Building a Robust MLOps Pipeline

Course: ID5003W
Submitted by: Rohit R Nath
Roll No.: CH24M561
Academic Year: 2025

ABSTRACT

This project delivers an end-to-end MLOps pipeline for MNIST digit classification. It uses Spark for scalable preprocessing, BigDL-Orca for distributed PyTorch training (CPU/GPU), MLflow for experiment tracking and model management, DVC for data versioning, and FastAPI for online serving. The pipeline supports drift detection (PSI), automated retraining, and deployment via Docker with Linux GPU support. We provide a browser-based UI to draw digits and query the API. The report details system design, implementation choices, operations, and provides placeholders for results and diagrams.

Keywords: *MLOps, Spark, bigdl-orca, MLFlow, DVC, Pytorch.*

I. INTRODUCTION

In the modern era of data science, building a machine learning model with high accuracy is only the first step. The true challenge lies in deploying, scaling, managing, and maintaining these models in production environments where data evolves and business needs change. This operational discipline, known as MLOps (Machine Learning Operations), is critical for creating real-world value from AI.

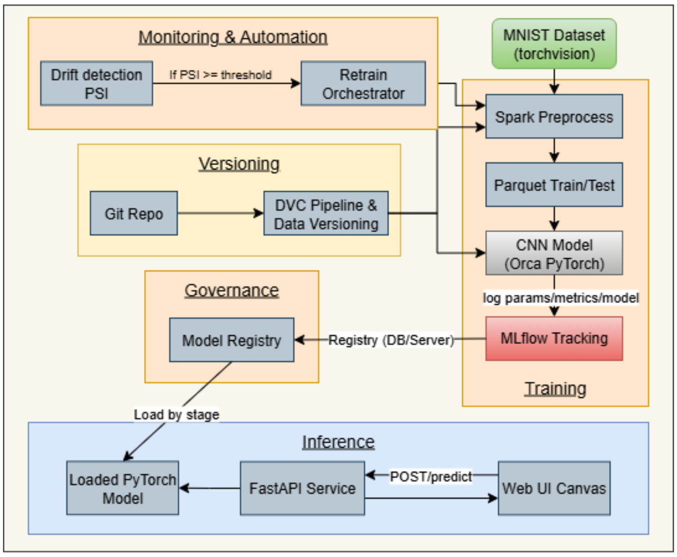
The objective of this project is to implement a robust pipeline that preprocesses data at scale, trains with distributed compute, tracks and registers models, deploys a REST API, detects drift, and supports automated retraining.

All the code used for the analysis and modeling are open and available here: https://github.com/rohitrnath/MNIST-MLOps-Pipeline

II. PROBLEM STATEMENT

The chosen task is the classification of handwritten digits from the MNIST dataset. While MNIST is academically simple, it provides an ideal testbed for demonstrating an end-to-end machine learning operations pipeline. The scope of this work includes every critical stage of the ML lifecycle: data preprocessing, distributed training, experiment tracking, model versioning and governance, online serving, monitoring, drift detection, and automated retraining. The system is designed to operate in Linux GPU environments and is compatible both with local development setups and with remote MLflow registry deployments.

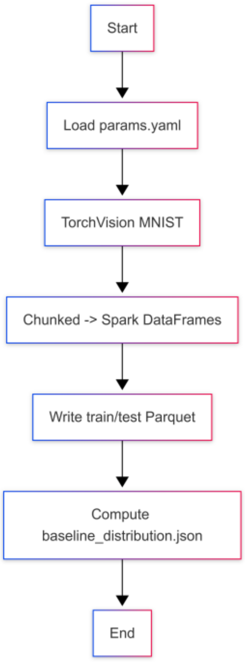
III. SYSTEM ATCHITECTURE



The pipeline brings together several specialized components in a cohesive manner. There are mainly 4 components in this end to end MLOps pipeline.

. Preprocessing

Spark is employed for preprocessing; it uses the torchvision based MNIST dataset and generating Parquet-based datasets and establishing baseline label distributions.



Flow Diagram: Preprocessing

A. Training

Training is carried out using BigDL-Orca, which distributes PyTorch's CNN model training while seamlessly logging experiments to MLflow. MLflow serves both as the experiment tracker and as the model registry, where available.

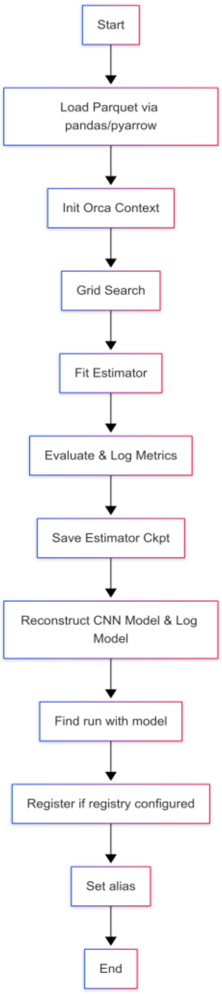
For reproducibility, DVC manages the data pipeline, ensuring lineage is preserved and reruns remain consistent.

B. Inference Service

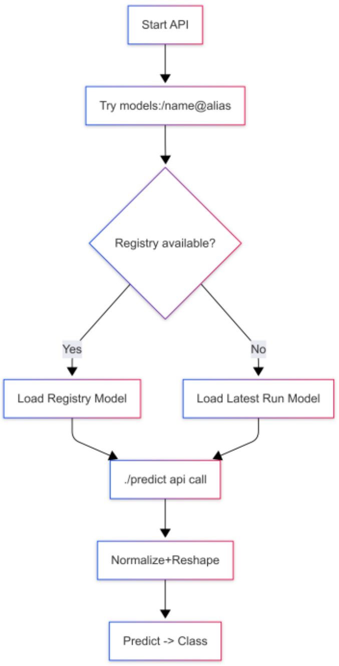
Once trained, models are deployed through a FastAPI-based inference service, which provides an API endpoint and a web-based interface for digit input.

C. Monitoring and Automation

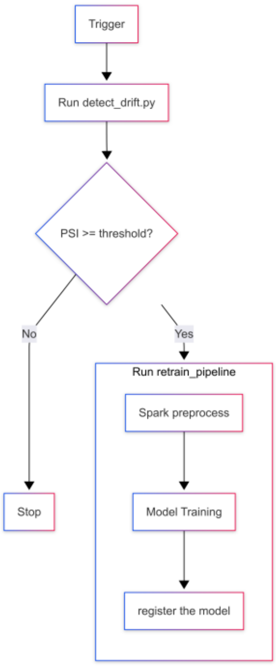
Drift detection scripts analyse incoming data distributions against the baseline, triggering retraining workflows when significant shifts are detected.



Flow Diagram: Training



Flow Diagram: Inference Service



Flow Diagram: Monitoring and Automation

IV. TOOLS AND RESOURCES

1. **BigDL-Orca 2.5.0** orchestrates distributed training on top of **PySpark 3.4.x** (local mode)
2. **PyTorch** + **Torchvision** (CUDA-enabled wheels for GPU) accelerates model learning.
3. **MLflow 2.13.0** handles tracking and model management.
4. **DVC** for reproducible data versioning.
5. Online serving is enabled by **FastAPI** + **Uvicorn**
6. **Docker** providing a GPU-capable deployment environment.
7. **Python 3.10 (Linux)**; **OpenJDK 11** supporting Spark execution
8. **WebUI:** HTML5 (HTML, CSS, Javascript)

V. DATA PIPELINE & VERSIONING

A. Preprocessing (Spark)

Data preprocessing begins with downloading the MNIST dataset via TorchVision. Spark transforms the raw data into normalized tensors and saves them as Parquet files. This approach provides efficient, scalable access for downstream consumers.

The spark configuration used is:

Spark Configuration	Value
Spark partition	8
Spark driver memory	8gb
Chunk size	2048

1. **Script:** src/preprocess_mnist_spark.py
2. **Input:** TorchVision MNIST
3. **Data Distribution**
    a. **Total Images:** 70000
    b. **Training Split:** 60000
    c. **Test Split:** 10000
4. **Transforms:** ToTensor + Normalize (mean=0.1307, std=0.3081)
5. **Output:** data/mnist/train.parquet, data/mnist/test.parquet, and baseline_distribution.json
6. **Performance:** Chunked ingestion to avoid large task serialization; configurable spark_driver_memory, spark_partitions, and chunk_size.
7. **Design rationale:** Parquet enables scalable downstream consumption; baseline distribution supports drift detection.

B. Data Versioning

The baseline distribution of labels is computed and stored to facilitate future drift detection. To ensure reproducibility, the preprocessing stage is defined in dvc.yaml, with DVC tracking lineage and outputs. This combination ensures that any dataset can be regenerated exactly, linking it back to code and parameters at the time of creation.

1. dvc.yaml defines preprocess stage; .dvcignore configured to not block outputs.
2. DVC captures data lineage and enables reproducible reruns.

VI. MODEL DESIGN

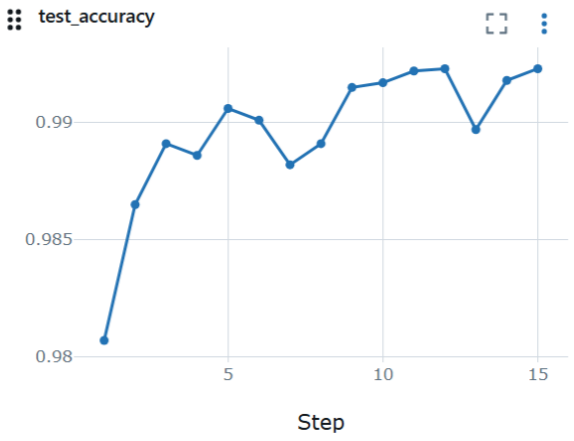
The model is designed with a step-by-step iterative approach with consistent progress in accuracy. I took a core model LeNet, which is extremely heavy for this application. Then designed a light-weight alternative with iterative approach which having 1/30th of the size of core model with better accuracy than LeNet.

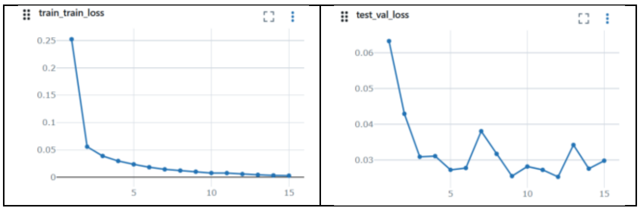
A. Core Model – LeNet

The core model is a LeNet variant optimized for digit recognition. The basic framework including training and testing loop developed with this model.

Results

1. **Parameters**: 4,40,812
2. **Best Test Accuracy:** 99.22
3. **Best Training Loss:** 0.0028
4. **Best Test Loss:** 0.0297





Analysis

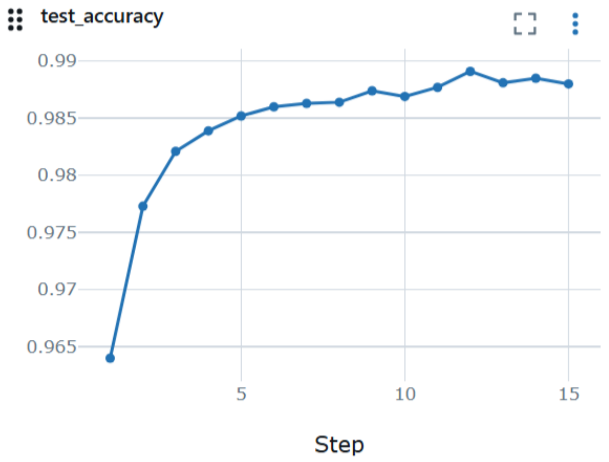
1. Extremely Heavy Model for such a problem.
2. The model is over-fitting, but we are changing our model in the next step.

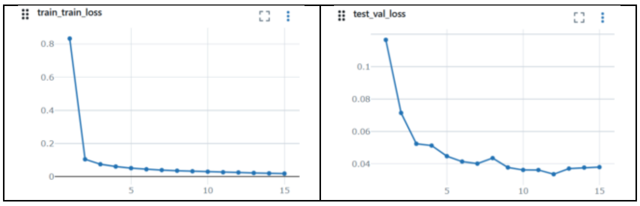
B. The Skelton Model

Get the basic skeleton right. We will try and avoid changing this skeleton as much as possible. Made this model 1/2nd of LE-Net without any complex operators.

Results

1. **Parameters**: 194,884
2. **Best Test Accuracy:** 98.79
3. **Best Training Loss:** 0.004
4. **Best Test Loss:** 0.0379





Analysis

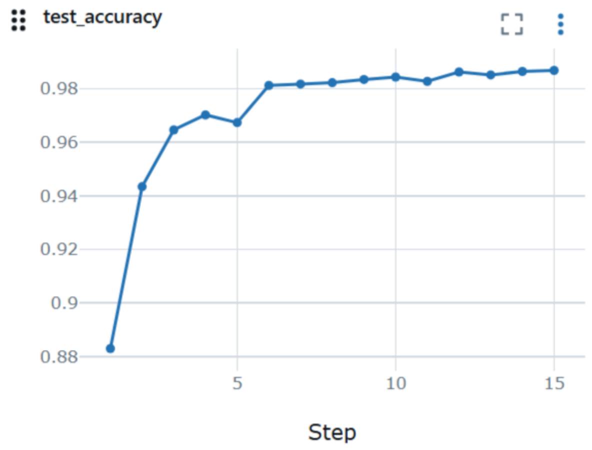
1. The model is still large, but working.
2. We see some over-fitting

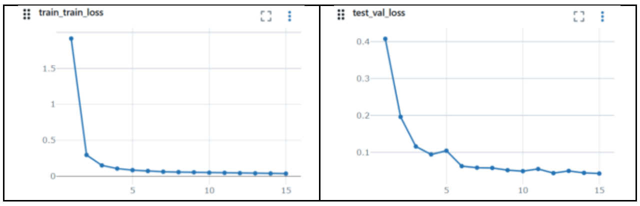
C. The Lighter Model

Make the model lighter.

Results

1. **Parameters**: 10,790
2. **Best Test Accuracy:** 98.68
3. **Best Training Loss:** 0.0122
4. **Best Test Loss:** 0.0428





Analysis

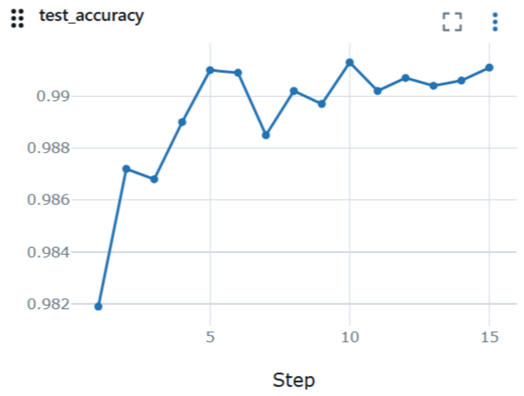
1. Good Model.
2. No over-fitting.
3. Model is capable if pushed further

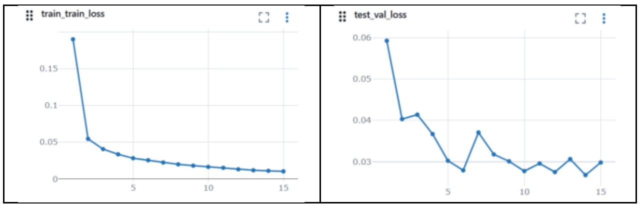
D. The Batch Normalization

Add Batch-norm to increase model efficiency.

Results

1. **Parameters**: 10,970
2. **Best Test Accuracy:** 99.12
3. **Best Training Loss:** 0.0098
4. **Best Test Loss:** 0.0298





Analysis

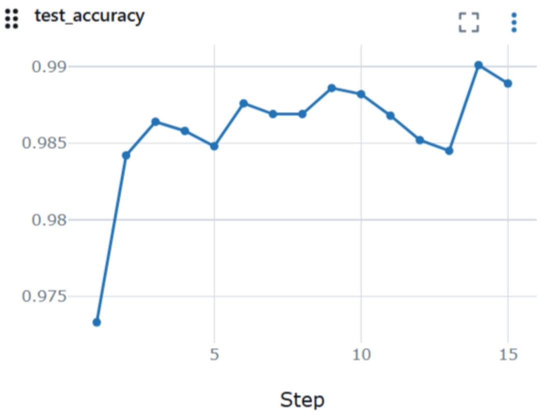
1. We have started to see over-fitting now.
2. Even if the model is pushed further, it won't be able to get to 99.2

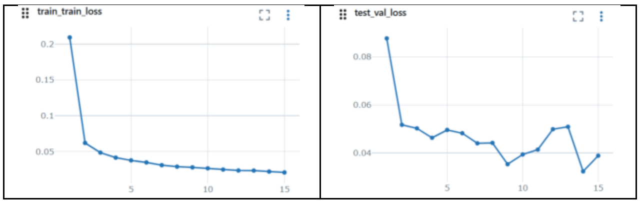
E. The Dropout

Add Dropout and analyse the results.

Results

1. **Parameters**: 10,970
2. **Best Test Accuracy:** 98.89
3. **Best Training Loss:** 0.0087
4. **Best Test Loss:** 0.0388





Analysis

1. Dropout working.
2. But with the current capacity, not possible to push it further.
3. We are not using GAP, but depending on a BIG-sized kernel.

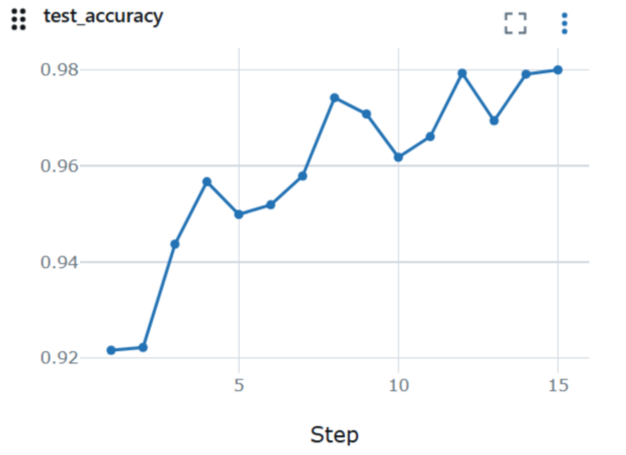
F. The Global Average Pooling

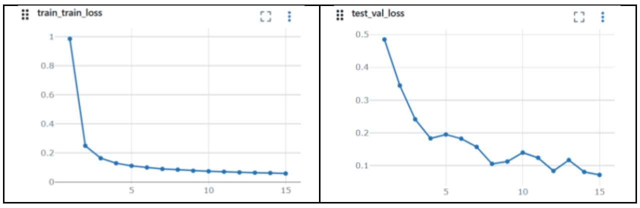
Add GAP and remove the last BIG kernel.

Results

1. **Parameters**: 6,070

2. **Best Test Accuracy:** 98.00
3. **Best Training Loss:** 0.0438
4. **Best Test Loss:** 0.0715





Analysis

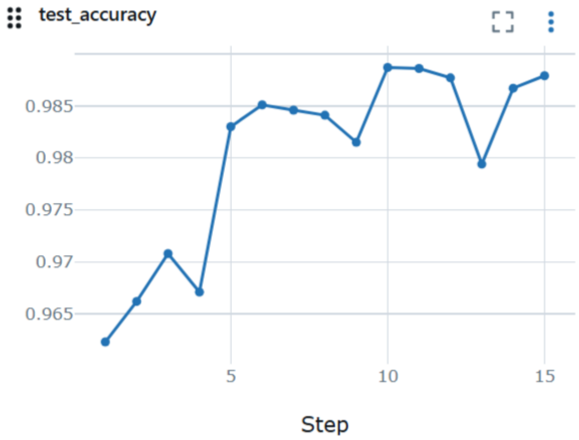
1. Adding Global Average Pooling reduces accuracy as the model becomes extra lighter.
2. We are comparing a 10.9k model with a 6k model. Since we have reduced model capacity, a reduction in performance is expected.

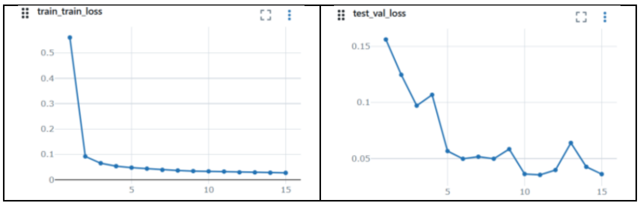
G. Increase the Capacity

Increase model capacity. Add more layers at the end.

Results

1. **Parameters**: 11,994
2. **Best Test Accuracy:** 98.79
3. **Best Training Loss:** 0.0108
4. **Best Test Loss:** 0.0359





Analysis

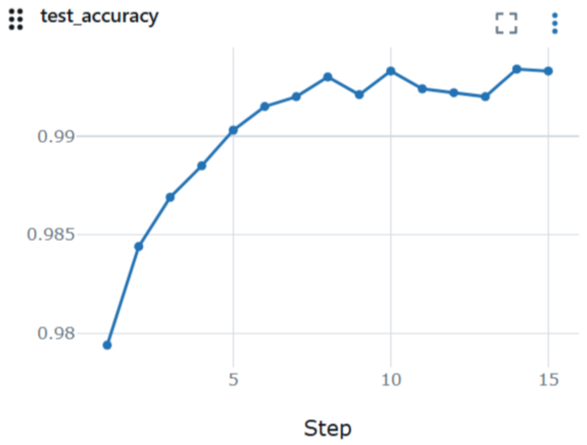
1. The model still shows over-fitting, possibly DropOut is not working as expected. We don't know which layer is causing over-fitting. Adding it to a specific layer wasn't a great idea.
2. Quite Possibly we need to add more capacity, especially at the end.
3. Closer analysis of MNIST can also reveal that just at RF of 5x5 we start to see patterns forming.
4. We can also increase the capacity of the model by adding a layer after GAP!

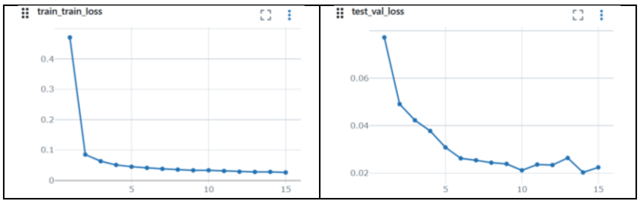
H. Correct MaxPooling Location

- Increase model capacity at the end (add layer after GAP)
- Perform MaxPooling at RF=5
- Fix DropOut, add it to each layer

Results

1. **Parameters**: 13,808
2. **Best Test Accuracy:** 99.33
3. **Best Training Loss:** 0.0092
4. **Best Test Loss:** 0.0223





Analysis

1. We crossed the accuracy of LeNet model.
2. The model is not over-fitting at all.
3. We can further improve the model accuracy by image augmentation and applying proper learning rate schedular.

I. Model Design Summary

- The light-weight model with 13,808 parameters beat the LeNet model with 4,40,812 parameters, which is almost 31 times more.
- The ligh-weight model does not show any overfitting.
- It is not converged so the accuracy can be improved by hyperparameter tuning.
- The accuracy can be defenitly improved with augmentation and proper LR scheduling.

VII. DISTRIBUTED TRAINING

A. Training Methodology

Training is distributed using BigDL-Orca, with GPU-aware configurations to fully utilize hardware acceleration.

Hyperparameters such as learning rate, hidden layer size, batch size, and epochs are defined in params.yaml, enabling flexible experimentation. Data is consumed from Parquet using pandas and PyArrow to avoid Spark overheads in workers.

Few of the hyperparameters are fixed during the model designing for effective comparison of models.

Below table representing the fixed hyperparameters during mode designing:

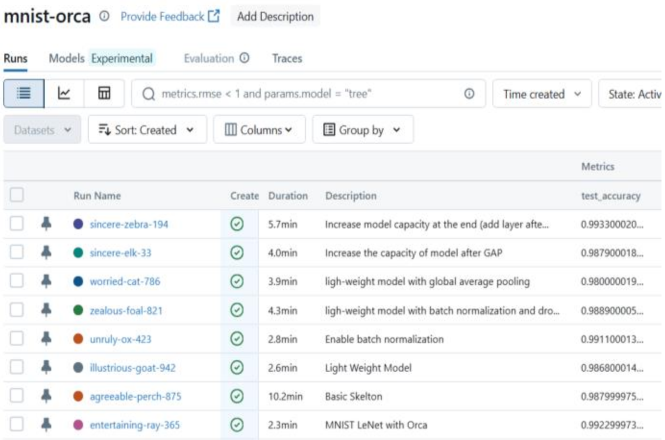
Parameter	**value**
Epochs	15
Batch size	128
Loss criterion	NLLoss
Learning rate	0.01
optimizer	SGD
Momentum	0.9
Weight decay	0.0

- **Model:** LeNet variant with NLLLoss and Adam.
- **Script:** src/train_mnist_orca.py
- **Distributed runtime:** BigDL-Orca (local mode); GPU-aware via workers_per_node.
- **Data loading:** Parquet read with pandas/pyarrow to avoid SparkContext usage in workers.

VIII. MLOPS: MODEL MANAGEMENT

A. Experiment Tracking (MLflow)

Experiment tracking and model lifecycle management are handled by MLflow. Successful runs are logged with parameters, metrics, and model artifacts. Importantly, the system reconstructs MLflow-compliant models from Orca checkpoints, ensuring compatibility with model registry features.



						Metrics
□		Run Name	Create	Duration	Description	test_accuracy
□		sincere-zebra-194	✓	5.7min	Increase model capacity at the end (add layer afte...	0.993300020...
□		sincere-elk-33	✓	4.0min	Increase the capacity of model after GAP	0.987900018...
□		worried-cat-786	✓	3.9min	ligh-weight model with global average pooling	0.980000019...
□		zealous-foal-821	✓	4.3min	ligh-weight model with batch normalization and dro...	0.988900005...
□		unruly-ox-423	✓	2.8min	Enable batch normalization	0.991100013...
□		illustrious-goat-942	✓	2.6min	Light Weight Model	0.986800014...
□		agreeable-perch-875	✓	10.2min	Basic Skelton	0.987999975...
□		entertaining-ray-365	✓	2.3min	MNIST LeNet with Orca	0.992299973...

When a registry backend is available, models are promoted through standard lifecycle stages (new → staging → production → archived), with aliases simplifying governance. In environments without a registry server, the FastAPI service can fall back to the most recent run's artifact, ensuring seamless local development.

B. MLflow Model Registry

- **Script:** `src/register_model.py`
- **Robust selection:** scans best accuracy for a valid `model/MLmodel` artifact.
- **Registry backend:** requires supported URI (e.g., `sqlite:///mlflow.db`, or HTTP server). If tracking store is `file:`, set `mlflow.registry_uri` to a supported backend.
- **Version governance:** sets model aliases (staging/production) by default.

IX. MODEL DEPLOYMENT & TESTING

Deployment is realized through FastAPI. The service can load models either directly from the MLflow registry or from local artifacts, depending on configuration. Predictions are exposed via a REST API, where the client submits a digit image (flattened or 28×28 array), and the service returns the predicted

A. Online Serving (FastAPI)

- **Script:** `src/api/main.py`
- **Loading behavior:**
  - **Preferred:** load `models:/<name>@alias` (or by stage) via registry.
  - **Fallback:** latest run model artifact from tracking store.
- **Endpoint:** POST `/predict` accepts flattened or 28×28 list; preprocesses to numpy array, scales and normalizes, reshapes to (1,1,28,28), calls `model.predict`, returns class index.

B. Web UI

A browser-based UI complements the API, allowing users to draw digits and submit them interactively.

- File: `static/index.html` – canvas for drawing digits;
- downscales to 28×28 grayscale [0,1];
- optional inversion to match training background;
- posts to `/predict`.

B. Docker Deployment

A Docker images are prepared with CUDA runtime support, enabling GPU-accelerated inference in production.

- **Image:** CUDA runtime base installs dependencies and Uvicorn.
- Run with `--gpus all`.
- If using local artifacts, bind-mount `mlruns` and set `MLFLOW_TRACKING_URI=file:/app/mlruns` or use an MLflow server (preferred for registry).

X. FUTURE-PROOFING & ROBUSTNESS

A. Handling Distributional Shift

To extend robustness, the pipeline integrates drift detection through Population Stability Index (PSI). Incoming data distributions are compared against the stored baseline, and alerts are raised if significant divergence is observed.

- **Script:** `src/detect_drift.py` – computes PSI on label distribution vs. baseline.
- Threshold triggers alert file.

B. Automated Retraining

A retraining script orchestrates the complete process—from preprocessing to training to model registration—allowing scheduled or triggered retraining workflows**.**

- Script: `src/retrain_pipeline.py` – orchestrates preprocess → train → register.
- Combine with drift alert or scheduler (cron/systemd) for automation.

C. Resource Optimization

Resource efficiency is emphasized by exposing Spark memory configurations, training batch sizes, and GPU builds as tunable parameters.

- Spark driver memory and partitions configurable; chunk size to control task size.
- Training batch size and epochs affect throughput and memory.
- **GPU support:** install matching CUDA PyTorch build (e.g., cu121 for RTX 4090).

The spark configuration used is:

Spark Configuration	Value
Spark partition	8
Spark driver memory	8gb
Chunk size	2048

XI. RESULT ANALYSIS

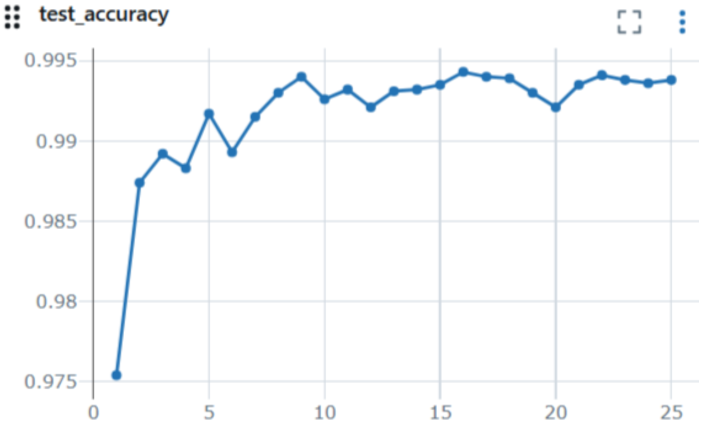
With the step-by-step approach we could able to identify the best model which exceeds the accuracy of LeNet with 30st times less parameters.

The final model having 13.808 parameters achieved 99.33% accuracy in 15 epochs and 99.43% accuracy in 25 epochs.

All the models during model designing is trained for 15 epochs. As we finalize the model, train it for 25 epochs.

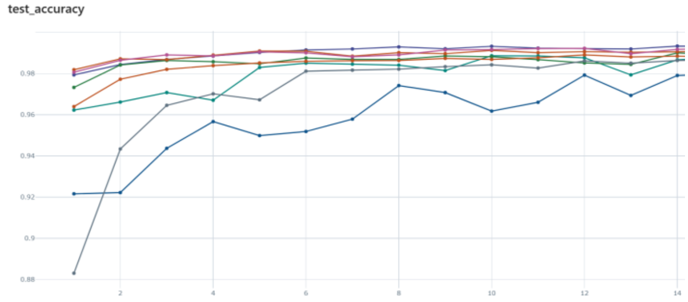
Results

1. **Epochs:** 25
2. **Parameters**: 13,808
3. **Best Test Accuracy:** 99.43 (16th Epoch)
4. **Best Training Loss:** 0.020
5. **Best Test Loss:** 0.019



Test Accuracy Plot

SL No	Description	Duration	Params	Accuracy	Train loss	Test Loss
1	Core Model- LeNet	2.3 min	4,40,812	99.22	0.0028	0.0297
2	Basic Skelton with 1/2nd of LeNet	10.2 min	1,94,884	98.79	0.004	0.0379
3	Make the model lighter with slight change in the Skelton model.	2.6 min	10,790	98.68	0.0122	0.0428
4	Add batch normalization to increase the efficiency.	2.8 min	10,970	99.12	0.0098	0.0298
5	Add dropout	4.3 min	10,970	98.89	0.0087	0.0388
6	Add GAP and remove the last BIG kernel	3.9 min	6,070	98.00	0.0438	0.0715
7	Add more layers at the end	4.0 min	11,994	98.79	0.0108	0.0359
8	Perform MaxPooling at RF=5, add dropout to all the layers	5.7 min	13,808	99.33	0.0092	0.0223

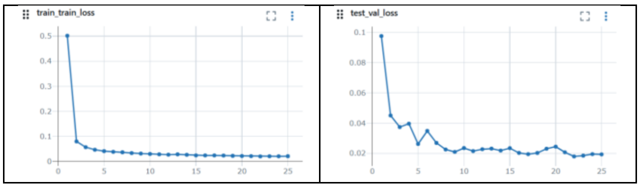


Accuracy Comparison Plot

A. Full Epoch Training

B. Best Model Architecture

```
Layer (type:idx)          Output Shape      Param #
==========================================================
Net                        [1,10]          --
├─Seq:1-1                  [1,16,26,26]      --
│   └─Conv2d:2-1           [1,16,26,26]      144
│   └─ReLU:2-2             [1,16,26,26]      --
│   └─BN2d:2-3             [1,16,26,26]      32
│   └─Dropout:2-4          [1,16,26,26]      --
├─Seq:1-2                  [1,32,24,24]      --
│   └─Conv2d:2-5           [1,32,24,24]      4,608
│   └─ReLU:2-6             [1,32,24,24]      --
│   └─BN2d:2-7             [1,32,24,24]      64
│    └─Dropout:2-8        [1,32,24,24]     --
├─Seq:1-3                 [1,10,24,24]     --
│    └─Conv2d:2-9         [1,10,24,24]     320
├─MaxPool2d:1-4           [1,10,12,12]     --
├─Seq:1-5                 [1,16,10,10]     --
│    └─Conv2d:2-10        [1,16,10,10]     1,440
│    └─ReLU:2-11          [1,16,10,10]     --
│    └─BN2d:2-12          [1,16,10,10]     32
│    └─Dropout:2-13       [1,16,10,10]     --
├─Seq:1-6                 [1,16,8,8]       --
│    └─Conv2d:2-14        [1,16,8,8]       2,304
│    └─ReLU:2-15          [1,16,8,8]       --
│    └─BN2d:2-16          [1,16,8,8]       32
│    └─Dropout:2-17       [1,16,8,8]       --
├─Seq:1-7                 [1,16,6,6]       --
│    └─Conv2d:2-18        [1,16,6,6]       2,304
│    └─ReLU:2-19          [1,16,6,6]       --
│    └─BN2d:2-20          [1,16,6,6]       32
│    └─Dropout:2-21       [1,16,6,6]       --
├─Seq:1-8                 [1,16,6,6]       --
│    └─Conv2d:2-22        [1,16,6,6]       2,304
│    └─ReLU:2-23          [1,16,6,6]       --
│    └─BN2d:2-24          [1,16,6,6]       32
│    └─Dropout:2-25       [1,16,6,6]       --
├─Seq:1-9                 [1,16,1,1]       --
│    └─AvgPool2d:2-26     [1,16,1,1]       --
├─Seq:1-10                [1,10,1,1]       --
│    └─Conv2d:2-27        [1,10,1,1]       160
==================================
Total params: 13,808
Trainable params: 13,808
Non-trainable params: 0
Mult-Adds (M): 3.39
-------------------------------------------------------
Fwd/Bwd size (MB): 0.57
Params size (MB): 0.06
Est. Total size (MB): 0.63
=====================================
```



Loss Plots

C. Model Summary

- The model is not over fitting at all.
- The model touched 99.4% accuracy in 9th epoch, then its oscillating in that range.
- We have to apply augmentation and other hyperparameter tuning to achieve accuracy beyond 99.45

D. BigDL-Orca Log Analysis

Below analysis done as per the `bigdl.log` file generated while model training using bigdl-orca.

Jobs

- Total Jobs: 618
- Duration ranges from 0.29 s to ~33 s, with some heavy jobs around 30+ s.
- Most jobs finish within 2–6 seconds.

Stages

- Total Stages: 618 (one per job in this run).
- Average stage duration: ~6.0 s, but with a high variance.
- Longest stage: 98.4 s, indicating a possible data-heavy or straggler task.

Tasks

- Total Tasks analyzed: 1472.
- Average duration: ~2.6 seconds, but highly variable.
- Fastest task: 52 ms, Slowest task: ~98.3 s (outlier, possibly skew or data imbalance).

Conclusion

- **Performance is generally good** for most jobs/stages (sub-2s median runtime).
- **Outliers exist** (jobs taking 30–98s, tasks ~98s), which could be due to data skew, large broadcast variables, or executor bottlenecks.
- Memory usage seems fine (logs show **~4.1 GiB free** consistently).
- Further optimization should target **long-tail tasks** and **heavier jobs (>30s)** to improve cluster utilization

XII. HOW TO RUN (Quick Reference)

1. Preprocess

```
$python src/preprocess_mnist_spark.py --
params params.yaml
```

2. Train

```
$ export
MLFLOW_TRACKING_URI="file:./mlruns"

$ python src/train_mnist_orca.py --params
params.yaml
```

4. MLflow Server

```
$ mlflow ui --backend-store-uri
file:./mlruns --host 0.0.0.0 --port 5000
# Open http://127.0.0.1:8000 in browser
```

3. Register (registry backend required)

```
$ python src/register_model.py --params
params.yaml
```

4. Online Serve

```
$ uvicorn src.api.main:app --host 0.0.0.0
--port 8000
# Open http://127.0.0.1:8000 in browser
```

5. Drift & Retrain

```
python src/detect_drift.py --params
params.yaml --new-data
data/mnist/test.parquet

python src/retrain_pipeline.py --params
params.yaml
```

XIII. Future Work

Although the system demonstrates a strong foundation, several avenues remain open for exploration.

- Extending experiments to transfer learning or more advanced model techniques such as augmentation, knowledge distillation or LR scheduling could improve performance.
- Introducing feature-level drift detection and explainability dashboards would enhance transparency.
- Full CI/CD integration with infrastructure-as-code and blue/green deployments would further harden the pipeline for enterprise environments.

XIV. CONCLUSION

Building the pipeline highlighted several key lessons.

- Separating preprocessing from training improved modularity and performance.
- Avoiding SparkContext dependencies within workers simplified data handling.
- Ensuring that models were logged as MLflow-compliant PyTorch artifacts was crucial for registry compatibility.
- Designing the serving code with registry fallbacks provided flexibility for both local and production environments.

XV. REFERENCES

[1] BigDL Orca: https://bigdl.readthedocs.io

[2] MLflow: https://mlflow.org/docs

[3] Spark: https://spark.apache.org/docs

[4] FastAPI: https://fastapi.tiangolo.com

[5] DVC: https://dvc.org/doc