

Name: Tai Wan Kim
Net ID: tk3510
University ID: N10254919
Course: Cloud and Machine Learning
Date: November 7th, 2025

HW3: Containerizing AI Workload

I. Abstract

In this assignment, we explore how an AI workload can be containerized for efficient replication, parameter tuning, and performance analysis. We train a model on the MNIST dataset, containerize the training process, and run experiments to see how execution time varies with different parameters. All experiments are conducted in my local machine (no GPU). Code is included in the submission and also available in this GitHub [repository](#).

II. Dataset and Model Architecture

The Modified National Institute of Standards and Technology (MNIST) dataset is a collection of handwritten digits commonly used to train Optical Character Recognition (OCR) models [\[1\]](#). For this assignment, we train a Convolutional Neural Network (CNN) on MNIST with the following architecture:

Input ($1 \times 28 \times 28$) \rightarrow Conv(32, 3×3) + ReLU \rightarrow Conv(64, 3×3) + ReLU \rightarrow MaxPool(2×2) \rightarrow Dropout(0.25) \rightarrow Flatten \rightarrow FC(128) + ReLU \rightarrow Dropout(0.5) \rightarrow FC(10) \rightarrow Log-Softmax

The code has been adapted from an example provided in PyTorch's GitHub repository [\[2\]](#). I made small modifications; these changes are documented in the code.

III. Containerization

We use Docker to containerize training. Containers are lightweight, isolated processes that package an application with its code, runtime, dependencies, and settings. By containerizing our code, we can reliably spin up the same environment with the required ML libraries.

A Docker image (defined by a Dockerfile) is the blueprint for a container. For this assignment, I start from the Python 3.10 base image and install dependencies (torch, torchvision). The container's entry point is configured to run training directly. In section V, I outline the exact steps to reproduce the experiments.

IV. Experiments

We run the container multiple times, each with a different set of parameters. Since we're running on a CPU, we keep the epoch count and batch size relatively low due to resource limits.

- a. *Epoch scaling*. Fix batch size = 128 and use Adadelta with $\text{lr} = 1.0$. Sweep epochs $\{1, 5, 10\}$. We expect elapsed time to grow roughly linearly with epochs, and accuracy to improve early and then plateau (with possible minor overfitting later).

- b. *Learning-rate sweep*. Fix epochs = 5 and batch size = 128; sweep Adadelata $\text{lr} \in \{1.0, 0.5\}$. Elapsed time should be about the same (compute per step doesn't change), but accuracy can vary—too high can destabilize updates, too low can slow convergence.
- c. *Batch-size sweep*. Fix epochs = 5 and $\text{lr} = 1.0$; sweep batch size $\in \{32, 64, 128\}$. On CPU, we expect a trade-off: larger batches reduce the number of steps per epoch but increase work per step. Total epoch time may decrease up to a point and then flatten or reverse.

V. Workflow

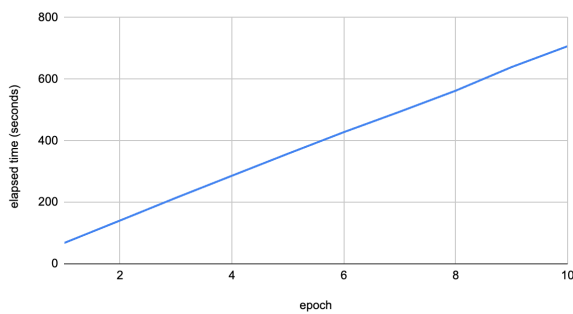
As instructed, I document the steps needed to reproduce the experiments. The exact commands are available in the README included in the submission, or my GitHub repository. To run locally, install and start the Docker daemon (Docker Desktop), navigate to the working directory, build the image from the Dockerfile, and run the container for each trial. Training arguments are passed via the command line. To avoid re-downloading MNIST on every run, we can optionally mount a local directory to persist the dataset (and optionally logs/results). After the experiments, clean up by removing containers and images.

VI. Results

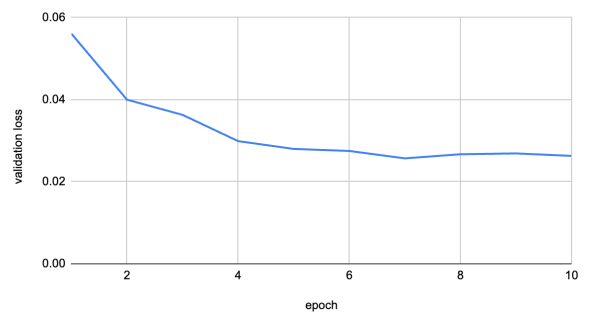
a. Epoch scaling

Epoch	1	5	10
Elapsed Time	62.54s	354.47s	707.03s
Validation Loss	0.0561	0.0280	0.0263
Validation Accuracy	0.9807	0.9908	0.9919

Elapsed Time for Ten Epochs



Validation Loss for Ten Epochs

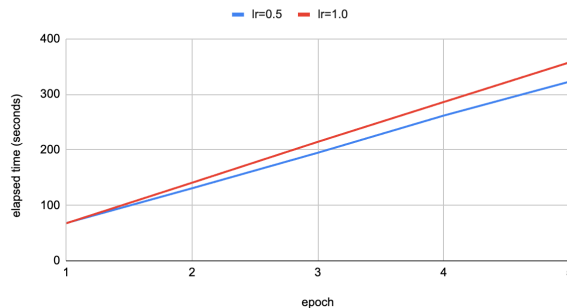


Elapsed time increased approximately linearly with the number of epochs, while average time per epoch was roughly constant. Validation accuracy improved rapidly in the first few epochs and then plateaued (around epochs 5–8).

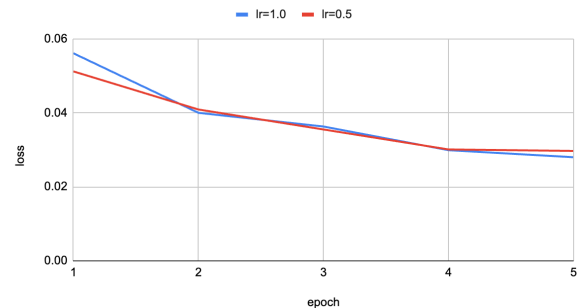
b. Learning-rate sweep

Learning-rate	1.0	0.5
Elapsed Time	354.47s	322.88
Validation Loss	0.0280	0.0297
Validation Accuracy	0.9908	0.9902

Elapsed Time



Validation Loss

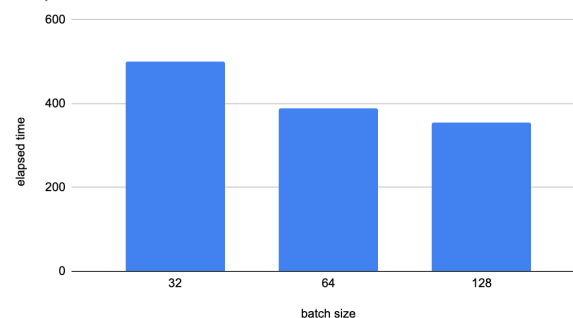


For fixed epochs, time per epoch was similar across learning rates. LR=1.0 achieved lower validation loss slightly earlier than LR=0.5, indicating faster convergence at equal compute.

c. Batch-size sweep

Batch-size	32	64	128
Elapsed Time	500.73s	388.76s	354.47s
Validation Loss	0.0332	0.0280	0.0280
Validation Accuracy	0.9899	0.9905	0.9908

Elapsed Time and Batch Size



Decreasing batch size increased time/epoch because steps per epoch grow ($\lceil N/B \rceil$) and per-step overhead (Python, data loader, optimizer) accumulates. When batch size = 32, the validation accuracy was slightly lower, possibly due to noisier gradients per update.

Overall, though the experiments were small in scale, the outcomes were consistent with the expected functions of batch size, epochs, and learning rate.

VII. Conclusion

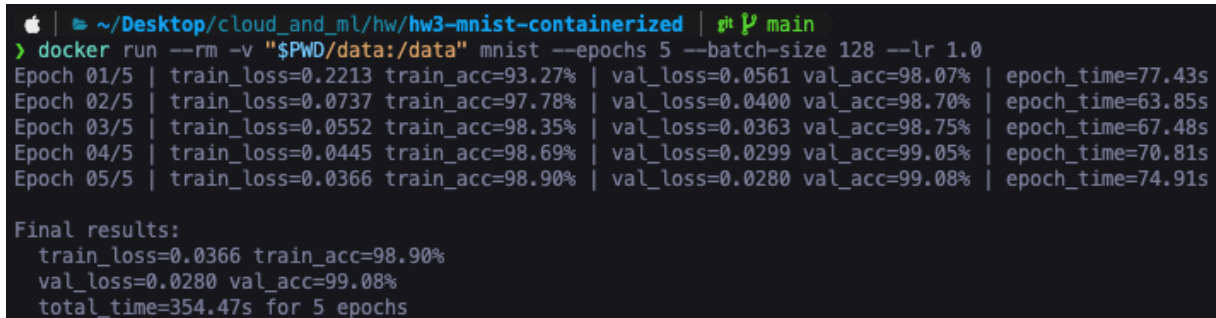
In this assignment, we containerized MNIST training and ran experiments by launching a fresh container for each trial. We observed how elapsed time and loss/accuracy change under different parameter choices. Containerization also made repeated batch runs easy to automate and reproduce.

VIII. References

1. L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," in IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141-142, Nov. 2012, doi: 10.1109/MSP.2012.2211477.
2. PyTorch Foundation, "PyTorch Examples." [Online]. Available: <https://docs.pytorch.org/examples/>

IX. Supplements

Screenshot for docker run.



```

> docker run --rm -v "$PWD/data:/data" mnist --epochs 5 --batch-size 128 --lr 1.0
Epoch 01/5 | train_loss=0.2213 train_acc=93.27% | val_loss=0.0561 val_acc=98.07% | epoch_time=77.43s
Epoch 02/5 | train_loss=0.0737 train_acc=97.78% | val_loss=0.0400 val_acc=98.70% | epoch_time=63.85s
Epoch 03/5 | train_loss=0.0552 train_acc=98.35% | val_loss=0.0363 val_acc=98.75% | epoch_time=67.48s
Epoch 04/5 | train_loss=0.0445 train_acc=98.69% | val_loss=0.0299 val_acc=99.05% | epoch_time=70.81s
Epoch 05/5 | train_loss=0.0366 train_acc=98.90% | val_loss=0.0280 val_acc=99.08% | epoch_time=74.91s

Final results:
train_loss=0.0366 train_acc=98.90%
val_loss=0.0280 val_acc=99.08%
total_time=354.47s for 5 epochs
```