MNIST-in-Docker Assignment Report

Ruturaj Tambe

Abstract

This report summarizes the experiment of running MNIST training inside a Docker container. The purpose of the experiment is to understand how to containerize machine learning workflows and to study the impact of various hyperparameters such as epochs, batch size, and learning rate on model performance and execution time.

1 Introduction

The MNIST-in-Docker assignment involves training a neural network on the MNIST dataset within a Docker container environment. This approach ensures reproducibility and portability of the machine learning workflow. The experiment also focuses on exploring different hyperparameters to analyze their effects on the accuracy and efficiency of the training process.

2 Workflow

2.1 Environment Setup

The environment used for this experiment is detailed below:

- Machine: MacBook M3 (Apple Silicon, ARM64)
- Docker version: 28.5.1
- Base image: pytorch/pytorch:latest
- Files used: main.py, requirements.txt, custom Dockerfile

2.2 Steps

The workflow followed these steps:

1. Clone the repository containing the MNIST training code:

   ```
   git clone https://github.com/yourusername/mnist-docker.git
   ```

2. Write a Dockerfile to set up the environment, specifying the base image, copying necessary files, and setting the command to run the training script:

   ```
   CMD ["python", "main.py", "--epochs=10", "--batch_size=32"]
   ```

3. Build the Docker image:

   ```
   docker build --no-cache -t mnist .
   ```

4. Run the Docker container with the desired hyperparameters:

```
docker run -it mnist
```

5. To capture the output and log it for analysis:

```
docker run -it mnist 2>&1 | tee docker-run.out
```

6. Modify hyperparameters such as epochs, batch size, and learning rate by changing the command in the Dockerfile or passing arguments, then repeat the build and run steps to observe effects on accuracy and execution time.

2.3 Files and Paths

Key source files, modified scripts, and outputs used in this assignment:

- Training script (modified): `examples/mnist/main.py`
- Experiment runner: `examples/mnist/mnist_experiments.py`
- Dockerfile: `examples/mnist/Dockerfile`
- Docker run output captures: `examples/mnist/docker-run-5ep.out`, `examples/mnist/docker-run-b256.out`
- Experiment artifacts (default locations when run from `examples/mnist/`):
  - Results CSV: `examples/mnist/mnist_results.csv`
  - Master log: `examples/mnist/mnist_experiments.log`
  - Plots: `examples/mnist/accuracy_vs_epochs.png`, `examples/mnist/time_vs_epochs.png`, `examples/mnist/ac` `examples/mnist/time_vs_batch.png`, `examples/mnist/accuracy_vs_lr.png`, `examples/mnist/time_vs_lr.p`
- Dataset cache (auto-downloaded by `torchvision.datasets.MNIST`): `examples/data/`
- Report sources: `Report/Report.pdf` (compiled)

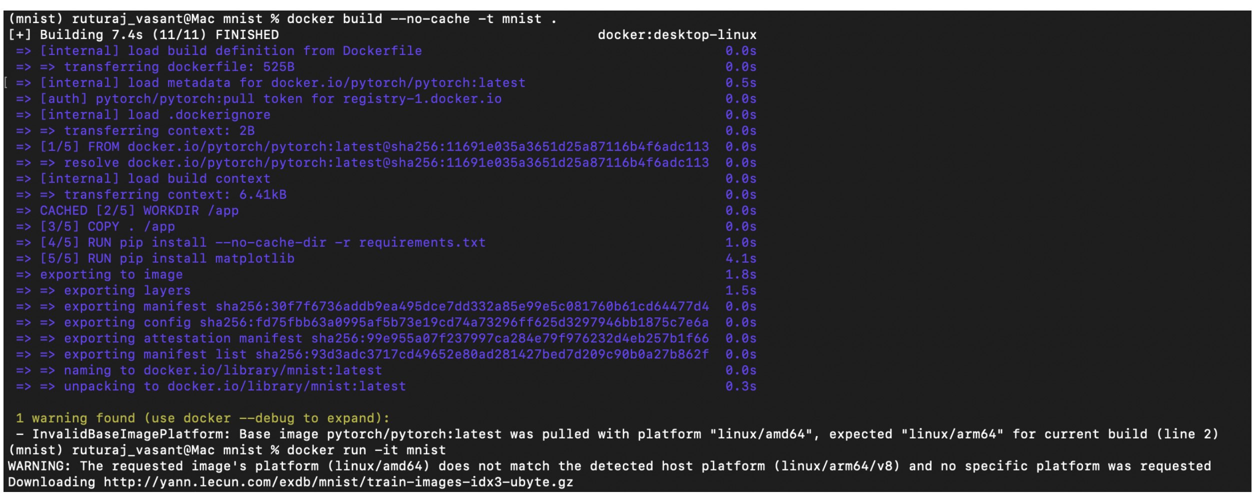


Figure 1: Docker build command using `--no-cache` showing image creation and dependency installation.

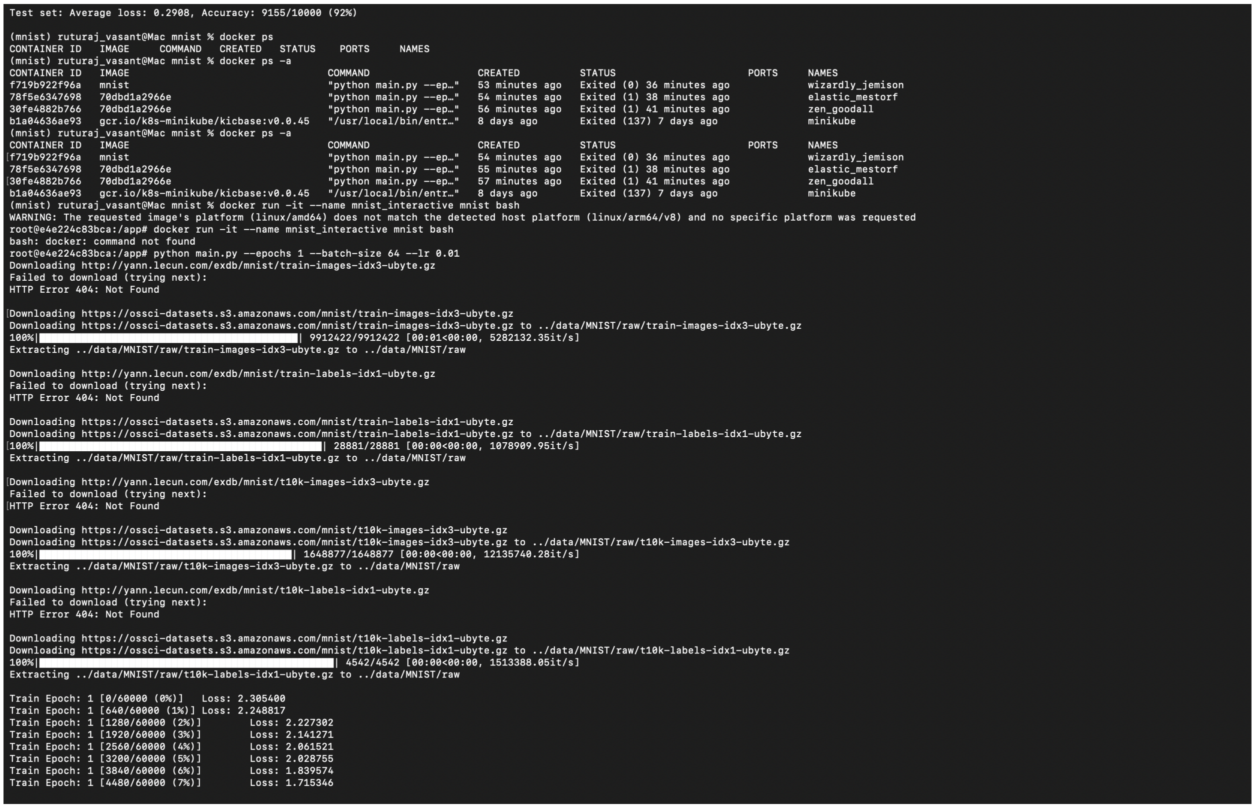


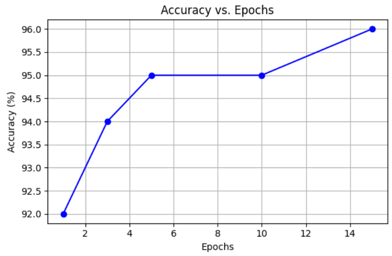
Figure 2: Docker container run showing MNIST data download and initialization.

3 Observations and Results

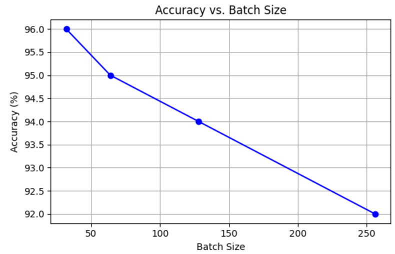
The table below summarizes the results obtained from different hyperparameter configurations. Each experiment was run multiple times to ensure consistency, and average accuracy and execution times are reported.

Category	**Epochs**	**Batch**	**LR**	**Accuracy (%)**	**Time (s)**
Epoch Sweep	1	64	0.01	92.0	65.03
Epoch Sweep	3	64	0.01	94.0	193.68
Epoch Sweep	5	64	0.01	95.0	330.95
Epoch Sweep	10	64	0.01	95.0	634.82
Epoch Sweep	15	64	0.01	96.0	915.94
Batch Sweep	5	32	0.01	96.0	363.99
Batch Sweep	5	64	0.01	95.0	345.56
Batch Sweep	5	128	0.01	94.0	268.55
Batch Sweep	5	256	0.01	92.0	251.71
LR Sweep	5	64	0.001	87.0	323.47
LR Sweep	5	64	0.005	93.0	321.00
LR Sweep	5	64	0.01	95.0	299.51
LR Sweep	5	64	0.05	98.0	306.29

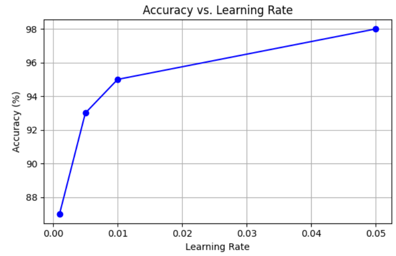
Table 1: MNIST experiment results aggregated from CSV (`examples/mnist/mnist_results.csv`).



(a) Accuracy vs. Epochs

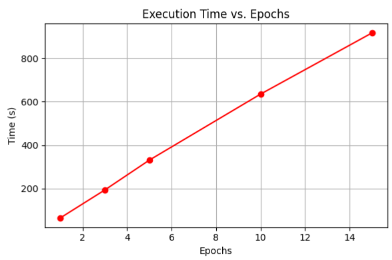


(b) Accuracy vs. Batch Size

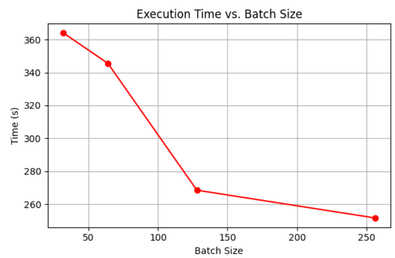


(c) Accuracy vs. Learning Rate

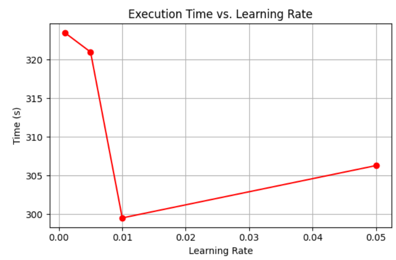
Figure 3: Accuracy for each sweep.



(a) Time vs. Epochs



(b) Time vs. Batch Size



(c) Time vs. Learning Rate

Figure 4: Execution time for each sweep.

Observations. Using the recorded runs, we observe clear trade-offs among epochs, batch size, and learning rate. Across the Epoch sweep (epochs = 1, 3, 5, 10, 15; batch = 64; lr = 0.01), accuracy rises from 92.0% at 1 epoch to 96.0% at 15 epochs, with diminishing returns after roughly 5–10 epochs (95.0% at 5–10). Runtime scales nearly linearly with epochs (approx. 65 s to approx. 916 s), reflecting the expected cost–accuracy trade-off. In the Batch Size sweep at fixed epochs = 5 and lr = 0.01, larger batches reduce wall-clock time (approx. 364 s at 32 to approx. 252 s at 256) but also reduce accuracy (96.0% to 92.0%), illustrating a throughput–generalization trade-off. In the Learning Rate sweep at epochs = 5 and batch = 64, a very small LR underfits (87.0% at 0.001), while higher LRs improve accuracy (93.0% at 0.005, 95.0% at 0.01, 98.0% at 0.05) with similar runtimes (approx. 300–323 s). This suggests an LR "sweet spot" around 0.01–0.05 for faster convergence without instability on this setup; the accompanying plots reinforce these trends.

3.1 Discussion and Analysis

The MNIST experiments were conducted systematically by varying key hyperparameters: number of epochs, batch size, and learning rate. Each of these parameters directly influences the model's learning dynamics and computational efficiency.

Understanding the Parameters:

- **Batch Size:** The number of samples processed before updating model weights. Smaller batches allow more frequent updates, potentially improving convergence at the cost of slower computation, while larger batches increase throughput but may generalize slightly worse.
- **Epochs:** The number of complete passes through the entire training dataset. Increasing epochs generally improves accuracy until the model begins to overfit.
- **Learning Rate:** Controls how large a step is taken in the direction of reducing loss. A learning rate too high can cause oscillations or divergence, while one too low can result in very slow convergence.

Code Comments and Implementation Insights: A few lines of the modified training script are shown below for clarity:

```
# Customizable parameters for tuning
parser.add_argument('--epochs', type=int, default=10, help='Number of training epochs')
parser.add_argument('--batch-size', type=int, default=64, help='Input batch size for training')
parser.add_argument('--lr', type=float, default=0.01, help='Learning rate')
```

These parameters were modified across runs to observe the trade-offs between runtime and accuracy. The accuracy was logged and plotted to visualize training performance under different configurations.

Dockerization Insights: Building and running the training environment inside Docker provided a consistent runtime across multiple trials. Using the `--no-cache` flag ensured the latest dependencies, while container isolation prevented host-environment conflicts. The `amd64` vs `arm64` warning indicated architectural differences between the base image and the host, an important reproducibility observation when running on Apple Silicon systems.

Overall, the experiments highlighted how small parameter adjustments can substantially affect performance, and how Docker serves as a vital tool for managing reproducibility and portability in machine learning workflows.

4 Mapping to Concepts

This experiment demonstrates key concepts in machine learning and software engineering:

- **Containerization Reproducibility:** Encapsulating the training environment in Docker ensures consistent, repeatable runs across machines.
- **Hyperparameter Tuning:** Adjusting epochs, batch size, and learning rate to optimize model performance.
- **Performance Optimization:** Analyzing accuracy vs. computation time to make informed trade-offs.
- **Resource Management:** Understanding how configurations affect execution time and memory usage.
- **Virtualization & Scaling:** The Docker setup mirrors deployment on cloud clusters, connecting classroom concepts of virtualization and resource scaling.
- **Portability Across Architectures:** Accounting for hostimage differences (e.g., `amd64` vs. `arm64`), especially on Apple Silicon.

5 Key Learnings

- Containerizing ML workflows simplifies deployment and collaboration.
- Hyperparameters significantly influence both the accuracy and efficiency of model training.
- Monitoring execution time alongside accuracy helps balance performance and resource consumption.

6 References

- Docker Documentation: `https://docs.docker.com/`
- MNIST Dataset: `http://yann.lecun.com/exdb/mnist/`
- Deep Learning with Python, Francois Chollet, Manning Publications.