MLOps for MNIST Classification: From Data to Deployment with Open Source Tools - A Tutorial

Ganapathy Krishnamurthi, Sidharth Ramesh, Gowthamaan Palani

Medical Imaging and Reconstruction Laboratory

Indian Institute of Technology

Target Audience:

Data Scientists, Machine Learning Engineers, DevOps Engineers, and anyone interested in learning practical MLOps.

Note: Basic Python and Machine Learning knowledge (especially PyTorch) is assumed.

Overall Goal:

To guide learners through the complete MLOps lifecycle for a simple MNIST classification model, demonstrating key MLOps principles and practices using free and open-source tools.

Tools to be Used (All Open Source and Free):

- **Programming Language:** Python
- **ML Framework:** PyTorch
- **Data Management & Versioning:** DVC (Data Version Control)
- **Experiment Tracking & Model Registry:** MLflow
- **Code Versioning:** Git & GitHub
- **Containerization:** Docker
- **Orchestration & Automation:** Make (for local), GitHub Actions (for CI/CD)
- **Deployment:** Flask (for API), Docker Hub / Github Packages (for image registry)
- **Monitoring:** Basic logging (Python's `logging` module), Simple terminal-based monitoring scripts (e.g., `watch`, `curl`)
- **Infrastructure (Local):** Your own machine (for training and local deployment)
- **Infrastructure (Cloud - Optional & Free Tier Focused):** Cloud providers like Google Cloud, AWS, Azure for potential scaling. They also offere free tier versions (primary focus on local for simplicity)

Tutorial Structure & Modules:

Module 1: Introduction to MLOps and Setting Up the Environment

- **1.1 What is MLOps?**
  - Define MLOps and its importance in the ML lifecycle.
  - Explain the key principles: Automation, CI/CD, Monitoring, Versioning, Collaboration.
  - Briefly compare MLOps to DevOps.
- **1.2 Tutorial Overview & Objectives:**
  - Outline the MNIST classification problem.
  - Explain the tools we will be using and why they are chosen.
- **1.3 Setting Up the Development Environment:**
  - **Software Installation:**
    - Python
    - Git
    - Docker Desktop (or Docker Engine + Docker Compose if preferred)
    - DVC (Data Version Control)
    - MLflow
    - PyTorch
    - Flask
  - **Project Structure:**
    - Create a well-organized project directory structure (e.g., `mnist-mlops-tutorial`):

```
mnist-mlops-tutorial/
├── data/
├── checkpoints/
├── src/
│   ├── data/
│   │   ├── download_mnist.py
│   │   ├── preprocess_data.py
│   ├── models/
│   │   ├── model.py
│   │   ├── train_model.py
│   │   ├── evaluate_model.py
│   ├── api/
│   │   ├── app.py
│   ├── utils/
│   │   ├── logger.py
│   ├── tests/
```

```
│   │   ├── test_model.py
├── notebooks/ (for exploration, optional)
├── Dockerfile
├── docker-compose.yml
├── Makefile
├── requirements.txt
├── .gitignore
└── README.md
```

- **Environment Setup:**
  - Create a virtual environment using `conda` or `venv`.
  - Install required Python packages using `requirements.txt`.
  - Initialize Git repository (`git init`).
  - Initialize DVC (`dvc init`).

Module 2: Data Engineering and Versioning (DataOps)

- **2.1 Data Acquisition (download_mnist.py):**
  - Use `torchvision.datasets.MNIST` to download the MNIST dataset.
  - Script to download and save raw MNIST data to `data/raw/`.
- **2.2 Data Preprocessing (preprocess_data.py):**
  - Implement data preprocessing steps:
    - Normalize pixel values (0-1).
    - Split data into training and validation sets.
    - Save preprocessed data to `data/processed/` (skipped in this tutorial).
- **2.3 Data Versioning with DVC:**
  - Track `data/raw/` and `data/processed/` directories with DVC (`dvc add data/raw/`, `dvc add data/processed/`).
  - Commit DVC changes to Git (`git commit -m "Track data with DVC"`).
  - Explain the benefits of data versioning for reproducibility and collaboration.
  - Demonstrate how to check out different data versions using DVC.

Module 3: Model Development and Experiment Tracking

- **3.1 Model Definition (models/model.py):**
  - Define a simple Convolutional Neural Network (CNN) for MNIST classification in PyTorch (e.g., using `torch.nn.Module`).
- **3.2 Training Script (models/train_model.py):**
  - Implement a training script using PyTorch:

        - Load preprocessed training data.
        - Instantiate the model.
        - Define loss function (CrossEntropyLoss) and optimizer (Adam).
        - Train the model for a few epochs.
        - Log training metrics (loss, accuracy) using MLflow:

          ```
          import mlflow

          with mlflow.start_run():
              # ... training loop ...
              mlflow.log_metric("epoch", epoch)
              mlflow.log_metric("loss", loss.item())
              mlflow.log_metric("accuracy", accuracy)
              # ...
          ```

        - Save the trained model weights to `checkpoints/trained_model.pth`.
- **3.3 Evaluation Script (models/evaluate_model.py):**
    - Implement an evaluation script:
        - Load preprocessed validation data.
        - Load the trained model.
        - Evaluate the model on the validation set.
        - Log evaluation metrics (validation loss, validation accuracy) using MLflow.
- **3.4 Experiment Tracking with MLflow:**
    - Run training script multiple times with different hyperparameters (e.g., learning rate, number of layers).
    - Use MLflow UI (`mlflow ui`) to:
        - Track experiments and runs.
        - Compare different runs based on metrics and parameters.
        - Visualize metrics over epochs.
    - Explain the benefits of experiment tracking for model improvement and reproducibility.

Module 4: Model Versioning and Testing

- **4.1 Model Versioning with DVC:**
    - Track the trained model file (`checkpoints/trained_model.pth`) with DVC (`dvc add checkpoints/trained_model.pth`).
    - Commit DVC changes to Git (`git commit -m "Track trained model with DVC"`).

    - Explain how DVC versioning works for models and how it's linked to code and data versions.
    - Demonstrate how to retrieve specific model versions.
- **4.2 Model Testing (tests/test_model.py):**
    - Write unit tests using Python's `unittest` or `pytest` (using `unittest` for simplicity in tutorial):
        - Test model input and output shapes.
        - Test model prediction on a sample input.
        - Test model loading and saving.
    - Run tests and ensure they pass.
    - Integrate tests into the CI/CD pipeline in later modules.

Module 5: Model Deployment as an API

- **5.1 Flask API Implementation (api/app.py):**
    - Create a Flask application to serve the MNIST classification model as an API:
        - Load the trained model weights.
        - Define an API endpoint (`/predict`) that:
            - Receives an image (e.g., as base64 encoded string).
            - Preprocesses the image to match model input.
            - Makes a prediction using the loaded model.
            - Returns the predicted class (digit).
        - Use Python's `logging` module to log API requests and predictions.
- **5.2 Dockerization (Dockerfile & docker-compose.yml):**
    - Create a `Dockerfile` to containerize the Flask API application:
        - Base image (e.g., `python:3.9-slim-buster`).
        - Install dependencies from `requirements.txt` .
        - Copy application code.
        - Expose port for the API (e.g., port 5000).
        - Define the entry point to run the Flask app (`python api/app.py`).
    - Create a `docker-compose.yml` file for easier local development and deployment:
        - Define services for the API application.
        - Potentially include other services (e.g., MLflow UI if needed for local demo).
- **5.3 Local Deployment and Testing:**
    - Build the Docker image (`docker build -t mnist-api .`) or (`docker build -f Dockerfile.prod -t mnist-api .`).

    - Run the Docker container using `docker run -p 5000:5000 mnist-api`.
    - Test the API using `curl` or a web browser (send a sample MNIST image to the `/predict` endpoint).
    - Verify API logs.

Module 6: Monitoring and Basic Automation

- **6.1 Basic Monitoring:**
    - **API Logging:** Review logs generated by the Flask API application (using Python's `logging` module).
    - **Terminal-based Monitoring:** Use command-line tools like `watch` and `curl` to:
        - Periodically send requests to the API endpoint.
        - Monitor API response times and error rates.
        - (Optional) Write a simple Python script to automate API monitoring and report metrics.
- **6.2 Local Automation with Make (Makefile):**
    - Create a `Makefile` to automate common MLOps tasks:
        - `make data`: Run data download and preprocessing scripts.
        - `make train`: Run model training script.
        - `make evaluate`: Run model evaluation script.
        - `make test`: Run unit tests.
        - `make deploy`: Build and run the Docker container.
        - `make monitor`: Run basic monitoring scripts.
    - Explain how `make` simplifies repetitive tasks and improves workflow efficiency.

Module 7: Continuous Integration and Continuous Deployment (CI/CD) with GitHub Actions

- **7.1 Setting up GitHub Repository:**
    - Push the project code to a GitHub repository.
- **7.2 CI/CD Pipeline with GitHub Actions:**
    - Create GitHub Actions workflows (`.github/workflows/`) for:
        - **Continuous Integration (CI):**
            - Triggered on code push or pull request.
            - Checkout code.
            - Set up Python environment.
            - Install dependencies.

            - Run unit tests (`make test`).
            - Build Docker image (`docker build -t mnist-api .`). (Optional - for CI, build might be enough, push to registry for CD)
        - **Continuous Deployment (CD):** (Simplified for tutorial - focusing on automated build & push to Docker Hub)
            - Triggered on tag creation (e.g., `git tag v1.0.0 && git push --tags`).
            - Checkout code.
            - Set up Python environment.
            - Install dependencies.
            - Run unit tests (optional - could be part of CI).
            - Build Docker image and tag it with the release version.
            - Log in to Docker Hub.
            - Push Docker image to Docker Hub (`docker push <your-dockerhub-username>/mnist-api:v1.0.0`).
    - Configure GitHub Secrets for Docker Hub credentials if needed for pushing images.
    - Explain the CI/CD pipeline stages and benefits for automated testing and deployment.
    - Trigger workflows by pushing code changes and creating tags.
    - Monitor workflow runs in GitHub Actions.

Module 8: Conclusion and Further Exploration

- **8.1 Recap of MLOps Principles and Practices Covered:**
    - Summarize the key MLOps concepts learned throughout the tutorial.
    - Reiterate the benefits of using MLOps for ML projects.
- **8.2 Further Exploration and Next Steps:**
    - Suggest advanced MLOps topics for further learning:
        - Model retraining and drift detection.
        - Advanced monitoring tools (Prometheus, Grafana, ELK stack).
        - Cloud deployment platforms (Kubernetes, AWS SageMaker, Google AI Platform).
        - Feature stores.
        - Model explainability and fairness.
        - Security in MLOps.
    - Encourage learners to apply MLOps principles to their own ML projects.

- **8.3 Resources and References:**
  - Provide links to documentation for all the tools used (PyTorch, DVC, MLflow, Docker, GitHub Actions, Flask).
  - Recommend MLOps books, articles, and online courses.

Optional Enhancements

- **Cloud Deployment (Free Tier):** Extend the tutorial to deploy the Docker image to a free tier cloud platform (e.g., Google Cloud Run, AWS ECS Fargate using free tier credits).
- **Advanced Monitoring (Prometheus/Grafana):** Integrate Prometheus and Grafana for more robust monitoring of the deployed API service.
- **Model Registry in MLflow:** Use MLflow's Model Registry to manage model versions and stages (Staging, Production).
- **A/B Testing:** A/B testing concepts and how MLOps can facilitate it.