Project 2: Containerization

This report documents the containerization of the DistilBERT GLUE training pipeline from Project 1. The primary challenge was deploying the model training in memory-constrained environments like GitHub Codespaces (2-core machines with 8GB RAM). The project required optimizing Docker images and training code to run efficiently without sacrificing model performance or reproducibility.

Methodology

I approached the containerization systematically through incremental optimization. I began by converting the Jupyter notebook into a modular Python script with full argument parsing. Afterward, I created an initial Dockerfile and identified out-of-memory failures in GitHub Codespaces. To diagnose these issues, I analyzed memory bottlenecks by comparing the implementation with working reference models and profiling resource usage. Based on these insights, I implemented several optimizations: installing a CPU-only version of PyTorch, pre-caching the model during the build process, disabling tokenizer parallelism and removing unnecessary accumulation of training outputs. Finally, I tested the containerized setup across two platforms: my local machine and GitHub Codespace.

Key Optimizations

The following optimizations were critical for successful deployment. I used “python:3.10-slim” as the base image to minimize size and pinned all of my dependencies in “requirements.txt” for consistent builds. At the Docker level, I installed a CPU-only version of PyTorch, reducing the image size by approximately 2.5 GB compared to the CUDA variant. I also pre-cached models and tokenizers during the build phase to prevent runtime memory spikes, disabled tokenizer parallelism by setting “TOKENIZERS_PARALLELISM=false” to lower threading overhead and removed build dependencies after installation to minimize image size. At the code level, I eliminated the “training_step_outputs” list that had been accumulating predictions across epochs, set “num_workers=0” for all DataLoaders to avoid multiprocessing overhead and reduced the default “max_seq_length” to 128 (which I accidentally set to 256 in project 1), effectively halving memory usage. For configuration, I established optimal hyperparameters from Project 1 as defaults while keeping all parameters customizable through command-line arguments. Additionally, I also integrated on this project my Weights & Biases logging to ensure full reproducibility and experiment tracking.

Results

The optimized Docker image successfully ran across the two environments. Memory usage decreased from around 10-12GB (original) to 3.41GB (optimized), enabling deployment on GitHub Codespaces. Model accuracy remained consistent on MRPC validation across all platforms, confirming that optimizations did not compromise quality. Training times varied by platform: 12 minutes on my local CPU and 44 minutes on Codespaces CPU.

Both Docker image runs, one executed locally and one in GitHub Codespaces, produced identical results across all evaluation metrics. The validation accuracy and every recorded loss value were exactly the same for each epoch, indicating complete reproducibility between environments.

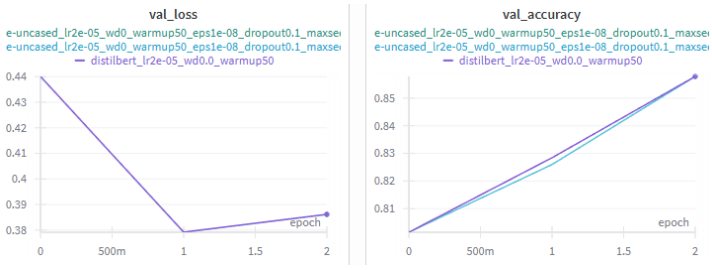

Figure 1: Validation loss and validation accuracy across three epochs of the best run from project 1 and the two docker image runs from project 2.

Furthermore, the two CPU-based Docker runs achieved the same final validation accuracy as the original GPU-based run from Project 1. In the validation accuracy curve, a minor deviation was observed at epoch 2, where the Docker runs reached 0.82598 compared to 0.82843 for the GPU run. By epoch 3, all runs converged to a validation accuracy of 0.85784. This behaviour suggests that the training process was fully deterministic under the defined configuration, with fixed random seeds, consistent data ordering and identical hyperparameters. The equivalence of CPU and GPU results further indicates that the chosen model and optimization procedure are stable across hardware platforms when executed with the same software environment and dependencies.

Environment	*Training Time*	*Final Accuracy*	*Platform*
Local GPU (CUDA)	1min	**0.85784**	NVIDIA RTX 4070 Ti SUPER (16 GB VRAM)
Local CPU	12min	**0.85784**	Intel i9-14900F (24 cores, 32 GB RAM)
GitHub Codespaces (CPU)	44min	**0.85784**	2 vCPUs, 8 GB RAM

Reflection

What Went Well:

The systematic diagnostic approach helped identify root causes rather than just symptoms. Comparing against a working reference implementation revealed that PyTorch CUDA overhead, missing model pre-caching and code-level memory leaks were the real issues, not just hyperparameter choices.

Areas for Improvement:

My initial approach focused too much on hyperparameter tuning (reducing batch_size and max_seq_length) rather than addressing the underlying Docker image and code issues. Earlier profiling and comparison with reference implementations would have saved time me a lot of time.

The Dockerfile could benefit from better layer caching optimization by copying requirements.txt before application code to leverage Docker's build cache during iterative development.

Conclusion

This project successfully containerized the DistilBERT training pipeline for deployment in memory-constrained environments. The key learning was that memory optimization requires a holistic approach: choosing the right base image (CPU-only PyTorch), pre-caching models during build, writing memory-efficient code (no unnecessary accumulation) and setting proper defaults. The result is a production-ready Docker image that runs reliably in GitHub Codespaces while maintaining the model quality achieved in Project 1.

The most valuable insight was learning to distinguish between symptoms (OOM errors) and root causes (CUDA bloat, runtime downloads, memory leaks). This required comparing against working examples and understanding the full stack - from Docker layers to PyTorch internals to training loop design. These are essential skills for deploying ML systems in production.

GitHub Repository

Here is the link to my GitHub repository: https://github.com/FabianDubach/MLOPS.git

Make sure to go to the folder “Project2” to be able to see all necessary files.