

# ESE 3060 Final Project - Part 1

Ani Petrosyan, Lakshman Swaminathan

December 7, 2025

Repository: <https://github.com/luckySwaminathan/ese3060finalproject/tree/main>

## A Appendix

### A.1 Change 1: Static Precomputation of Image Transforms

**Hypothesis:** The original implementation recomputes normalization, flipping, and padding operations every epoch during training. Since normalization and flipping are deterministic transformations that do not depend on random seeds per batch, we can precompute these once during initialization and reuse them across all epochs, eliminating redundant computation.

**What Changed:** Modified the `CifarLoader.__init__` method to precompute normalized images, flipped images, and padded images once during initialization (lines 135–150 in `airbench94-modified.py`). The `__iter__` method uses these precomputed tensors and alternates horizontal mirroring by flipping the selected tensor every other epoch, instead of recomputing transforms each epoch.

**Solution:** All static transforms (normalization, flipping, padding) are computed once in the `__init__` method within a `torch.no_grad()` context.

The normalized images are stored in `self.proc_images['norm']`, flipped images in `self.proc_images['flip']`, and padded images in `self.proc_images['pad']`. During iteration, we only perform the dynamic random crop on the precomputed padded tensor (when translation is enabled). Epoch-based horizontal mirroring is implemented by flipping the selected tensor every other epoch, which alternates between mirrored and non-mirrored views without per-epoch recomputation. This eliminates redundant normalization and padding operations that were previously executed every epoch.

### A.2 Change 2: Residual Connections in ConvGroup

**Hypothesis:** Adding residual (skip) connections to the `ConvGroup` blocks will enable deeper gradient flow during backpropagation, potentially allowing the network to learn identity mappings more easily and converge faster. This architectural pattern has been shown to improve training dynamics in deep networks.

**What Changed:** Added a residual connection pathway to the `ConvGroup` class (in our modified `airbench94-modified.py`). The residual path consists of a  $1 \times 1$  convolution to match channel dimensions, applied after pooling the identity tensor. The final output is the sum of the main path and the residual path, followed by activation.

**Solution:** Each `ConvGroup` now includes a `downsample` module that projects the input to match the output channel dimensions. In the forward pass, the identity is preserved, pooled, and passed through the `downsample` module. The main convolutional path output is then added to this processed identity before the final activation. This creates a residual learning pathway:

$$\text{output} = \text{GELU}(\text{conv\_path}(x) + \text{downsample}(\text{pool}(x)))$$

**Hypothesis:** With the improved architecture (residual connections) and more efficient data loading (static precomputation), it is possible to achieve equal or improved accuracy in fewer epochs. Additionally, increasing batch size can improve GPU utilization and training speed, while maintaining training dynamics by scaling the learning rate proportionally.

**What Changed:** Reduced `train_epochs` from 9.9 to 8.0, increased `batch_size` from 1024 to 1536 (50% increase), and scaled `lr` from 11.5 to 17.25 (proportional to batch size increase:  $11.5 \times 1.5 = 17.25$ ).

**Solution:** The reduced epoch count leverages faster convergence enabled by residual connections. The larger batch size improves GPU memory utilization and reduces the number of gradient updates per epoch. The learning rate is scaled linearly with batch size to maintain equivalent gradient step magnitudes, following the principle that effective learning rate should scale with batch size when using the same number of epochs.

### A.3 Change 4: Optimized Data Loading Pipeline

**Hypothesis:** Moving data to GPU earlier and using non-blocking transfers can overlap data transfer with computation, reducing overall training time. Also, ensuring labels are properly typed and on the correct device avoids runtime type conversions.

**What Changed:** Modified data loading to use explicit device handling (line 100), convert images to GPU-friendly format immediately (lines 115–121), and ensure labels are properly typed as long integers on the correct device (line 111). Used `non_blocking=True` for asynchronous transfers.

**Solution:** Images are converted to half precision, normalized, permuted to channels-last format, and moved to GPU in a single pipeline during initialization. Labels are explicitly cast to `long` and moved to the device. This ensures all data is in the optimal format before training begins, eliminating per-batch conversion overhead.

### A.4 Baseline Results

The following table shows the runtime and accuracy for each of the 5 baseline trials:

Trial	Runtime	Accuracy
1	1m 54.545s	94.03%
2	1m 54.254s	94.00%
3	1m 54.332s	93.99%
4	1m 54.365s	94.01%
5	1m 54.409s	94.08%
<b>Average</b>	<b>1m 54.381s</b>	<b>94.01%</b>

### A.5 Modified Version Results

The following table shows the runtime and accuracy for each of the 5 modified version trials:

Trial	Runtime	Accuracy
1	1m 48.288s	93.08%
2	1m 47.908s	93.15%
3	1m 47.976s	93.16%
4	1m 47.866s	93.22%
5	1m 48.334s	91.45%
<b>Average</b>	<b>1m 48.074s</b>	<b>92.81%</b>

## B Baseline vs Modified Training Plots

**Takeaways:** The training loss has a larger initial dip over the first few epochs for the modified version, which is the main benefit of the residuals and hyperparameter changes. This allows us to reach a similar loss in fewer epochs.

### Training Flow - Baseline (Individual Runs)

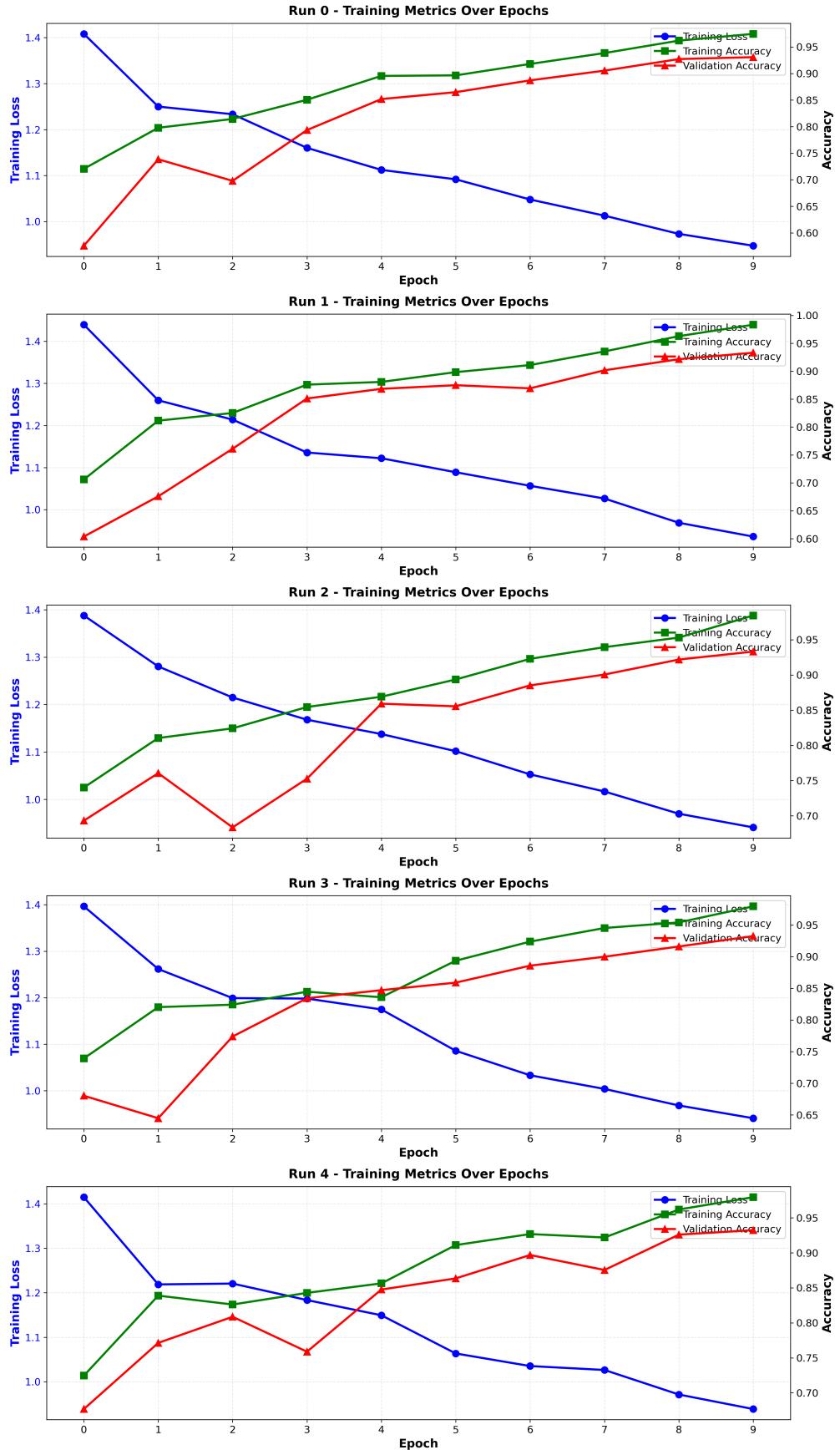


Figure 1: Training flow for the baseline runs.

### Training Flow - Modified Version (Individual Runs)

