SPARROW DETECTION User Manual

Table of Contents

Introduction

Sparrow Detection is an end-to-end MLOps system for training, serving, and monitoring object detection models. The system is built with a focus on reproducibility, scalability, and monitoring to ensure reliable machine learning operations.

Key Components

- **DVC**: Data version control for dataset management
- **MLflow**: Experiment tracking and model registry
- **Docker**: Service orchestration and containerization
- **Prometheus**: System and model performance monitoring
- **Flask**: Backend API for model serving
- **Streamlit**: Interactive frontend UI
- **Grafana**: Visualization dashboards for metrics
- **Faster R-CNN**: Object detection model architecture

System Requirements

- **Operating System**: Linux (recommended), macOS, or Windows with WSL2
- **Python**: Version 3.8 or higher

- **Storage**: Minimum 10GB available for datasets and models
- **RAM**: Minimum 8GB (16GB+ recommended for training)
- **GPU**: Optional but recommended for training (CUDA compatible)
- **Docker**: Version 20.10.x or higher
- **Docker Compose**: Version 2.x or higher
- **Git**: For version control and DVC operations

Installation Guide

Step 1: Clone the Repository

```
git clone https://github.com/your-username/sparrow-detection.git
cd sparrow-detection
```

Step 2: Install Dependencies

For training, API, and UI components, install the required dependencies:

```
# Create a virtual environment (recommended)
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install core training dependencies
pip install -r requirements.txt

# Install API dependencies
pip install -r api_requirements.txt

# Install UI dependencies
pip install -r app_requirements.txt
```

Step 3: Set Up DVC

Initialize DVC and pull existing datasets:

```
# Initialize DVC (if not done already)
dvc init

# Configure DVC remote (if not configured)
dvc remote add -d myremote s3://your-bucket/path  # Example with S3
# OR
dvc remote add -d myremote /path/to/shared/storage  # Local storage
```

```
# Pull existing data
dvc pull
```

Step 4: Start Services with Docker

To run all services together using Docker Compose:

```
docker-compose up --build
```

This will start the API server, Streamlit frontend, Prometheus, and Grafana services.

Data Management with DVC

Data Version Control (DVC) is used to track and version datasets without storing them in Git.

Accessing Existing Data

To pull existing datasets from the remote storage:

```
dvc pull
```

This will download all datasets defined in DVC files to your local machine.

Adding New Data

To add a new dataset:

1. Place your dataset in the `data/` directory
2. Track it with DVC:

```
dvc add data/your_new_dataset
git add data/.gitignore data/your_new_dataset.dvc
git commit -m "Added new dataset for sparrow detection"
dvc push
```

Switching Between Dataset Versions

To switch to a specific version of the data:

```
git checkout <commit-hash-or-tag>
dvc checkout
```

Model Training

Configuring Training Parameters

Before training, configure the hyperparameters in the `params.yaml` file:

```yaml
# Example params.yaml
model:
  backbone: 'resnet50'
  pretrained: true
  num_classes: 2  # Background + Sparrow

training:
  batch_size: 8
  learning_rate: 0.001
  epochs: 50
  optimizer: 'adam'
  momentum: 0.9
  weight_decay: 0.0005
  early_stopping_patience: 5
```

Running Training

To train the model with the configured parameters:

```
python src/train.py
```

The training script will:

1. Load data according to DVC configuration
2. Initialize the Faster R-CNN model
3. Train using the specified hyperparameters
4. Log metrics to MLflow
5. Save the best model

Monitoring Training Progress

Training logs are saved to `train.log`. You can monitor the progress with:

```
tail -f train.log
```

Additionally, you can view training metrics in the MLflow UI:

```
mlflow ui
```

Then open your browser at http://localhost:5000 to see experiment results.

Using the API

The API provides endpoints for model inference and monitoring.

Running the API Locally

To start the API server locally:

```
python api/main.py
```

The API will be available at http://localhost:5000

API Endpoints

Endpoint	Method	Description	Request Format
`/predict`	POST	Upload an image and get detections	Form data with 'image' field
`/metrics`	GET	Expose Prometheus metrics	None

Example API Usage

Using curl to send an image for prediction:

```
curl -X POST -F "image=@/path/to/your/image.jpg" http://localhost:5000/predict
```

Sample response:

```
{
  "detections": [
    {
      "bbox": [120, 45, 280, 215],
      "class": "sparrow",
      "confidence": 0.92
    }
  ],
  "inference_time": 0.153,
  "success": true
}
```

Working with the UI

The Streamlit UI provides a user-friendly interface for interacting with the model.

Accessing the UI

After starting the services with Docker Compose, access the UI at:

```
http://localhost:8501
```

UI Features

1. **Upload Images**: Drag and drop or upload images for detection
2. **View Results**: See detected objects with bounding boxes
3. **Confidence Threshold**: Adjust detection sensitivity
4. **Download Results**: Save detection results as JSON or CSV
5. **View Wrong Predictions**: Access images that were incorrectly classified

Wrong Predictions

The system saves images with potentially incorrect predictions to: `frontend/wrong_predictions/`

Review these images periodically to identify model weaknesses and collect data for future model improvements.

Monitoring with Prometheus and Grafana

Prometheus Metrics

Access the Prometheus dashboard at:

```
http://localhost:9090
```

Key metrics available:

- `model_inference_time` : Time taken for each prediction
- `model_request_count` : Number of requests processed
- `model_error_rate` : Percentage of failed requests
- `system_memory_usage` : Memory consumption of the API service
- `system_cpu_usage` : CPU utilization

Grafana Dashboards

Access Grafana at:

```
http://localhost:3000
```

Default login credentials:

- Username: admin
- Password: admin

Pre-configured dashboards include:

1. **Model Performance**: Inference time, throughput, and error rates
2. **System Resources**: CPU, memory, and disk usage
3. **Request Traffic**: Request volume and response codes

Advanced: Adding New Models

Preparing a New Dataset

1. Organize your dataset with the following structure:

```
data/
├── your_dataset/
│   ├── images/
│   │   ├── image1.jpg
│   │   ├── image2.jpg
│   │   └── ...
│   └── annotations/
│       ├── image1.xml  # Pascal VOC format
│       ├── image2.xml
│       └── ...
```

2. Track with DVC:

```
dvc add data/your_dataset
git add data/.gitignore data/your_dataset.dvc
git commit -m "Added new sparrow dataset"
dvc push
```

Training a New Model

1. Update `params.yaml` with appropriate hyperparameters

2. Run training:

```
python src/train.py --dataset your_dataset
```

Deploying the New Model

1. The best model will be registered in MLflow
2. Update the model version in the API:

```
# Option 1: Edit the API config
nano api/config.yaml  # Set model_version to the new version

# Option 2: Set environment variable
export MODEL_VERSION=your_new_version
```

3. Restart the API:

```
docker-compose restart api
```

Troubleshooting

Common Issues

Data not found after `dvc pull`

- Verify DVC remote configuration: `dvc remote list`
- Check connectivity to remote storage
- Ensure you have permissions to access the remote storage

Training fails to start

- Check if CUDA is available: `python -c "import torch; print(torch.cuda.is_available())"`
- Verify dataset paths in DVC configuration
- Check for sufficient disk space: `df -h`

API returns errors

- Check API logs: `docker-compose logs api`
- Verify the model path is correct
- Test API endpoints with simple requests

UI not displaying correctly

- Check browser console for JavaScript errors
- Verify Streamlit is running: `docker-compose ps`
- Try clearing browser cache or using incognito mode

Logging Locations

- Training logs: `train.log`
- API logs: `docker-compose logs api`
- UI logs: `docker-compose logs frontend`
- Prometheus logs: `docker-compose logs prometheus`
- Grafana logs: `docker-compose logs grafana`

Best Practices

1. **Version Control**: Always commit DVC files and configuration changes to Git
2. **Experiment Tracking**: Use MLflow to track all training runs
3. **Resource Management**:
   - Stop unnecessary services when not in use: `docker-compose stop <service-name>`
   - Remove unused images: `docker image prune`
4. **Security**:
   - Don't expose the API directly to the internet without proper authentication
   - Rotate Grafana and other service credentials regularly
5. **Backup**:
   - Regularly backup MLflow database
   - Ensure DVC remote storage has redundancy t