Featured

Getting started

Hello, world

Simple web scraper

Serving web endpoints

Large language models (LLMs)

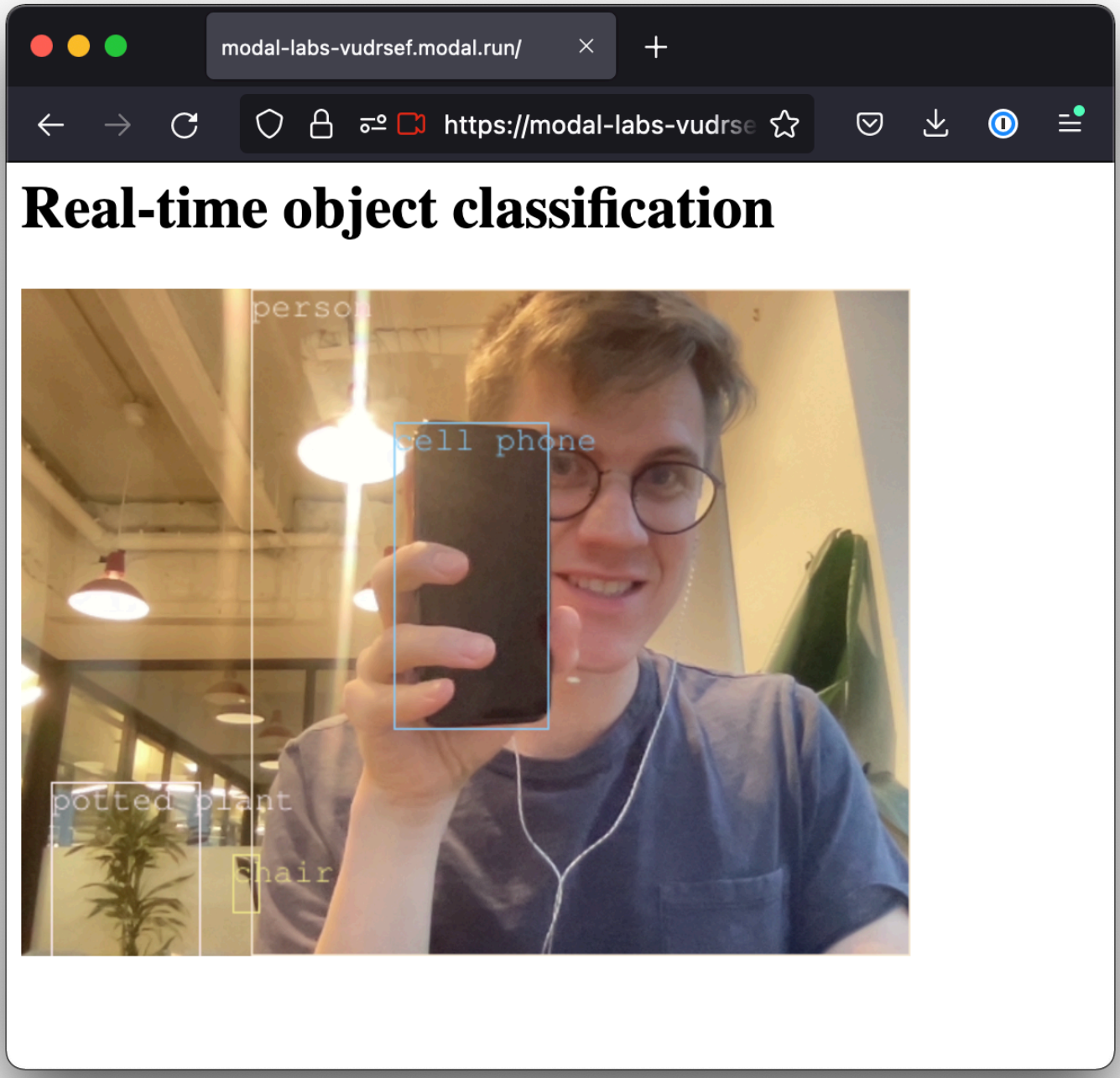
Machine learning model inference endpoint that uses the webcam

View on GitHub

This example creates a web endpoint that uses a Huggingface model for object detection.

The web endpoint takes an image from their webcam, and sends it to a Modal web endpoint. The Modal web endpoint in turn calls a Modal function that runs the actual model.

If you run this, it will look something like this:



Live demo

Take a look at the deployed app.

A couple of caveats:

- This is not optimized for latency: every prediction takes about 1s, and there's an additional overhead on the first prediction since the containers have to be started and the model initialized.
- This doesn't work on iPhone unfortunately due to some issues with HTML5 webcam components

Code

Starting with imports:

```
import base64
import io
from pathlib import Path

from fastapi import FastAPI, Request, Response
from fastapi.staticfiles import StaticFiles
from modal import App, Image, Mount, asgi_app, build, enter, method
```

We need to install transformers which is a package Huggingface uses for all their models, but also Pillow which lets us work with images from Python, and a system font for drawing.

This example uses the `facebook/detr-resnet-50` pre-trained model, which is downloaded once at image build time using the `@build` hook and saved into the image. 'Baking' models into the `modal.Image` at build time provided the fastest cold start.

```
model_repo_id = "facebook/detr-resnet-50"


app = App("example-webcam-object-detection")
image = (
    Image.debian_slim()
    .pip_install(
        "huggingface-hub==0.16.4",
        "Pillow",
        "timm",
        "transformers",
    )
    .apt_install("fonts-freefont-ttf")
)
```

Prediction function

The object detection function has a few different features worth mentioning:

- There's a container initialization step in the method decorated with `@enter()`, which runs on every container start. This lets us load the model only once per container, so that it's reused for subsequent function calls.
- Above we stored the model in the container image. This lets us download the model only when the image is (re)built, and not everytime the function is called.

- We're running it on multiple CPUs for extra performance

Note that the function takes an image and returns a new image. The input image is from the webcam The output image is an image with all the bounding boxes and labels on them, with an alpha channel so that most of the image is transparent so that the web interface can render it on top of the webcam view.

```
with image.imports():
    import torch
    from huggingface_hub import snapshot_download
    from PIL import Image, ImageColor, ImageDraw, ImageFont
    from transformers import DetrForObjectDetection, DetrImageProcessor


@app.cls(
    cpu=4,
    image=image,
)
class ObjectDetection:
    @build()
    def download_model(self):
        snapshot_download(repo_id=model_repo_id, cache_dir="/cache")

    @enter()
    def load_model(self):
        self.feature_extractor = DetrImageProcessor.from_pretrained(
            model_repo_id,
            cache_dir="/cache",
        )
        self.model = DetrForObjectDetection.from_pretrained(
            model_repo_id,
            cache_dir="/cache",
        )

    @method()
    def detect(self, img_data_in):
        # Based on https://huggingface.co/spaces/nateraw/detr-object-detection/blob/main/a
        # Read png from input
        image = Image.open(io.BytesIO(img_data_in)).convert("RGB")

        # Make prediction
        inputs = self.feature_extractor(image, return_tensors="pt")
        outputs = self.model(**inputs)
        img_size = torch.tensor([tuple(reversed(image.size))])
        processed_outputs = (
            self.feature_extractor.post_process_object_detection(
                outputs=outputs,
                target_sizes=img_size,
                threshold=0,
            )
```

```
        )
        output_dict = processed_outputs[0]

        # Grab boxes
        keep = output_dict["scores"] > 0.7
        boxes = output_dict["boxes"][keep].tolist()
        scores = output_dict["scores"][keep].tolist()
        labels = output_dict["labels"][keep].tolist()

        # Plot bounding boxes
        colors = list(ImageColor.colormap.values())
        font = ImageFont.truetype(
            "/usr/share/fonts/truetype/freefont/FreeMono.ttf", 18
        )
        output_image = Image.new("RGBA", (image.width, image.height))
        output_image_draw = ImageDraw.Draw(output_image)
        for _score, box, label in zip(scores, boxes, labels):
            color = colors[label % len(colors)]
            text = self.model.config.id2label[label]
            box = tuple(map(int, box))
            output_image_draw.rectangle(box, outline=color)
            output_image_draw.text(
                box[:2], text, font=font, fill=color, width=3
            )

        # Return PNG as bytes
        with io.BytesIO() as output_buf:
            output_image.save(output_buf, format="PNG")
            return output_buf.getvalue()
```

Defining the web interface

To keep things clean, we define the web endpoints separate from the prediction function. This will introduce a tiny bit of extra latency (every web request triggers a Modal function call which will call another Modal function) but in practice the overhead is much smaller than the overhead of running the prediction function etc.

We also serve a static html page which contains some tiny bit of Javascript to capture the webcam feed and send it to Modal.

```
web_app = FastAPI()
static_path = Path(__file__).with_name("webcam").resolve()
```

The endpoint for the prediction function takes an image as a data URI and returns another image, also as a data URI:

```
@web_app.post("/predict")
async def predict(request: Request):
    # Takes a webcam image as a datauri, returns a bounding box image as a datauri
    body = await request.body()
    img_data_in = base64.b64decode(body.split(b",")[1])  # read data-uri
    img_data_out = ObjectDetection().detect.remote(img_data_in)
    output_data = b"" + base64.b64encode(img_data_out)
    return Response(content=output_data)
```

Exposing the web server

Let's take the Fast API app and expose it to Modal.

```
@app.function(
    mounts=[Mount.from_local_dir(static_path, remote_path="/assets")],
)
@asgi_app()
def fastapi_app():
    web_app.mount("/", StaticFiles(directory="/assets", html=True))
    return web_app
```

Running this locally

You can run this as an ephemeral app, by running

```
modal serve webcam.py
```