MSENSIS MLE Task

Cats vs Dogs Image Classification

Foivos Ntoulas-Panagiotopoulos

November 21, 2025

1 Overview

This report describes my solution to the MSENSIS ML Engineering task. Given the limited time and lack of GPU, the focus was on a clean, reproducible pipeline rather than maximum accuracy.

2 Data and Preprocessing

The provided data consist of:

- a folder `dataset/images/` with JPEG images of cats and dogs,
- a CSV file `dataset/labels.csv` with:
  - `image_name`: image filename (e.g. `1234.jpg`),
  - `label`: class label (`Cat` or `Dog`, sometimes missing).

I used **Pandas** and **NumPy** for data handling. The main cleaning steps were:

1. Load `labels.csv`.
2. Drop rows with missing labels (`NaN`/`None`). An alternative would be to keep the unlabeled images, run k-NN in the pretrained model's feature space, and assign pseudo-labels only when the neighborhood is confident.
3. Normalize labels to lowercase and map them to integers (`cat` $\rightarrow$ 0, `dog` $\rightarrow$ 1).

Some images referenced in the CSV were missing or corrupted. To avoid runtime errors I added:

- `test.py`: checks that each `image_name` exists in `dataset/images/` and can be opened; prints a summary of missing/unreadable images.
- `clean_labels.py`: drops problematic rows, saves a backup, and overwrites `labels.csv` with a cleaned version.

After cleaning, the training set contains only valid *(image, label)* pairs and the loaders no longer crash.

3 Model and Training

Pretrained Model

I used the Vision Transformer model from Hugging Face:

- `google/vit-base-patch16-224-in21k`.

It is loaded via `ViTForImageClassification` and `AutoImageProcessor`. The original classification head (for ImageNet-21k) is replaced with a new linear layer with two outputs (cat/dog), reusing the pretrained feature extractor and only learning a small task-specific head.

Dataset and DataLoaders

A custom PyTorch `Dataset` (`CatsDogsDataset`):

- reads filenames and labels from `labels.csv`,
- applies the cleaning and label encoding described above,
- opens each image with PIL,
- uses the ViT image processor (resize, normalization) and
- returns `{pixel_values, labels}` for the Hugging Face model.

I use `torch.utils.data.random_split` to create an 80/20 train/validation split and wrap both subsets in `DataLoader` objects. Training is performed on CPU, so the batch size is kept relatively small (32) to avoid memory issues.

Training Procedure

Training is implemented in `train_custom_model()` in `main.py`. The setup is:

- optimizer: AdamW with learning rate 5×10^{-5},
- loss: cross-entropy (handled by `ViTForImageClassification` when labels are passed),
- metrics: running loss and accuracy for train and validation, shown with `tqdm` progress bars.

Due to compute limitations (CPU-only and a relatively large model), I trained for a small number of epochs (currently 1). The goal was to demonstrate the pipeline rather than fully converge the model.

After each epoch I evaluate on the validation set. If the validation accuracy improves, the model weights are saved to `models/cats_dogs_vit.pt`, which is later used for inference.

Pretrained checkpoint. Because of GitHub's file size limits, the fine-tuned weights file `cats_dogs_vit.pt` is not stored in the public repository. Instead, it can be shared separately. If this file is placed under `models/` (i.e. as `models/cats_dogs_vit.pt`), the inference code automatically loads it at startup. When the file is absent, the API falls back to the base pretrained ViT checkpoint and still runs correctly, albeit with lower accuracy.

4 API and Web UI

The serving layer is implemented with **FastAPI** in the same `main.py` file.

At startup, the script:

1. Loads the same ViT architecture.
2. If a fine-tuned checkpoint exists at `models/cats_dogs_vit.pt`, it is loaded; otherwise, the base pretrained weights are used.
3. Sets the model to evaluation mode.

The application exposes:

- **GET** `/health`: health check returning `{"status": "ok"}`.
- **GET** `/`: minimal HTML page with a file upload form for manual testing.

- **POST /predict**: accepts a JPEG/PNG file, validates the type, preprocesses the image, runs a forward pass and returns JSON with:
  - `predicted_label` $\in \{$`"cat"`, `"dog"`$\}$,
  - `confidence` $\in [0, 1]$.

This design supports both browser-based testing and programmatic use of the model.

5 How to Run

The repository includes a `README.md` with detailed steps. In summary:

1. Create a virtual environment and install dependencies with `pip install -r requirements.txt`.
2. Place the dataset under `dataset/images/` and `dataset/labels.csv`.
3. Run `python test.py` and `python clean_labels.py` to clean the dataset. It would be optional but there are some corrupt images that crash the dataloader.
4. Train the model with `python main.py -train` (this creates `models/cats_dogs_vit.pt`).
5. Alternatively, if the fine-tuned checkpoint `cats_dogs_vit.pt` has been provided separately, copy it to the `models/` folder and skip training.
6. Start the API/Web UI with `python main.py` and open `http://localhost:8000/`.