ASL Alphabet Classification

1. Dataset Preparation

This project utilizes the ASL Alphabet dataset, which contains 29 classes representing the letters A through Z, as well as the commands for Space, Delete, and Nothing. The dataset is structured with images organized into separate directories for each class, containing RGB images in JPG or PNG format, and includes pre-defined train/test splits.

Our preprocessing pipeline, implemented in data_loader.py, manages the entire data handling process. It loads images using the Python Imaging Library (PIL) and resizes them to a configurable input resolution (112×112, 128×128, or 224×224). The images are normalized using the mean and standard deviation from the ImageNet dataset. To improve model generalization and combat overfitting, we employed several data augmentation techniques, including horizontal flipping, colour jittering, and random affine transformations involving minor rotations and translations. Finally, the pipeline converts images to the PyTorch-compatible CHW tensor format and creates balanced train/validation splits to ensure robust model training.

2. Model Development

We developed two model variants to address different deployment needs. The primary model, LightSignLanguageCNN, is optimized for edge devices. Its architecture consists of four convolutional blocks, each containing a Conv layer, followed by Batch Normalization, a ReLU activation function, and a MaxPool layer. The network progressively increases channel depth from 3 to 128. A key feature is the use of an AdaptiveAvgPool2d(7x7) layer, which makes the model independent of the input image resolution. The classifier head consists of fully connected layers that reduce the 128×7×7 features down to 256 hidden units before the final 29-class output, utilizing a 40% dropout rate for regularization. This entire design results in a lightweight model of approximately 850,000 parameters.

For reference, we also created a standard CNN with larger layers, prioritizing higher accuracy without the constraints of edge deployment. Core design decisions across both models included:

- BatchNorm: For stable and accelerated training.
- ReLU Activation: For computational efficiency and gradient stability.
- Adaptive Pooling: To handle variable input sizes flexibly.
- Dropout: To significantly reduce overfitting.
- Lightweight Design: To ensure low-latency inference on resource-constrained devices.

3. Model Training Process

The model was trained with the following configuration: an SGD optimizer with a momentum of 0.9, an initial learning rate of 0.01 (with step decay), CrossEntropyLoss, a batch size of 32, and training for 20-30 epochs with early stopping. The training workflow, managed by train.py, handles dataset loading, model initialization, the forward pass, loss calculation, and backpropagation. Validation is performed after each epoch, and the system saves the best checkpoint, which includes the model state dict, optimizer state, class names, and metadata, while logging all performance metrics.

Through extensive experimentation, we determined that a 128×128 input resolution offered the best trade-off between accuracy and inference speed. Data augmentation proved critical in reducing overfitting. Several key challenges were identified and resolved during training:

Challenge	Solution
Overfitting	Addressed through extensive data augmentation, a 40% dropout rate, and early stopping.
Class Confusion (e.g., M/N, S/A)	Improved model discrimination using rotation and perspective augmentations.
Slow CPU Training	Mitigated by using the lighter architecture and enabling mixed-precision training on GPUs.
Input-Size Flexibility	Resolved by using AdaptiveAvgPool2d to maintain consistent internal feature geometry.

4. Model Optimization & Conversion

For efficient deployment, the model was converted to the ONNX format. The export pipeline in convert_to_onnx.py generates a dummy input based on the target resolution and performs the export with constant folding optimizations. A robust fallback sequence is in place: if the primary method fails, it disables constant folding and attempts to export again using a TorchScript trace.

We further optimized the model by applying dynamic quantization to INT8 precision. This post-training quantization (PTQ) process used representative samples to calibrate the model. The benefits were substantial:

- ~75% reduction in model size.
- 30-40% improvement in inference speed.
- <1% drop in accuracy.

On-Device Performance

On-device performance

MCUs

		EON COMPILER		TFLITE	
DEVICE	LATENCY	RAM	ROM	RAM	ROM
Low-end MCU	55,681 ms.	667.9K	6.2M	1.2M +515.5K	6.3M +18.8K
High-end MCU	1,262 ms.	667.9K	6.3M	1.2M +515.7K	6.3M +20.6K
+ AI accelerator	211 ms.	667.9K	6.3M	1.2M +515.7K	6.3M +20.6K

Microprocessors

DEVICE	LATENCY	MODEL SIZE
CPU	43 ms.	6.2M
GPU or accelerator	8 ms.	6.2M