ESE 3060 Final Project - Part 1

Ani Petrosyan, Lakshman Swaminathan

December 7, 2025

1 Hypothesis

Our goal is to reduce the wall-clock training time of `airbench94.py` on CIFAR-10 while staying as close as possible to the original 94.0% test accuracy. Our hypothesis is that (1) precomputing deterministic image transforms, (2) adding residual connections to the convolutional blocks, (3) tuning the batch size, learning rate, and number of epochs, and (4) optimizing data movement to the GPU will shorten training time by around 15–20% while keeping test accuracy within about 0.5–1.0 percentage points of the 94.0% baseline.

2 Methodology

To test this hypothesis, we started from the provided `airbench94.py` baseline and made a small set of targeted changes aimed at either (a) removing redundant computation or (b) improving convergence per unit time:

- **Static precomputation of image transforms.** Normalization, flipping, and padding are computed once in `CifarLoader.__init__` and stored in `self.proc_images`, instead of being recomputed every epoch. During training, the loader only does the remaining random crop on precomputed padded images.
- **Residual connections in `ConvGroup`.** Each `ConvGroup` now has a residual path: the pooled input is passed through a 1×1 convolution "downsample" layer and added back to the main convolutional path before the GELU activation. This is meant to improve gradient flow and let the network learn near-identity mappings more easily, so it can converge in fewer effective epochs.
- **Hyperparameter changes.** We increased the batch size from 1024 to 1536 and scaled the learning rate from 11.5 to 17.25 (1.5×, matching the batch-size change). We reduced the number of epochs from 9.9 to 8.0 to trade a small loss in optimization for faster overall runtime. These changes are intended to better use GPU memory and reduce per-epoch overhead while keeping training dynamics similar.
- **Optimized data loading and device transfers.** Images are converted to the final dtype/layout and moved to the GPU once, with non-blocking transfers and labels cast to `long` on the correct device. This removes per-batch conversion and host-to-device overhead that does not contribute to learning.

To evaluate these changes, we ran the baseline and the modified script on the same GPU and recorded (a) final CIFAR-10 test accuracy and (b) wall-clock runtime. The detailed experiment log, including commands and multiple runs, is provided in the separate appendix.

3 Results & Conclusions

In 5 runs, the baseline achieved an average of 94.01% test accuracy with a runtime of roughly 1m 54.381s (real category in the logs). Over 5 runs, the modified version ("modified" variant) ran in an average of 1m 48.074s, which is roughly an 5.3% reduction in wall-clock time, but its accuracy dropped slightly to around 92.81% (a loss of about 1.2 percentage points compared to the baseline).

These results partially support the hypothesis: the combination of precomputed transforms, residual connections, and hyperparameter / data-loading tweaks clearly reduces training time, and the accuracy remains very close to the original 94.0% target. However, the small accuracy drop suggests there is still a trade-off between speed and performance. In future iterations, we would tune the number of epochs and learning rate more finely (or add mild regularization changes) to see if we can recover the lost ∼1.2 percentage points while keeping most of the ∼5.3% speed-up.

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 now selects from these precomputed tensors instead of recomputing them 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 operation on the precomputed padded tensor, and apply epoch-based flipping by selecting from precomputed tensors. 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 1×1 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:

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

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%
Average	**1m 54.381s**	**94.01%**

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%
Average	**1m 48.074s**	**92.81%**