UM ASL Detection Project - Final Report

By Sachin Bhat UMID15042530602

1. Overview

This project focuses on building a deep learning-based image classification model capable of recognizing American Sign Language (ASL) alphabets using convolutional neural networks (CNNs). The model is designed to classify 29 categories (A–Z, space, delete, nothing) from static hand gestures. The objective is to provide an accurate, GPU-accelerated detection system suitable for deployment in accessibility tools, educational software, or real-time gesture recognition interfaces.

2. Problem Statement

Manual interpretation of sign language is resource-intensive and inaccessible for many applications, especially in digital environments. Traditional systems rely on hardware-intensive sensors or require specialized datasets and interfaces. The lack of lightweight, camera-only recognition tools limits real-world deployment of ASL recognition systems.

This project aims to build an end-to-end deep learning solution that can:

- Learn to recognize static ASL alphabet gestures from image data
- Generalize across large intra-class variance (lighting, hand orientation, skin tone)
- Run efficiently on GPU-enabled machines using only camera-captured images

3. Project Goals

The primary objective of this project was to design and train a deep learning model capable of accurately recognizing static American Sign Language (ASL) hand gestures from RGB images. The focus was on building a robust, well-generalized image classification model using

convolutional neural networks (CNNs) and a publicly available dataset, while addressing common challenges in preprocessing, dataset management, and model stability.

The specific goals of the project were:

- **1. Design a high-accuracy ASL detection model:**
  Build a CNN capable of learning to distinguish between 29 different hand signs (A–Z, space, delete, nothing) with minimal overfitting and strong generalization.

- **2. Restructure the dataset for full supervised learning:**
  Since the provided dataset lacked a labeled test set, the goal was to restructure the labeled training data into three supervised subsets (train, validation, and test) using a 70/15/15 split, enabling full pipeline training and evaluation.

- **3. Ensure model stability with early stopping and validation monitoring:**
  Implement techniques like early stopping and validation loss tracking to prevent overfitting and automatically select the best model during training.

- **4. Build an adaptable and deployment-friendly pipeline:**
  Create a modular codebase that allows for future enhancements such as real-time ASL detection, integration into applications, or conversion to other frameworks (e.g., ONNX).

While not a formal goal, enabling **GPU acceleration** via WSL2 and CUDA was an important enhancement undertaken during the project to drastically reduce model training time and improve development efficiency for future deep learning tasks.

4. Methodology Summary (Elaborated)

This project implements a **supervised image classification pipeline** using **PyTorch**, tailored specifically for the task of American Sign Language (ASL) letter recognition. The pipeline includes data preprocessing, CNN-based model architecture design, training with regularization techniques, and evaluation using accuracy and loss metrics. Below is a detailed explanation of each component:

● **Dataset: Kaggle ASL Alphabet (29 Classes)**

The dataset used for this project was sourced from Kaggle and contains over **87,000 labeled images** of hand signs corresponding to 29 different classes. These include the 26 English alphabet letters (A–Z) and 3 functional gestures: space, delete, and nothing.

Each class has its own directory, and images are color photos taken with consistent backgrounds and lighting. However, variability in hand shapes, positions, and angles introduces natural challenges for generalization,making it an ideal real-world dataset for deep learning.

● **Input Shape: 64x64 RGB Images**

All input images were resized to a uniform shape of **64×64 pixels** with **3 color channels** (RGB). This standardization was essential to:

- Reduce computational cost
- Ensure compatibility with the model's input layer
- Maintain enough visual detail for classification

Although larger image resolutions could provide more fine-grained features, 64×64 was chosen as a balance between **model efficiency and accuracy**.

● **Preprocessing: Resize, Normalize, Tensor Conversion**

Each image passed through a preprocessing pipeline consisting of:

1. **Resizing** to 64×64 pixels using bilinear interpolation
2. **Conversion to PyTorch tensors** using transforms.ToTensor()
3. **Normalization** with mean = 0.5 and standard deviation = 0.5 across all channels, mapping pixel values from [0, 1] to [-1, 1]

This preprocessing ensures that the input distribution is stable for gradient-based optimization and avoids saturation in activation functions like ReLU.

● **Model: Custom CNN with Adaptive Pooling**

A custom **Convolutional Neural Network (CNN)** was developed for this task, consisting of:

- Two **Conv2D + ReLU + MaxPool** blocks
- A fixed-size feature map generated using AdaptiveAvgPool2d((4, 4)), ensuring consistency regardless of image size
- A **fully connected layer** with 512 neurons followed by dropout for regularization
- A final output layer with 29 neurons (equal to the number of classes), representing raw logits for each class

The use of **Adaptive Pooling** eliminates shape mismatch errors during flattening and ensures architectural flexibility, allowing for future changes in input resolution without breaking the model.

● **Training Split: 70/15/15 (Train/Val/Test)**

While the original ASL Alphabet dataset included a separate test directory, it contained **very few unlabeled samples** and lacked the structured folder format necessary for supervised learning. As a result, it was **not usable for quantitative evaluation** or automated testing in PyTorch.

To maintain a robust and controlled evaluation pipeline, the **entire labeled training set** was manually split into:

- **70% for training**
- **15% for validation** (to monitor model performance and prevent overfitting)
- **15% for testing** (used strictly for final evaluation after training)

This approach ensured:

- A consistent data structure using ImageFolder
- Proper stratification across all 29 gesture classes

- Prevention of data leakage between train, validation, and test sets

By repurposing the full training dataset, the model could be trained and evaluated in a fully **supervised, statistically valid** manner,which would not have been possible using the minimal, unlabeled test set provided.

● Batch Size: 64

The data was loaded in mini-batches of **64 samples** using PyTorch's DataLoader. This size was selected based on:

- Memory availability of the GPU (RTX 2060 Super)
- Stable gradient estimates
- Faster convergence compared to full-batch training

Shuffling was enabled for the training loader to ensure batches were diverse across epochs.

● Optimizer: Adam

The model used the **Adam optimizer**, a popular variant of stochastic gradient descent that adapts learning rates per parameter using first and second moment estimates. Adam was chosen for:

- Fast convergence
- Low sensitivity to learning rate tuning
- Stability with sparse gradients common in CNNs

A learning rate of 0.001 was used, consistent with typical starting points for image classification tasks.

● Loss Function: CrossEntropyLoss

For multi-class classification, the model was trained using CrossEntropyLoss, which combines LogSoftmax and NLLLoss in one function. It is appropriate when the output is a set of unnormalized logits and the targets are class indices (0 to 28 in this case).

This loss function penalizes incorrect predictions more heavily when the model is confident in the wrong direction, encouraging calibrated confidence.

● Metrics: Accuracy, Loss

Model performance was monitored using:

- **Training Loss** (to track convergence)
- **Validation Loss** (for early stopping)
- **Test Accuracy** (as the final performance metric)

5. Solution Workflow

5.1 Dataset Loading and Preprocessing

- The original dataset was a single folder of class-named directories (A-Z, del, nothing)
- The test dataset was discarded due to missing label subfolders.
- The training folder was split into three partitions:
    - 70% training set

- 15% validation set
- 15% test set

Preprocessing Steps

- All images resized to 64x64 pixels
- Normalized to [-1, 1] using mean=0.5 and std=0.5
- Data loaded via torchvision.datasets.ImageFolder
- Torch DataLoader used with shuffling and batch processing

5.2 Model Architecture and Training

A simple CNN architecture was built with:

- 2 convolutional layers
- 2 max pooling layers
- Adaptive average pooling to ensure consistent flattening size
- Fully connected layers with dropout
- Final output layer of 29 logits

Architecture:

Input: (3, 64, 64)

→ Conv2d (32 filters) → ReLU → MaxPool

→ Conv2d (64 filters) → ReLU → MaxPool

→ AdaptiveAvgPool2d (4x4)

→ Flatten → Dropout → FC(512) → ReLU → FC(29)

Training Parameters:

- Epochs: 20 (with early stopping after 3 stagnant validation losses)
- Loss Function: CrossEntropyLoss
- Optimizer: Adam
- Progress display: tqdm integrated

5.3 Evaluation and Performance Metrics

Metric	Result
Test Accuracy	**99.74%**
Training Loss	**0.0250**
Validation Loss	**0.0063**

- Accuracy was calculated using sklearn.metrics.accuracy_score
- Early stopping saved the best model checkpoint
- Model was saved using torch.save() for future reuse

6. Tools and Frameworks

- **Language:** Python 3.10

- **Libraries:** PyTorch, torchvision, scikit-learn, tqdm
- **Environment:** WSL2 on Ubuntu 22.04, RTX 2060 Super (CUDA)
- **IDE:** VS Code (Linux Remote Extension)

7. Project Advantages

GPU-Accelerated Training Using Consumer Hardware

The complete model training pipeline was eventually executed on an **RTX 2060 Super GPU** using **WSL2 with CUDA support**, which led to a **massive reduction in training time** and made deep learning development far more efficient.

Initially, training was attempted on an **Apple M2 MacBook Air** using **TensorFlow on CPU**. Under this setup, each epoch took approximately **42 minutes**, leading to a total training time of over **14 hours** for 20 epochs,a rate that severely limited the speed of iteration and experimentation.

Switching to a **desktop machine running a Ryzen 5 5500 CPU** resulted in only a marginal improvement, bringing the per-epoch time down to around **35 minutes**, which was still impractical for any meaningful development cycle.

The final breakthrough came after transitioning to **GPU-accelerated training using CUDA on the RTX 2060 Super** via WSL2 and PyTorch. With this setup, training time **dropped drastically to approximately 1.5 minutes per epoch**,a **20× improvement** over the original setup. This speedup enabled real-time monitoring, rapid experimentation, and efficient use of early stopping techniques, allowing the model to be refined and finalized much more effectively.

This experience demonstrates that **consumer-grade GPUs**, when properly configured, offer a powerful and accessible solution for training deep learning models,even those with large datasets,without the need for high-end cloud infrastructure.

Model Architecture Adapts Automatically to Input Shape

The CNN architecture uses **AdaptiveAvgPool2d** to automatically flatten feature maps to a fixed output size, regardless of input image dimensions. This eliminates hardcoding of flatten sizes and prevents shape mismatch errors during training. As a result, the model is **flexible and robust** to changes in image resolution,a key advantage for scalability and deployment in environments with varying camera inputs or preprocessing settings.

Discarded Incomplete Test Set in Favor of Controlled Dataset Split

While the dataset provided a separate test directory, it lacked class labels and consistent structure, making it unsuitable for supervised evaluation. Rather than attempting to adapt an incomplete set, the project repurposed the full training dataset by applying a **70/15/15 split for train, validation, and test sets**. This approach ensured statistical soundness, eliminated potential data leakage, and provided a **reliable framework for model validation** and early stopping.

Modular Script Enables Integration into Notebooks, Web Apps, or Further Research Pipelines

The project was developed using clean, modular Python scripts based on PyTorch, making it highly **reusable and extensible**. The training script can easily be embedded into:

- Jupyter Notebooks for educational use

- Web frameworks (e.g., Flask, Streamlit) for real-time inference demos
- Research workflows for transfer learning or benchmarking

This modularity supports rapid adaptation to future goals, such as webcam integration, live ASL detection, or deployment as an assistive accessibility tool.

Achieved State-of-the-Art Test Accuracy with Minimal Complexity

The model achieved a **test accuracy of 99.74%**, while maintaining a simple architecture with just two convolutional blocks and minimal tuning. This underscores the effectiveness of well-selected preprocessing, data management, and early stopping,proving that **you don't need complex architectures or massive compute resources** to reach high performance. This result reflects both the **strength of the modeling strategy** and the quality of the implementation.

8. System Workflow and Integration

1. **Input:**
    - Static hand gesture images (RGB)
    - Dataset folder with 29 labeled subfolders
2. **Preprocessing:**
    - Resize, normalize, batch-load
3. **Model Inference:**
    - CNN forward pass → softmax over 29 classes
4. **Output:**
    - Class prediction (e.g., “B”, “space”, “del”)

9. Execution Process

1. Dataset path set to Windows directory via WSL (/mnt/c/...)
2. Code run from Ubuntu terminal using:

   source tf-venv/bin/activate

   python asl_cnn_tqdm.py

3. Epoch-wise training logs shown via tqdm
4. Early stopping saved best-performing model
5. Final test accuracy reported after evaluation

10. Obstacles and Resolutions

1. **CUDA nvcc Not Found in WSL2**

   One of the first technical barriers encountered was the inability to access nvcc, the NVIDIA CUDA compiler, within the WSL2 (Ubuntu) environment. Initial attempts to install the CUDA Toolkit using apt failed due to unmet dependencies such as libtinfo5. These dependency conflicts, combined with inconsistent support for certain CUDA versions on WSL, made it challenging to configure GPU computation. The issue was ultimately resolved by following NVIDIA's official guidance for CUDA support in WSL2. A compatible Game Ready driver with WSL2 support was already installed on the Windows host. Inside WSL2, CUDA libraries were configured manually through local toolkit installation, and GPU availability was confirmed using nvidia-smi.

2. **TensorFlow Failed to Detect GPU**

   Despite CUDA being installed correctly, TensorFlow running under WSL2 was unable to

detect the GPU, consistently returning zero available GPUs. Numerous attempts were made to install a GPU-compatible version of TensorFlow, and various CUDA/cuDNN combinations were tested. However, TensorFlow remained unable to utilize the GPU. To overcome this, the framework was switched from TensorFlow to PyTorch, which immediately recognized the available GPU via torch.cuda.is_available() without additional configuration. This switch not only solved the compatibility issue but also streamlined the rest of the model development process.

3. **Shape Mismatch in Flattening Layer (mat1 and mat2 cannot be multiplied)**

   During model development, a critical runtime error occurred involving a mismatch between the size of the flattened feature maps and the expected input to the fully connected layer (fc1). This happened particularly after experimenting with higher image resolutions (e.g., resizing inputs to 200×200 pixels), which changed the output dimensions of the convolutional layers. Attempts were made to calculate the flatten size dynamically using dummy tensors, but these led to further complications when model and dummy inputs resided on different devices (CPU vs. GPU). This problem was permanently resolved by introducing AdaptiveAvgPool2d, which ensured that the feature map output was consistently sized at 4×4, regardless of the input resolution. This eliminated shape mismatches and simplified the model architecture.

4. **Unstructured Test Dataset**

   The original dataset included a separate test directory, but it lacked labeled subfolders, making it incompatible with torchvision.datasets.ImageFolder, which expects class-wise

folder structures. Additionally, the test set contained very few images and was unsuitable for any meaningful evaluation. To address this, the test set was discarded, and the labeled training dataset was manually split into three statistically balanced subsets: 70% for training, 15% for validation, and 15% for testing. This approach preserved class distribution, enabled proper validation and early stopping, and ensured that test results reflected the model's generalization performance.

5. **No Visible Training Progress During Epochs**

   Upon running the training script, it initially appeared to freeze or stall after displaying dataset statistics. This was due to the large size of the dataset (over 60,000 training images), which caused long delays during epoch processing,with no intermediate feedback to the user. The absence of print statements or indicators led to confusion about whether training was functioning correctly. This issue was addressed by integrating the tqdm library, which wrapped the training loop with a dynamic progress bar showing real-time updates on batch completion and loss values. This made it significantly easier to monitor and debug the training process.

6. **Apparent Freeze During First Full Training Run**

   On the first complete training attempt, the model seemed to freeze indefinitely after loading the dataset. No errors were raised, and nothing indicated that training was progressing. Upon closer investigation using nvidia-smi, it was discovered that the model was training,but extremely slowly due to CPU-only computation. Benchmarking was performed across different hardware setups to better understand the bottleneck: on the

MacBook Air (M2, TensorFlow CPU), training took approximately 42 minutes per epoch; on the Ryzen 5 5500 desktop (TensorFlow CPU), it improved slightly to 35 minutes per epoch. Only after switching to GPU-based training with PyTorch and CUDA on an RTX 2060 Super was a substantial speed-up achieved, reducing training time to approximately 1.5 minutes per epoch,a ~20x performance improvement.

7. **Continued Shape Mismatch Despite Resizing Inputs**

   Even after resizing input images back to 64×64 to match the model's initial expectations, errors related to mismatched input shapes persisted. These were due to inconsistencies in flattening logic and how dummy inputs were used to calculate the number of features entering the fully connected layer. The model's initialization process was incorrectly mixing CPU and GPU tensors, leading to runtime errors. This issue was conclusively solved by removing manual flattening logic altogether and relying entirely on AdaptiveAvgPool2d to standardize the output size of the final convolutional layer, thereby eliminating the need for flatten size estimation.

8. **Missing Python Packages and pip Inside WSL2**

   During initial setup in the WSL2 environment, several essential packages,including pip, torchvision, and tqdm,were either missing or improperly configured. As a result, import errors and command-not-found issues prevented progress. The problem was resolved by installing python3-pip via apt, and then activating the virtual environment (tf-venv). Within the environment, required libraries were installed using pip install, ensuring that

all dependencies were correctly isolated and accessible to the Python interpreter being used for training.

11. Observations

1. The final CNN model demonstrated strong generalization capabilities across all 29 ASL classes, as evidenced by the **very low validation loss of 0.0063**. This indicated that the model not only learned the training data effectively but also maintained performance on unseen validation samples, with minimal signs of overfitting.
2. The integration of **AdaptiveAvgPool2d** proved to be a key architectural improvement. It eliminated all issues related to flattening mismatches by dynamically adjusting the spatial dimensions of the convolutional output to a fixed size, regardless of input image resolution. This allowed for greater flexibility in preprocessing choices and significantly reduced the risk of shape-related runtime errors.
3. Training performance improved drastically after enabling GPU acceleration. When benchmarked against CPU-based training on both the Apple M2 and Ryzen 5 5500 systems, the GPU-powered pipeline on an **RTX 2060 Super achieved a ~20x speedup**, reducing per-epoch training time from over 35–42 minutes to just **1.5 minutes per epoch**. This acceleration enabled faster experimentation and more frequent tuning iterations, which contributed to the final model's high performance.
4. Lastly, working with **PyTorch** proved to be both efficient and developer-friendly. Compared to TensorFlow, PyTorch offered greater transparency in model behavior, easier debugging, and smoother integration with GPU resources. Its dynamic computation graph

and modular design made it ideal for iterative development, particularly in a research-focused setting like this internship project.

12. Future Enhancements

1. **Integrate Real-Time ASL Detection from Webcam Input**

   A practical next step would be to enable real-time classification by capturing video frames from a webcam using OpenCV and passing them through the trained CNN model. This would provide immediate visual feedback for users and demonstrate the model's potential for interactive applications. The implementation would primarily involve integrating live frame capture and basic image preprocessing.

2. **Wrap the Model in a Simple Web Interface Using Streamlit**

   To make the model more accessible and user-friendly, it could be deployed as a basic web app using Streamlit. This would allow users to upload images or test webcam inputs directly from a browser interface. Streamlit is well-suited for beginners and requires minimal web development experience, making it a realistic enhancement within your current scope.

3. **Implement Per-Class Accuracy and Error Analysis Tools**

   Adding a module for evaluating **class-wise accuracy** (e.g., confusion matrix, misclassified examples) would offer more insight into which letters the model struggles with. This would help guide future improvements and improve the model’s

interpretability. Python libraries like sklearn and matplotlib can be used to visualize these metrics easily.

4. **Train with Data Augmentation for Better Generalization**

   To make the model more robust to real-world variations, simple data augmentation techniques such as random rotations, flips, brightness adjustments, and background changes could be added during training. This would simulate more realistic scenarios and improve the model's performance on varied input conditions,especially useful if transitioning to real-time detection.