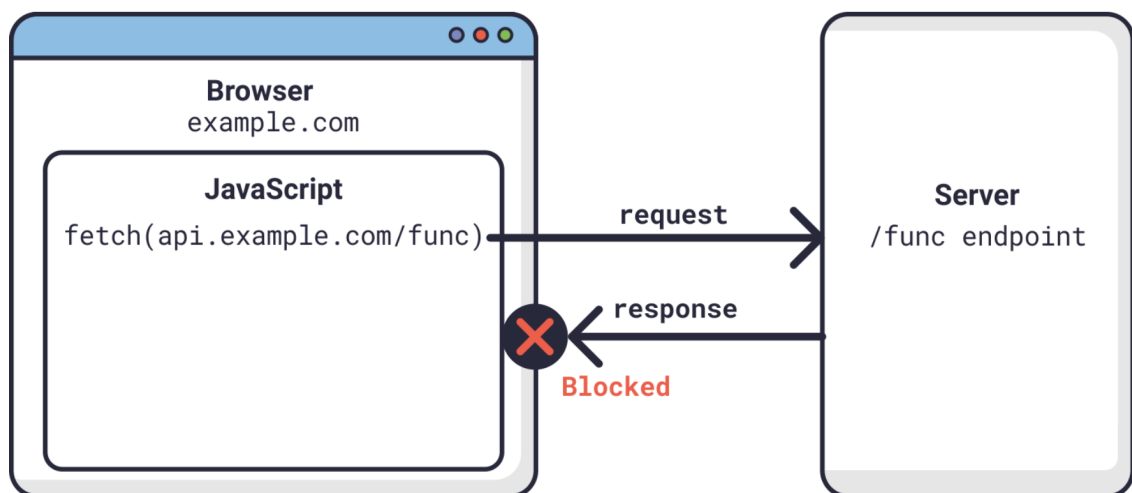
46         "probs": probs,
47         "best_prob": best_prob,
48         "predicted_id": predicted_id,
49         "predicted_class": predicted_class,
50         "predictor_name": self.model_name,
51     }
52
53     return resp_dict
54
55     async def model_inference(self, input):
56         input = input.to(self.device)
57         with torch.no_grad():
58             output = self.loaded_model(input).cpu()
59         return output
60
61     def load_model(self):
62         try:
63             # Initialize and load the model
64             model = CatDogModel(CatDogDataConfig.N_CLASSES)
65             model.load_state_dict(torch.load(self.model_weight,
map_location=self.device))
66             model.to(self.device)
67             model.eval()
68
69             self.loaded_model = model
70
71         except Exception as e:
72             LOGGER.log.error("Load model failed")
73             LOGGER.log.error(f"Error: {e}")
74             self.loaded_model = None
75
76     def create_transform(self):
77         # Define image transformations
78         self.transforms_ = torchvision.transforms.Compose([
79             torchvision.transforms.Resize((CatDogDataConfig.IMG_SIZE,
CatDogDataConfig.IMG_SIZE)),
80             torchvision.transforms.ToTensor(),
81             torchvision.transforms.Normalize(
82                 mean=CatDogDataConfig.NORMALIZE_MEAN,
83                 std=CatDogDataConfig.NORMALIZE_STD
84             ),
85         ])
86
87     def output2pred(self, output):
88         # Convert output to predictions
89         probabilities = F.softmax(output, dim=1)
90         best_prob = torch.max(probabilities, 1)[0].item()
91         predicted_id = torch.max(probabilities, 1)[1].item()
92         predicted_class = CatDogDataConfig.ID2DLABEL[predicted_id]
93
94         return probabilities.squeeze().tolist(), round(best_prob, 6),
predicted_id, predicted_class
95

```

f, Cập nhật folder middleware/

Middleware là một thành phần quan trọng API. Có thể hiểu nôm na Middleware là các hàm tiền và hậu xử lý các request mà API nhận được. Trong bài này, chúng ta sẽ triển khai như sau:

- `middleware/cors.py`: Ta cập nhật nội dung file này với `CORSMiddleware`. Hiểu một cách đơn giản, `CORSMiddleware` là một tính năng bảo mật cho phép ta định nghĩa một tập các tên miền, URL... (gọi chung là origin) thực hiện request và đọc được response trả về từ API.



Hình 1.8: Minh họa việc bị block khi đọc response từ một server khác tên miền do same-origin policy của các trình duyệt. Ảnh [Link](#)

Với demo này, ta sẽ cho phép tất cả origin được quyền request và đọc response.

Ta định nghĩa như sau:

```
1 from fastapi.middleware.cors import CORSMiddleware
2
3 def setup_cors(app):
4     app.add_middleware(
5         CORSMiddleware,
6         allow_origins=["*"], # Allow all origins
7         allow_credentials=True, # Allow cookies and other credentials
8         allow_methods=["*"], # Allow all HTTP methods
9         allow_headers=["*"], # Allow all HTTP headers
10    )
11
```

Hàm `setup_cors()` nhận đầu vào là một thực thể (instance) FastAPI. Về sau, khi đã khai báo một thực thể FastAPI, ta sẽ đưa vào hàm này để cập nhật policy này.

- `middleware/http.py`: Ta cập nhật nội dung file này với một định nghĩa

Middleware được custom cho việc logging:

```
1 import time
2 import sys
3 from pathlib import Path
4
5 # Add the parent directory to the Python path for imports
6 sys.path.append(str(Path(__file__).parent.parent))
7
8 from starlette.middleware.base import BaseHTTPMiddleware
9 from starlette.requests import Request
10 from utils.logger import Logger
11
12 # Initialize logger
13 LOGGER = Logger(__file__, log_file="http.log")
14
15 class LogMiddleware(BaseHTTPMiddleware):
16     """
17     Middleware for logging HTTP requests and responses with processing
18     time.
19     """
20     async def dispatch(self, request: Request, call_next):
21         """
22         Log details of the incoming HTTP request, response, and processing
23         time.
24
25         Args:
26             request (Request): Incoming HTTP request.
27             call_next: Function to process the next middleware or endpoint
28             .
29
30         Returns:
31             Response: HTTP response after processing.
32         """
33         # Start timing the request
34         start_time = time.time()
35
36         # Process the request and get the response
37         response = await call_next(request)
38
39         # Calculate processing time
40         process_time = time.time() - start_time
41
42         # Log the details
43         LOGGER.log.info(
44             f"{request.client.host} - \"{request.method} {request.url.path} "
45             f"{request.scope['http_version']}\" {response.status_code} "
46             f"{process_time:.2f}s"
47         )
48         return response
```


Ta sử dụng class logger đã khai báo trước đó, khi một request đã được thực thi xong, ta sẽ log lại thông tin của response này trước khi trả về cho người dùng. Như đã đề cập trước đó, việc sử dụng class `Middleware` này sẽ được thực hiện khi ta khai báo một thực thể FastAPI.

- `middleware/__init__.py`: Ta tạo file init để import sẵn các hàm, class của hai file trên để việc import ở các file trong vị trí folder khác thuận tiện hơn như sau:

```
1 from .http import LogMiddleware
2 from .cors import setup_cors
3
```

g, Cập nhật folder `schemas/`:

Các API thường biểu diễn dữ liệu request/response ở dạng bán cấu trúc như XML, JSON... Trong hầu hết tất cả các trường hợp, các trường thông tin trong dữ liệu sẽ được xác định trước. Vì vậy, sẽ thật tốt nếu chúng ta có một cách thức kiểm tra tự động mà không cần phải thao tác thủ công.



Hình 1.9: Minh họa dữ liệu request và dữ liệu response.

Trong FastAPI, chúng ta có thể sử dụng Pydantic Model, một dạng data validation. Pydantic Model cho phép chúng ta định nghĩa cấu trúc dữ liệu và đảm bảo rằng cấu trúc này sẽ luôn được tuân thủ bởi API. Đối với bài này, chúng ta sẽ cho người dùng gửi một tấm ảnh và API sẽ trả về kết quả dự đoán của tấm ảnh dưới dạng dictionary (JSON). Như vậy, ta sẽ định nghĩa pydantic model cho response này trong file `schemas/catdog_schema.py` như sau:

```
1 from pydantic import BaseModel
2
3 class CatDogResponse(BaseModel):
4     probs: list = []
```

```

5 | best_prob: float = -1.0
6 | predicted_id: int = -1
7 | predicted_class: str = ""
8 | predictor_name: str = ""

```

Việc tạo pydantic model gần như tương tự với việc tạo một class trong Python, điểm khác biệt là bạn cần phải kế thừa class BaseModel.

h, Cập nhật folder routes/:

Một API có thể sẽ có nhiều những chức năng khác nhau, từ việc truy vấn dữ liệu trong một database đến inference các Deep Learning model khác nhau. Vì vậy, để phân biệt rõ các chức năng trong một API, chúng ta cần định nghĩa các API endpoints.



Hình 1.10: Minh họa về API Endpoints. Với tên miền API là https://api_domain.com, chúng ta có thể có các endpoints với các chức năng khác nhau.

Trong demo này, chúng ta sẽ chỉ có một endpoint với chức năng inference model Cat Dog Classification. Vì vậy, nội dung folder này như sau:

- routes/catdog_route.py: Định nghĩa router (endpoint) cho phép nhận vào một tấm ảnh upload từ máy tính và trả về kết quả dự đoán của mô hình. Cấu trúc của response sẽ giống với pydantic model mà ta đã định nghĩa trước đó:

```

1 | import sys
2 | from pathlib import Path
3 | from fastapi import File, UploadFile, APIRouter
4 | from schemas.catdog_schema import CatDogResponse
5 | from config.catdog_cfg import ModelConfig
6 | from models.catdog_predictor import Predictor
7 |
8 | # Cập nhật đường dẫn
9 | sys.path.append(str(Path(__file__).parent))
10 |
11 | # Tạo router và predictor
12 | router = APIRouter()
13 | predictor = Predictor(
14 |     model_name=ModelConfig.MODEL_NAME,
15 |     model_weight=ModelConfig.MODEL_WEIGHT,
16 |     device=ModelConfig.DEVICE

```

```

17 )
18
19 @router.post("/predict")
20 async def predict(file_upload: UploadFile = File(...)):
21     """
22     ử dụng đoán ảnh loại Cat/Dog từ ảnh ảnh.
23
24     :param file_upload: File ảnh ảnh.
25     :return: kết quả đoán dưới dạng CatDogResponse.
26     """
27     response = await predictor.predict(file_upload.file)
28     return CatDogResponse(**response)
29

```

Trong đây, ta khởi tạo instance router của FastAPI để định nghĩa một endpoint, đồng thời khởi tạo một instance Predictor của model Cat Dog Classification để router này có thể gọi và lấy kết quả dự đoán.

- `routes/base.py`: Trong trường hợp các bạn có nhiều router, chúng ta có thể tạo một file `.py` để import tất cả router của từng file và gom lại thành một router duy nhất. Trong trường hợp này, mặc dù vậy, chúng ta sẽ chỉ import một router:

```

1 from fastapi import APIRouter
2 from .catdog_route import router as catdog_cls_route
3
4 # Tạo router chính và bao gồm các route từ catdog_cls_route
5 router = APIRouter()
6 router.include_router(catdog_cls_route, prefix="/catdog_classification")
7

```

i, Cập nhật `app.py` và `server.py`:

Cuối cùng, với các thành phần ở trên, ta sẽ khởi tạo một thực thể FastAPI để host API của chúng ta. Đầu tiên, ta code file `app.py` như sau:

```

1 import sys
2 from pathlib import Path
3
4 # Thêm thư mục hiện tại vào sys.path để hỗ trợ nhập các module
5 sys.path.append(str(Path(__file__).parent))
6
7 from fastapi import FastAPI
8 from middleware import LogMiddleware, setup_cors
9 from models.routes.base import router
10
11 # Tạo ứng dụng FastAPI
12 app = FastAPI()
13
14 # Thêm middleware ghi log
15 app.add_middleware(LogMiddleware)

```

```

16
17 # Cấu hình CORS
18 setup_cors(app)
19
20 # Thêm router chính
21 app.include_router(router)

```

Ta tạo instance FastAPI với tên biến `app`, sau đó cập nhật middleware cũng như router mà ta đã định nghĩa ở các file khác vào trong biến này. Cuối cùng, để triển khai API, ta sẽ cập nhật đoạn code này vào file `server.py`:

```

1 import uvicorn
2
3 if __name__ == "__main__":
4     uvicorn.run("app:app", host="0.0.0.0", port=8000, reload=True)

```

Theo đó, ta sử dụng thư viện `uvicorn` để host API trên địa chỉ `0.0.0.0` (địa chỉ máy local) trên port `8000`. Trong trường hợp các bạn bị lỗi port đã được sử dụng, hãy đổi sang port khác nhé.

j, Thực thi chương trình:

Để khởi động API, chúng ta hãy cài đặt các gói thư viện cần thiết trong file `requirements.txt` với lệnh sau trong terminal (các bạn nên cài trong môi trường `conda`, hướng dẫn cài đặt `conda` tại [đây](#)):

```

1 pip3 install -r requirements.txt
2
3 python3 server.py

```

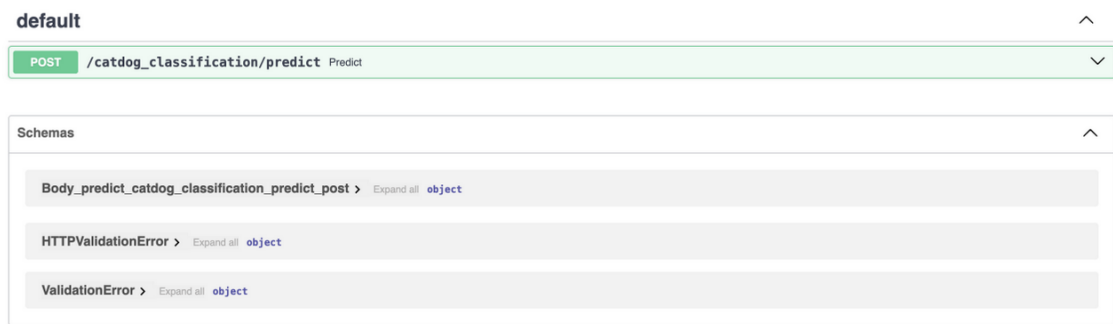
Theo đó, nếu cài đặt tương tự như ví dụ trong bài, các bạn có thể truy cập vào địa chỉ `http://0.0.0.0:8000/docs` trên trình duyệt và sẽ có kết quả như sau:

```

~/Desktop/deep-learning-book/dev/5_fastapi/root on main [+]
% python server.py
INFO: Will watch for changes in these directories: ['/home/hungmanh/Desktop/deep-learning-book/dev/5_fastapi/root']
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO: Started reloader process [266300] using WatchFiles
INFO: Started server process [266312]
INFO: Waiting for application startup.
INFO: Application startup complete.

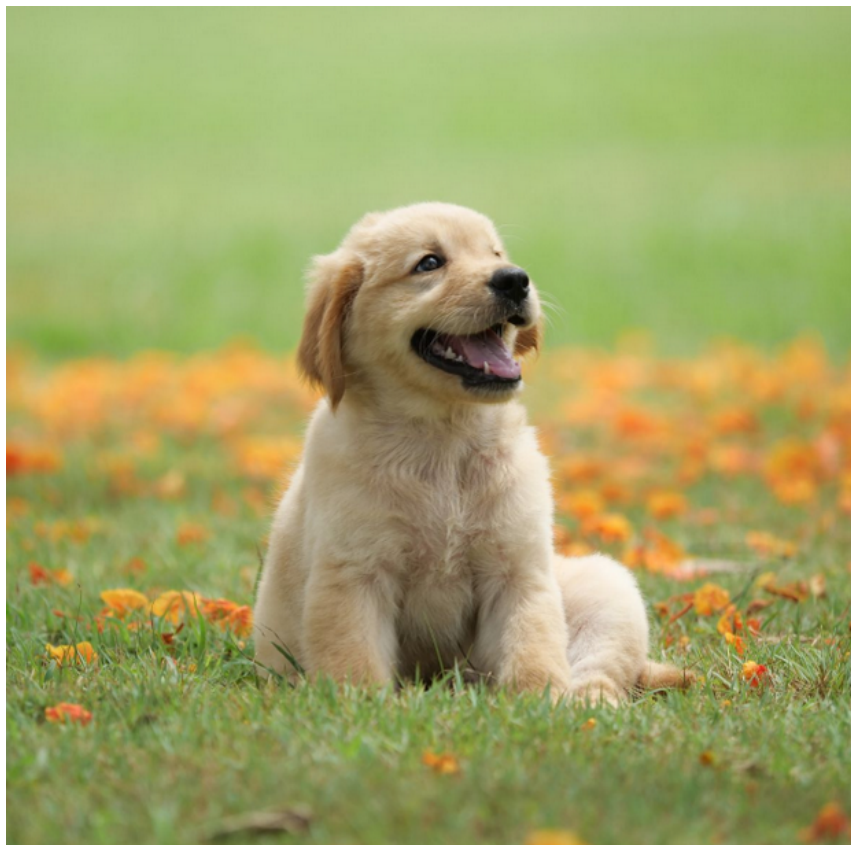
```

Hình 1.11: Nội dung trên màn hình terminal khi API đã được host thành công.

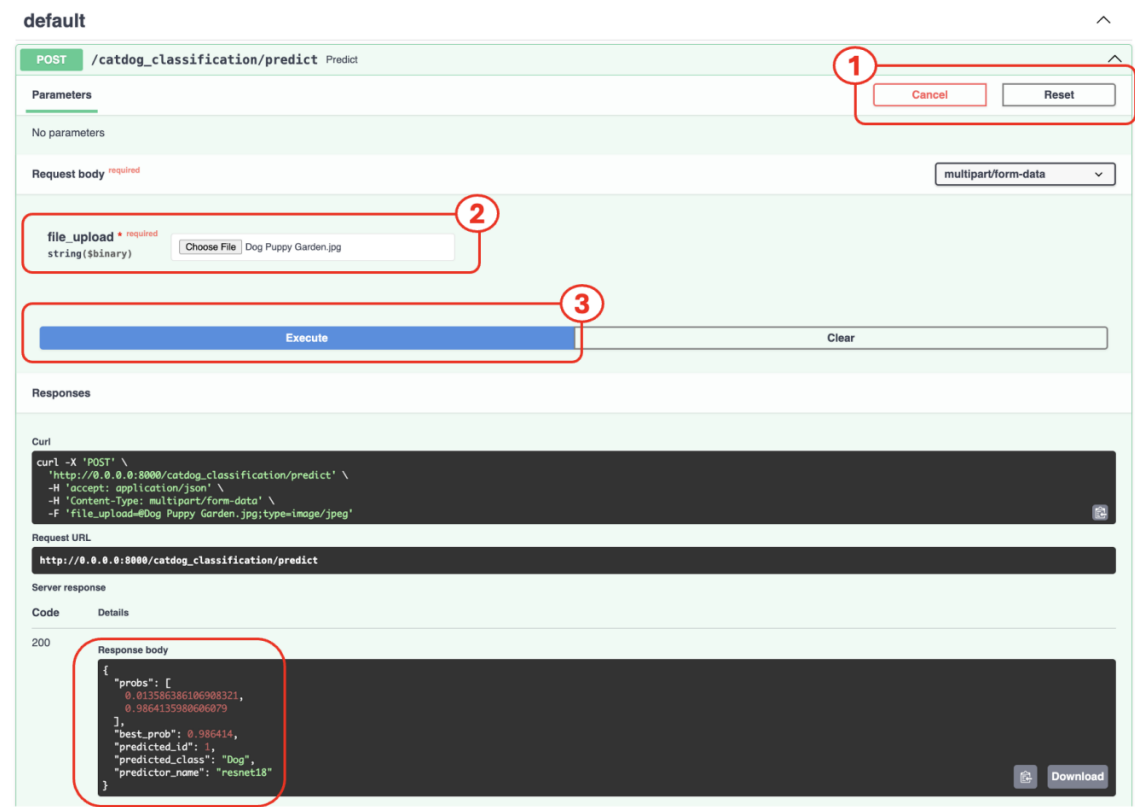


Hình 1.12: FastAPI docs. Giao diện được khởi tạo tự động cho phép tương tác với các chức năng của API.

Từ đây, các bạn có thể sử dụng mô hình của mình bằng cách upload ảnh lên và thực hiện request. Kết quả sẽ được trả về dưới dạng JSON:



Hình 1.13: Mẫu dữ liệu test. Ảnh [link](#)



Hình 1.14: Các bước sử dụng mô hình trên FastAPI docs.

Code	Details
200	<p>Response body</p> <pre>{ "probs": [0.013586386106908321, 0.9864135980606079], "best_prob": 0.986414, "predicted_id": 1, "predicted_class": "Dog", "predictor_name": "resnet18" }</pre>

Hình 1.15: Kết quả của mô hình, tương ứng với dữ liệu đầu vào là ví dụ mẫu.

Như vậy, chúng ta đã hoàn tất việc host một API với chức năng Cat Dog Classification. Các bạn giờ đây có thể tắt chương trình bằng cách nhấn tổ hợp phím Ctrl + C ở terminal.